Parallel techniques for construction of trees and related problems

By

Teresa Maria Przytycka

Magister, Warsaw University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August, 1990

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date September 17, 1990

DE-6 (2/88)

Abstract

The concept of a tree has been used in various areas of mathematics for over a century. In particular, trees appear to be one of the most fundamental notions in computer science. Sequential algorithms for trees are generally well studied. Unfortunately many of these sequential algorithms use methods which seem to be inherently sequential. One of the contributions of this thesis is the introduction of several parallel techniques for the construction of various types of trees and the presentation of new parallel tree construction algorithms using these methods. Along with the parallel tree construction techniques presented here, we develop techniques which have broader applications.

We use the Parallel Random Access Machine as our model of computation. We consider two basic methods of constructing trees:*tree expansion* and *tree synthesis*.

In the *tree expansion method*, we start with a single vertex and construct a tree by adding nodes of degree one and/or by subdividing edges. We use the parallel tree expansion technique to construct the tree representation for graphs in the family of graphs known as cographs.

In the *tree synthesis method*, we start with a forest of single node subtrees and construct a tree by adding edges or (for rooted trees) by creating parent nodes for some roots of the trees in the forest. We present a family of parallel and sequential algorithms to construct various approximations to the Huffman tree. All these algorithms apply the tree synthesis method by constructing a tree in a level-by-level fashion. To support one of the algorithms in the family we develop a technique which we call the *cascading sampling technique*.

One might suspect that the parallel tree synthesis method can be applied only to trees of polylogarithmic height, but this is not the case.We present a technique which we call the *valley filling technique* and develop its accelerated version called the *accelerated valley filling technique*. We present an application of this technique to an optimal parallel algorithm for construction of minimax trees.

Table of Contents

List of figures

Acknowledgements

First of all, I would like to thank my supervisor, David Kirkpatrick, for his guidance, the support, and the encouragement that he has given me through my years at the University of Biritish Columbia. I had the very good fortune to have studied under his supervision.

I am also indebted to Derek Corneil for his guidance and hospitality during may half year visit to the University of Toronto.

I would like to extend my gratitude to the members of my committee Feng Gao, Maria Klawe, and Nick Pippenger for their suggestions on ways in which I could improve my presentation.

I would like also to thank Józef, my husband and coauthor of several papers, for his love, encouragement, and introducing me to knot theory, topology, and hyperbolic geometry.

Finally I would like to thank all UBC graduate students, faculty, and staff who made my graduate study a rewarding experience. Special thanks to Nou Dadoun and Lisa Higham for many stimulating discussions and reading of various drafts of papers (and translating several of them into English); to Yueli, Jadranka, and Hilde for putting up with me as an office mate.

This research has been founded in part by the University of British Columbia Graduate Fellowship, the I.W. Killam Predoctoral Fellowship and the B.C. Advanced Systems Institute Graduate Student Scholarship.

I would like to dedicate this thesis to my children Tomek and Pawełek who were born during my graduate studies.

CHAPTER 1: INTRODUCTION

The concept of a tree has been used in various areas of mathematics for over one century. Among the first people known to use this notion were Kirchhoff [Ki1847] and Cayley ([Cal1857],[Cal1859]). Since then trees have appeared over and over again in different branches of mathematics, physics, and computer science. In computer science, especially, trees appear to be one of the most fundamental notions. It is hard to enumerate all aspects in which trees are used in computer science. Let us mention here their application as data structures (in particular hierarchical data structures), as a systematic way of exploring graphs, and as a representation of other objects (for example expressions, certain families of graphs). Parallel algorithms have brought a new application of trees, namely as a structure supporting scheduling techniques. For this reason parallel algorithms use trees even more critically than corresponding sequential algorithms.

Sequential algorithms for trees have been generally well studied. Unfortunately many of these sequential algorithms use methods which seem to be inherently sequential. One of the contributions of this thesis is the introduction and investigation of several parallel techniques for the construction of various types of trees and the presentation of new parallel tree construction algorithms using these methods. Along with parallel tree construction techniques we develop techniques which have broader applications.

The remainder of this chapter is devoted to a brief introduction to parallel computation related to trees. In Section 1.1, we introduce our parallel computation model. In Section 1.2, we briefly discuss parallel techniques for trees and relate our work to other work in the

field. The section also provides an overview of Chapters 3-5. In Section 1.3, we introduce the notion of an algorithmic schema, which provides a tool for a uniform presentation of computational techniques. In Section 1.4, we introduce basic definitions and notation concerning trees. In Chapter 2, we present general parallel techniques which support our tree construction techniques. We then describe in more detail our contribution to tree contraction (Section 2.5) and the cascading sampling technique (Section 2.6). In Section 2.7, we briefly discuss the parallel divide-and-conquer technique and its application to an algorithm for the All Dominating Neighbors problem which is used in Chapter 5.

1.1.The Computation Model

We use the Parallel Random Access Machine (PRAM) as our model of computation ([ForWyl87],[Gol76]). This is a synchronous parallel model of computation in which all processors have access to a common memory in addition to their local memory. We assume that a processor can access any cell of the shared memory in unit time. The processors are indexed by natural numbers and each processor knows its index. We consider the following PRAM models classified according to the manner of resolving read/write conflicts :

- Exclusive Read Exclusive Write (EREW) PRAM - no concurrent reads or writes are allowed,
- Concurrent Read Exclusive Write (CREW) PRAM- concurrent reads are allowed but no concurrent writes,
- Concurrent Read Concurrent Write (CRCW) PRAM- concurrent reads and writes are allowed.

According to the method of resolving write conflicts we consider several further submodels of the CRCW PRAM model. In the COMMON CRCW PRAM model,

processors are allowed to write simultaneously under the condition that they write exactly the same value. In the ARBITRARY CRCW PRAM model, among processors which try to write simultaneously, an arbitrary processor will succeed, and in the PRIORITY CRCW PRAM model, the processor with the highest index will succeed.

The above models are not equivalent. The most powerful among them is the PRIORITY CRCW PRAM and the weakest is the EREW PRAM. However any algorithm which runs on the PRIORITY CRCW PRAM can be implemented on the EREW PRAM using the same number of processors with the time complexity multiplied by a logarithmic factor (see for example [Vis83]). A discussion of relations between PRAM models can be found, for example, in the survey of Karp and Ramachandran [KarRam86] and [FicRagWid88], [Bop89].

The PRAM model of computation is an idealized model of computation. Existing parallel machines usually consist of a number of small processors connected in a fixed network. However the methods used with a network model are usually closely related to the topology of the network. The PRAM model neglects the hardware constraints existing in real parallel machines. Thus it allows us to concentrate on the logical structure of the problem which makes it very convenient for the theoretical study of parallel computation. On the other hand, the PRAM model can be efficiently simulated by more realistic models of computation ([UpfWig87], [AltHagMehPre87], [HerBil88], [HoPre89], [KarUpf88], [Ran87], [Her89]).

Let n be the size of a problem. We evaluate the performance of a parallel algorithm which solves the problem using two parameters: $t(n)$ and $p(n)$, where $t(n)$ (resp., $p(n)$) is equal to the maximum, over all instances of size n, of the number of time units (resp., the number of processors) used by the algorithm. Since we can always save a constant factor in the number of processors at the cost of the same constant factor in the running time we

omit the "big oh" when specifying the number of processors. The *total work* of a parallel algorithm is defined as the product $p(n)t(n)$. We are especially interested in designing algorithms whose running time is polylogarithmic (i.e. is $O(\log^k n)$ for some constant k) using a polynomial number of processors. The class of problems which can be solved by such algorithms is denoted by *NC*. This class was first defined by Pippenger [Pip79] and is broadly accepted as the class of problems which can be solved "fast" using a "reasonable" number of processors. An important property of the class *NC* is that it is invariant under the choice of model of computation over a broad range of models including the PRAM models described above, uniform circuit models [Bor77],[Ruz81], Alternating Turing Machines [ChaKozSto81][Ruz80], and Vector Machines [PraSto79] (see also the survey of Karp and Ramachandran [KarRam86]).

While designing efficient parallel algorithms we are searching for algorithms which minimize total work. Our secondary goal is to minimize the time factor. We usually compare the performance of a parallel algorithm with the time complexity of the best known sequential algorithm to solve the given problem. A parallel algorithm is said to achieve *optimal speedup* over a sequential algorithm which solves the same problem if the total work of the parallel algorithm is of the same order as the time complexity of the sequential algorithm. The notion of optimal speedup clearly depends on the model. Typically, we look for optimal speedup on the weakest possible PRAM model. For many algorithms presented in the thesis the use of concurrent reads improves the time complexity while the use of concurrent writes is not essential. Therefore, unless otherwise specified, we use the CREW PRAM model as our default. If the use of concurrent writes is known to improve the time complexity of the algorithm we note this explicitly.

We use the adjective "optimal" in connection with some of our algorithms. Whenever we refer to an algorithm as an optimal parallel algorithm the total work performed by the algorithm is of the same order as the sequential lower bound for the problem, and the

sequential algorithm obtained by simulating the parallel algorithm by one processor satisfies the constraints for which the lower bound was established.

1.2. Parallel Construction of Trees

While many sequential tree construction algorithms are well studied (we have optimal sequential algorithms for a large number of tree construction problems) designing parallel tree construction algorithms is not an easy task. Many sequential tree construction algorithms seem to be inherently sequential. For example, currently no deterministic efficient parallel algorithm to construct the Depth First Search (DFS) tree for an arbitrary graph is known. In fact it has been shown ([Rei85]) that given a graph with fixed adjacency lists the problem of computing of the lexicographic DFS order is P-complete (i.e. given any polynomial-time sequential algorithm A with input of length n we can construct, in O(log n) space, an instance of the lexicographic DFS problem which has a positive solution exactly when A accepts its input). Also the best algorithm known for constructing the Breadth First Search (BFS) tree for a directed graph takes $O(\log^2 n)$ time using $M(n)$ processors, where $M(n)$ denotes the number of processors needed to multiply two $n \times n$ matrices in O(log n) time. The best upper bound on $M(n)$ is currently $M(n)=O(n^{2.376})$ [CopWin87]. Thus this algorithm does not achieve optimal speedup.

We consider two basic methods of constructing trees, namely *tree expansion* and *tree synthesis*. In the *tree expansion method* we start with a single vertex and construct the tree by adding nodes of degree one and/or by subdividing edges. Typical examples of sequential algorithms constructing trees using this approach are algorithms for finding the BFS and the DFS spanning trees of a graph, Prim's algorithm for finding the Minimum Spanning Tree (MST) [Jar30], [Pri55], [Dij59], and most recursive and top-down tree construction algorithms. In a parallel setting this technique consists of introducing an

independent set of nodes forming new leaves or subdividing edges at each parallel step. Thus it is closely related to the tree contraction method ([MilRei85], [Rei85], [GibRyt87], [ColVis86c], [AbrDadKirPrz87], [GazMilTen87], [AbrDadKirPrz88], [GibRyt89]) discussed in Section 2.5. We introduce the parallel tree expansion method in Chapter 3. We use this technique to construct the tree representation for graphs in the family of graphs known as cographs. In the same chapter we outline a parallel construction of a tree representation for graphs in the family of graphs known as parity graphs. Some of the results of Chapter 3 are reported in [KirPrz87] and [PrzCor89]. Cographs, like various other families of graphs have a recursive description which allows us to represent them with the help of a tree. Among other families of graphs which permit a tree representation are parity graphs [BurUhr84], outerplanar graphs [Gol80], Halin graphs [Sys84], chordal graphs [Gol80], partial k-trees [Gol80] (in particular, series-parallel graphs), and graphs generated by context free grammars [Sli82]. Because of the number of parallel techniques related to trees, families of graphs which have tree representations often admit efficient parallel algorithms to solve problems which are difficult for general graphs (see for example [BerLawWon85], [He86b], [Ede87], [DahKar87], [NaoNaoSc87], [AbrDadKirPrz89], [RytSzy89]).

In the *tree synthesis method* we start with a forest of single node subtrees and construct the tree by adding edges or (for rooted trees) by creating parent nodes for some roots of the trees in the forest. Kruskal's algorithm for constructing a minimum cost spanning tree and algorithms to construct other families of weighted trees which minimize certain cost functions serve as examples of the tree synthesis method. A number of such families of trees (for example Huffman trees, Binary Search Trees) can be constructed using the parallel dynamic programming technique of Miller, Ramachandran, and Kaltofen ([MilRamKal86]). A major limitation of this technique is its high processor cost. In certain circumstances one can significantly reduce the number of processors used by the parallel

dynamic programming algorithm [AtaKosLarMilTen89]. It also appears that a slight relaxation of the optimality criterion can lead to more efficient parallel algorithms. We define the error of a tree to be the difference between the cost of the tree and the cost of an optimal tree for the given input. In the relaxed version of a problem, instead of looking for an optimal solution we are searching for an almost optimal tree, i.e. for a tree whose error is small. Following this idea, we present in Chapter 4 a family of parallel and sequential algorithms to construct various approximations to the Huffman tree. In particular,we present an algorithm which achieves almost optimal speedup ($O(\log n \log^* n)$ [1]time using n processors) over the Huffman algorithm and produces a tree with a very small error. This result has been achieved by applying a *cascading sampling technique* - a new technique related to that used in Cole's merging sort algorithm [Col86]. The best currently known parallel algorithm for (exact) construction of the Huffman tree is due to Atallah et. al. [AtaKosLarMilTen89]. It runs in $O(\log^2 n)$ using $n^2/\log n$ CREW processors. Some of the results presented in this chapter were reported in [KirkPrz90]. All algorithms presented in Chapter 4 apply the tree synthesis method by constructing a tree in a level-by-level fashion. At each parallel step we construct parents for elements which are (up to some approximation) on the same level of the constructed tree. The relaxation of the problem allows us to concentrate on trees of logarithmic height and consequently leads to a fast algorithm.

One might suspect that the parallel tree synthesis method can be applied only to trees of polylogarithmic height, but this is not the case. In Chapter 5 we present a technique which we call the *valley filling technique*. Our technique can be considered as an accelerated version of the technique proposed independently by Atallah et. al. [AtaKosLarMilTen89]. We present an application of this technique to an optimal parallel

[1] $\log^* n = \{\min i: \log^{(i)} n \leq 1\}$ where $\log^{(i)} n$ donotes i times composotion of the log function

algorithm for construction of minimax trees. This, as a special case, leads also to an optimal parallel algorithm for construction of trees from a given sequence of leaves' heights, improving one of the results of Atallah et. al. [AtaKosLarMilTen89].

1.3. A systematic approach to algorithmic problems

In this section we present a formal framework for the presentation of computation techniques and families of algorithms. Sections 2.2, 2.3, 2.5 of Chapter 2 provide examples (of increasing difficulty) of the application of this method. The reader may choose to read this section together with one of those sections.

While describing an algorithmic technique we try to abstract a common method used in a family of algorithms. In our approach we would like to see the description of the method and an algorithm which solves a problem using the given method to be in a similar relation as an abstract algebra and its model. To achieve this goal we describe computational methods with the help of *algorithmic schemes*. An algorithmic scheme is a partially uninterpreted algorithm. This notion was probably first introduced in 1960 by McCarthy ([McC60]) and subsequently developed and popularized by Paterson and Hewitt ([PatHew70]). We define an algorithmic scheme as a sequence $\mathcal{A} = \langle \mathcal{D}, p_1,...,p_n, Ax, \mathcal{P} \rangle$ where $p_1,...p_n$ are names of procedures or functions, Ax is a set of axioms defining properties of $p_1,...,p_n$, $\mathcal{P}$ is a program which in addition to the terms of a programming language contains names $p_1,...,p_n$, and $\mathcal{D}$ is the domain of the input. One can think of $\mathcal{P}$ as an algorithm whose meaning depends on the definitions of $p_1,...,p_n$. A sequence $p_1,...,p_n$ is called an interpretation of the algorithmic scheme $\mathcal{A}$ if $p_1,...,p_n$ is a sequence of procedures and functions satisfying axioms Ax. An interpretation of an algorithmic scheme $\mathcal{P}$ uniquely determines an algorithm, namely the algorithm obtained from $\mathcal{P}$ by substituting names $p_1,...,p_n$ for procedures $p_1,...,p_n$. Thus we usually identify an

interpretation with the algorithm it defines. For a simple application of this description method see, for example, Section 2.2.

We can also view an algorithmic scheme as universal description of a family of algorithms. We take a great advantage of this approach in Chapter 4 where the use of an algorithmic scheme not only simplifies the description but also allows us to state and prove properties common to all algorithms in the family.

We use algorithmic schemes in descriptions of many but not all techniques depending on how much can be gained by using this sort of formalism. We often replace an axiomatic presentation of properties of procedures and functions by a less formal but more intuitive description.

1.4. Basic definitions and notation

Our graph theoretical terminology is standard (*cf.* [AhoHopUll74], [Tar83], [Knu68]). A *tree* is a connected graph without cycles. A tree together with a distinguished node, called *the root*, is called a *rooted tree*. This thesis mainly concerns rooted trees so we omit the adjective "rooted" whenever no confusion arises. Let U be the set of nodes of a rooted tree T. Nodes of degree one (for a rooted tree nodes of degree one different than the root) are called *leaves*. A node which is not a leaf is called an *internal node*. For a node $u \in U$ we use $l_T(u)$ to denote the length of the unique path from the root to u. $l_T(u)$ is refered to as the *depth* of node u in tree T. The value $\max_{u \in U} l_T(u)$ is called the *height* of T. The i^{th} *level* of a tree is the set of all nodes of depth i. By convention we think of a tree as having the root at the top and the leaves at the bottom.

For any two nodes $u,v \in U$ if u belongs to the path from the root to node v then u is an *ancestor* of v and v is a *descendant* of u. If in addition u and v are connected by an

edge then u is the *parent* of v (we denote it by *parent*(v)) and v is a *child* of u. Nodes with a common parent node are called *siblings*.

A rooted tree is called a *t-ary tree* if every internal node has at most t children. A rooted tree whose every internal node has exactly t children is called a *full t-ary tree*. A full binary tree all of whose leaves belong to at most two adjacent levels is called an *almost balanced binary tree*.

An *ordered tree* is a tree together with an ordering of each internal node's children. We think of this ordering as the left-to-right ordering. In particular, for an internal node of an ordered binary tree we distinguish its *left child* and *right child*. If v is an internal node of an ordered binary tree then *right*(v) denotes its right child and *left*(v) denotes its left child. An ordering is often implicit in a representation of a tree. Typically our tree construction algorithms construct ordered trees.

A *subtree* rooted at an internal node u of a tree T is the tree induced on the set of nodes which are descendants of u (including u).

We also consider trees whose leaves are labelled with elements from some set. A family of ordered trees such that the right to left order of the leaf labels is fixed is called a family of *alphabetic trees* . If the leaves of a tree are labelled by elements from R^+ then we are dealing with *(leaf) weighted trees*. Let V be a set of leaves. A (leaf) weighted tree is a tree together with a function *w*: V—> R^+ (where V is the set of leaves of T) called the *weight function*. The value *w*(*v*) is called the *weight of the element v*. Frequently the weight function can be extended to all nodes by defining the weight of an internal node to be some function of the weights of its children.

Two weighted trees, T and T', are isomorphic if there exists a weight preserving isomoprhism from T to T'. Two weighted alphabetic trees, T and T', are isomorphic if there exists a weight and order preserving isomorphism from T to T'.

Let $\mathfrak{T}$ be a finite family of weighted (possibly alphabetic) trees. A function c : $\mathfrak{T} \longrightarrow R^+$ such that if T, T'$\in \mathfrak{T}$ are isomorphic then $c(T) = c(T')$ and if T'' is a subtree of T then $c(T'') \leq c(T)$ is called a *cost function.* A tree $T^* \in \mathfrak{T}$ such that $c(T^*) = \min_{T \in \mathfrak{T}} c(T)$ is called an *optimal tree with respect to the cost function c.*

CHAPTER 2: BASIC TECHNIQUES FOR DESIGN OF PARALLEL ALGORITHMS

In this chapter we present general parallel techniques which support our tree construction techniques, and we describe in more detail our contribution to the tree contraction technique (Section 2.5) and the cascading sampling technique (Section 2.6). In Section 2.7, we discuss the parallel divide-and-conquer technique and present a divide-and-conquer algorithm for the All Dominating Neighbor problem which will be used in Chapter 5.

2.1. Brent's scheduling principle

Brent's scheduling principle [Bre74] is a method which is often used to reduce the number of processors used by a parallel algorithm. Consider an algorithm which performs $t(n)$ parallel steps $s_1,\ldots,s_{t(n)}$. Assume that in step s_i the algorithm performs x_i parallel operations. The number of processors used by the algorithm is $m = \max x_i$. Let $x=\sum_{i=1}^{t(n)} x_i$. We can simulate step s_i of the algorithm with $p<m$ processors in at most $\lceil \frac{x_i}{p} \rceil$ time. Thus the total time needed to simulate the entire algorithm using p processors is bounded by $\frac{x}{p} + t(n)$. This technique, known as *Brent's scheduling principle*, allows us to reduce the number of processors if a large number of processors are idle during a substantial part of the computation. An example of such a computation is the prefix sum algorithm presented in the next section. It should be noted that to apply Brent's scheduling principle one must

know how to associate each processor to its job. This problem is called the *processor allocation problem* and may be nontrivial in general.

2.2. Prefix sum computation

Associated with an array $X=[x_1,...,x_n]$ of n real numbers is an array $S=[s_1,...,s_n]$ of prefix sums, where $s_j = \sum_{i=1}^{j} x_i$ $(j=1,...,n)$. The prefix sum problem is, given an input array X compute the array S of its prefix sums. A parallel prefix sum algorithm was probably first proposed by Ofman ([Ofm63]), however more attention has been given to a later paper of Ladner and Fischer ([LadFis80]). The algorithm can be described as follows: Build an almost balanced binary tree (embedded in the plane) whose consecutive leaves correspond to the consecutive elements of the input array and such that every internal node knows the address of the preceding node from the same level (which we call its *left neighbor* if such a vertex exists). Then for every internal node v compute the sum of the elements corresponding to the leaves of the subtree rooted at v. This can be done in O(log n) time with n processors by a bottom-up computation (see Figure 2.1 a). Now process the vertices in top-down fashion. For every internal node v the value of its right child is set to be equal to the value of v and the value of its left child is increased by the value of the left neighbor of v (if it exists). At the end of this computation the i-th leaf has the value of the i-th prefix sum associated with it. An O(log n)-time n-processor EREW PRAM implementation is obvious. To reduce the number of processors to n/log n we apply Brent's principle, assigning each of n / log n processors to a block of log n successive leaves. In this way the obvious simulation of the O(n)-processor implementation incurs an additional additive overhead of only O(log n) time.

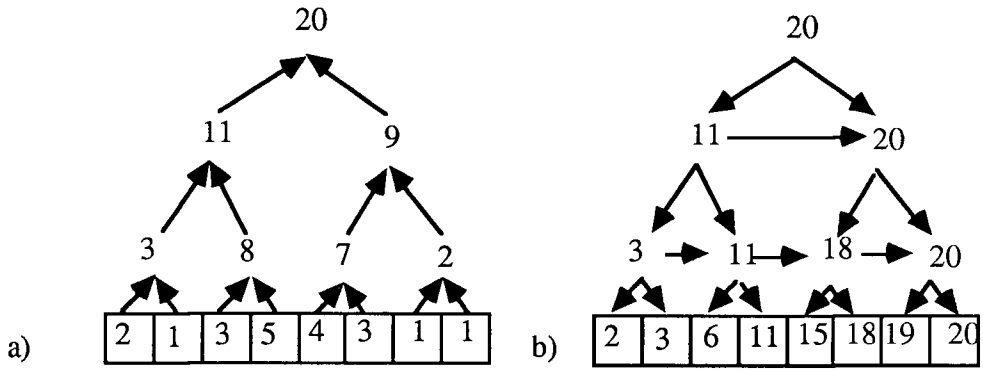


Figure 2.1. A prefix sum computation

The method used to solve the prefix sum computation problem can be described by the following algorithmic scheme (we assume for simplicity that $n = 2^m$) :

$$\mathcal{P}\mathcal{S} = <\mathcal{D},\oplus, Ax, \mathcal{P}>$$

where:

$\mathcal{D}$ is a domain,

$\oplus : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$,

$Ax : x,y,z \in \mathcal{D} \Rightarrow x\oplus(y\oplus z)= (x\oplus y)\oplus z$

$\mathcal{P}$: Input : array $X=[x_1,...,x_n]$ of elements from $\mathcal{D}$.

Output :$S= [s_1,...,s_n]$

Scheme: **For all** i **do** $s_i^0 := x_i$;

For k:=1 **to** m **do for each** i=1,...,2^{m-k} **do** $s^k_i := s^{k-1}_{2i-1} \oplus s^{k-1}_{2i}$;

For k:=m **to** 1 **do for each** i=1,...,2^{m-k} **do** $s^{k-1}_{2i} := s^k_i$;

$s^{k-1}_{2i-1} := s^k_{i-1} \oplus s^{k-1}_{2i-1}$;

For all i **do** $s_i := s_i^0$;

It is easy to see that for any interpretation of $\oplus$ which satisfies the axiom in Ax the above scheme yields an algorithm which computes $s_i = x_1 \oplus x_2 \oplus ... \oplus x_i$. Furthermore if $\oplus$ can be computed in $O(\mathbb{T})$ time with $\mathbb{P}$ processors then the corresponding interpretation of $\mathcal{P}\mathcal{S}$ can be implemented in $O(\mathbb{T} \log n)$ time with $\mathbb{P}n$ processors. To reduce the number of processors to $\mathbb{P}n/\log n$ we apply Brent's principle, assigning each of

$\mathbb{P}n / \log n$ processors to a block of $\log n$ successive leaves and simulate the $O(\mathbb{P}n)$ processor implementation with $\mathbb{P}n / \log n$ processors just as we simulated the n processor implementation of the prefix sum algorithm with $n/\log n$ processors.

Note that if X is a boolean array and $\oplus$ is interpreted as the "or" function then the above scheme leads to an algorithm which computes the "or" function of the boolean values given in the array in $O(\log n)$ time using $n/\log n$ EREW processors. Cook, Dwork, and Reischuk [CooDwoRes86] showed that the "or" function requires time $\Omega(\log n)$ on a CREW PRAM no matter how many processors or memory cells are used. Therefore the above algorithm achieves the best possible bound. Berkman, Breslauer, Galil, Schieber, and Vishkin [BeBrGalSchVis89] showed that the time factor can be reduced to $O(\log\log n)$ (preserving the total work up to a constant factor) if a CRCW model is used.

2.3. List ranking

The list ranking problem is to compute, for each element of a list of elements, its distance from the end of the list. The list ranking problem can be solved using a method closely related to the prefix sum method. The list ranking problem is equivalent to a suffix sum computation, provided that the input sequence is presented in the form of a list and all elements have an assigned value equal to one. Wyllie [Wyl81] proposed a simple list ranking algorithm which can be viewed as an interpretation of the following algorithmic scheme (we assume for simplicity that $n = 2^m$):

$$\mathfrak{IR} = <\mathfrak{D}, \oplus, Ax, \mathfrak{P}>$$

where:

$\mathfrak{D}$ is a domain;

$\oplus : \mathfrak{D} \times \mathfrak{D} \rightarrow \mathfrak{D}$,

$Ax : x,y,z \in \mathfrak{D} \Rightarrow x\oplus(y\oplus z) = (x\oplus y)\oplus z$

$\mathcal{P}$: Input: $X = [x_1,...,x_n]$; $SUCC= [succ_1,...,succ_n]$
where X is an array containing elements of the list and $succ_i$ is equal to the index in X of the element following x_i in the list ;

Output : $D=[d_1,...d_n]$ where $d_i = \bigoplus_{j=i...n} x_j$

Scheme: **For** log n iterations **repeat**
in parallel for i =1 ..n **do** X(i):=X(i) ⊕X(SUCC(i));
SUCC(i): =SUCC(SUCC(i)).
in parallel for i =1 ..n **do** D(i):=X(i);

To obtain a parallel algorithm for the list ranking problem we simply set $x_i=1$ for i=1,..n and interpret ⊕ as addition.

In general if ⊕ can be computed in $O(\mathbb{T})$ time with $\mathbb{P}$ processors then the above algorithmic scheme leads to an $O(\mathbb{T} \log n)$-time and $\mathbb{P}n$ -processor algorithm. The technique used in this scheme is also called the *pointer jumping technique*. Much effort has been made to find a list ranking algorithm which solves the problem in O(log n) time using only n/log n processors. Like the prefix sum computation this is the best one can do on a CREW PRAM model. The algorithm given by Wyllie [Wyl81] has been gradually improved through a series of deterministic and randomized results ([Vis84], [KruRudSni85], [MilRef85], [AbrDadKirPrz87b]). A deterministic algorithm achieving these complexity bounds (on a EREW PRAM) has been given by Cole and Vishkin in [ColVis86]. Because of the large constant in the time factor this algorithm is mainly of theoretical interest. Subsequently, a simpler optimal algorithm (which involves reasonable constants) has been proposed in Anderson and Miller [AndMil88]. Any of these algorithms can be abstracted to obtain an algorithmic scheme leading to an $O(\mathbb{T} \log n)$-time $\mathbb{P}n/\log n$-processor algorithm.

2.4. Euler tour technique

The Euler tour technique [TarVis84] provides a general method for reducing computation of certain functions on trees (for example preorder number, number of ancestors) to list ranking. The basic idea is to replace each tree edge by a pair of directed edges with opposite directions, and consider an Euler path which starts and ends at the root. This Euler path, together with some labeling forms the input to a list ranking problem. Assume, for simplicity, that the input tree is a binary tree. To describe the technique we present an algorithm to compute the left-to-right numbering of tree leaves:

Example 2.1. Computing the left-to-right numbering of leaves of an input tree

Input : a rooted binary tree T;

Output : left-to-right numbering of leaves

STEP 1. Build a unique list L corresponding to the input tree :

Each node v of T is split into three nodes v_T, v_L and v_R. For each of the resulting nodes we define a *next* field as follows. If v is a leaf then $v_T.next = v_L$ and $v_L.next = v_R$. If w is the right child of v then $v_L.next = w_T$ and $w_R.next = v_R$. If w is the left child of v then $v_T.next = w_T$ and $w_R.next = v_L$. What results is a list that starts at $root_T$ and ends at $root_R$ and traverses each edge of T once in each direction. This is the unique list L corresponding to the input tree.

STEP 2. Reduce the computation of the tree function to some interpretation of $\mathfrak{LR}$ which takes the list L as its input.

Label the elements of L as follows: $x^T_i = 1$, $x^L_i = x^R_i = 0$ if x_i corresponds to a leaf and $x^T_i = x^L_i = x^R_i = 0$ otherwise and interpret $\oplus$ as addition. It is easy to confirm that after running this interpretation of $\mathfrak{LR}$ on the list L, the value at each x^T_i corresponding to a leaf is equal to the left-to-right number of this leaf.

2.5. Tree Contraction

The tree contraction problem occurs most naturally in the context of dynamic expression evaluation for an expression presented in the form of a parse tree. The size of an expression is defined by the number of leaves of the parse tree. Brent [Bre74] showed that expressions of size n could be rewritten in straight-line code of depth O(log n). His approach is a top-down approach and its natural dynamic implementation seems to require $O(\log^2 n)$ time. Miller and Reif [MilRei85] describe a deterministic algorithm for dynamic expression evaluation which runs in O(log n) time with O(n) processors. A similar result has been independently reported by Rytter [Ryt85]. A single step of the algorithm of Miller and Reif converts the current binary parse tree to a simpler one by removing, in parallel, all leaves (RAKE operation) and compressing maximal chains of nodes with only one child (COMPRESS operation). During the RAKE operation we compute the values of the expressions at internal nodes all of whose children are leaves. We also modify the functions associated with internal nodes whose number of children has been reduced as the result of the RAKE operation, to obtain functions with the number of arguments equal to the number of children. During the COMPRESS operation we perform a symbolic computation consisting of the composition of the one argument functions associated with the nodes on the compressed paths. Miller and Reif show that after O(log n) such steps the given tree will be reduced to its root (and thus the expression defined by the tree is computed).

Miller and Reif apply their method to construct parallel algorithms for problems which can be reduced to computation on trees. They give a randomized algorithm for testing isomorphism of trees, a deterministic algorithm for constructing a tree of 3-connected components of a planar graph, as well as other algorithms. Among the important contributions of Miller and Reif [MilRei85] is their abstraction of the problem of tree contraction. This leads to a separation of the problem from its familiar applications

(notably, dynamic expression evaluation) and places it, along with list ranking, among the fundamental problems of parallel computation. A natural side effect has been the identification of new and unforeseen applications [DadKir87], [He86], [GibRyt86].

Another approach to dynamic expression evaluation is presented by Gibbons and Rytter [GibRyt86]. Their algorithm runs in $O(\log n)$ time using $O(n/\log n)$ processors. They assume, however, that the input (the string representing the expression to be computed) is given in an array and is hence preordered. They prove that the algorithm can be applied to compute an algebraic expression in any algebra with finite carrier.

Cole and Vishkin [ColVis86c] propose an alternative method for computation on trees. They solve the tree contraction problem by parallel reduction to the list ranking problem. In this respect their approach is similar to the approach described in this thesis. Our reduction, an abstraction of the technique used for the parallel preprocessing of region trees for fast (sequential) subdivision search [DadKir87], is simpler and more explicit than that of Cole and Vishkin; in particular, it completely avoids the centroid decomposition techniques that lie at the heart of their reduction. Similar algorithms have been proposed independently by Gibbons and Rytter [GibRyt88] and Kosaraju and Delcher [KosDel88].

All the algorithms considered above share a basic similarity: in each parallel step a parse tree is shrunk by independent vertex removal. For this reason the technique used for dynamic expression evaluation is called the *tree contraction technique*. One can see a tree contraction technique as a scheduling technique. Namely, for each processor we decide to which node it is assigned at a given time unit. The processor assigned to a tree node at a given time is available for performing a computation which consists either of computing a subexpression or performing a symbolic computation. Additionally the nodes which have been assigned a given time unit are removed from the tree at this time unit.

The tree contraction method appears to be a very powerful technique. Thus several researchers have tried to formalize a class of problems which can be solved using this technique. He [He86b] defines the binary tree algebraic computation (BTAC) problem and applies Miller and Reif's technique to obtain a parallel algorithm for this problem. Roughly speaking, the BTAC problem is to compute the value of an algebraic expression given in the form of a parse tree under the assumption that the algebra in which the computation is performed has a finite carrier. A more abstract approach has been taken by Abrahamson, Dadoun, Kirkpatrick and Przytycka [AbrDadKirPrz87] and Miller and Teng [MilTen87]. Our description again follows [AbrDadKirPrz87] but an influence of [MilTen87] should be noted.

2.5.1. A tree contraction algorithm.

We will assume that trees are presented as an (unordered) array of vertices each of which has associated with it a parent pointer and a doubly-linked list of children. (Of course, it is possible to efficiently convert to or emulate such a representation starting with other more primitive representations.) (Successive) vertices on any list of children are said to be (immediate) siblings.

Let T be any binary tree with vertex set $V(T)$. A sequence of trees $T_1, T_2, \ldots T_k$ is said to be a *tree contraction sequence* of length k for T if,

(i) $T_1 = T$;
(ii) $V(T_i) \subseteq V(T_{i-1})$;
(iii) $|V(T_k)| \leq 3$; and
(iv) if $v \in V(T_{i-1}) - V(T_i)$ then either
  (a) v is a leaf of T_{i-1}, or
  (b) v has exactly one child, x, in T_{i-1}, $x \in V(T_i)$, and the parent of v in T_{i-1} is the parent of x in T_i.

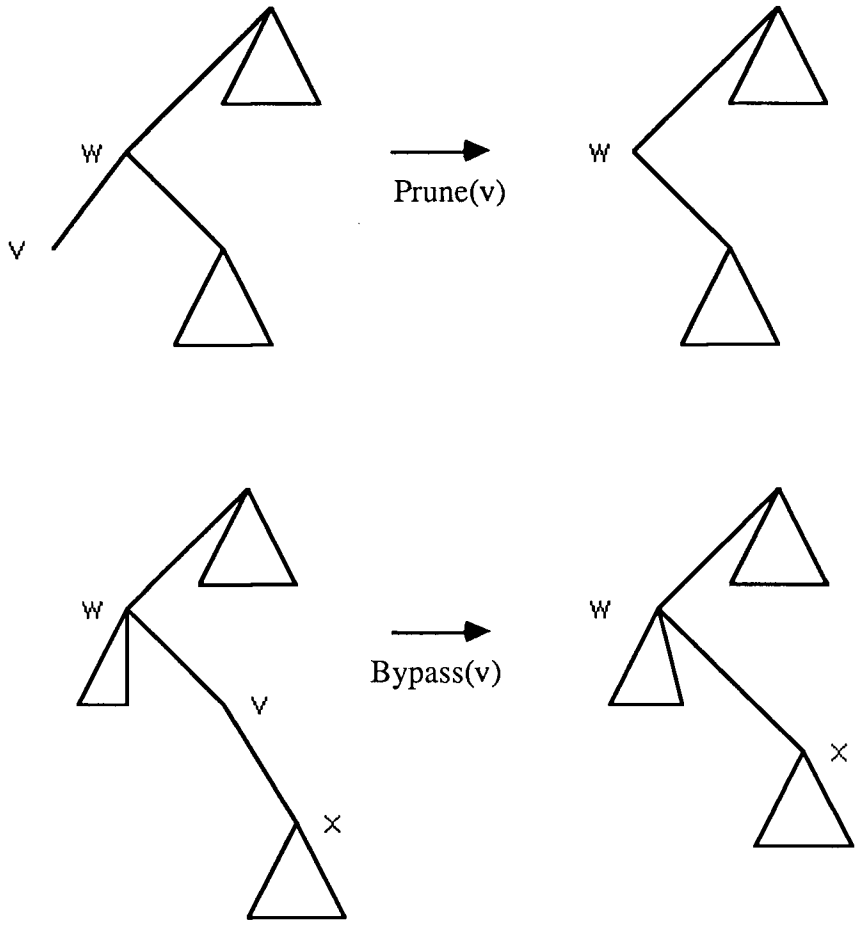


Figure 2.2. Prune and Bypass operations

It is clear from the definition that successive trees in a tree contraction sequence are formed by "independent" executions of the following two fundamental operations (see Figure 2.2): Prune(v) — leaf v is removed from the current tree; and Bypass(v) — non-root node v with exactly one child x is removed from the current tree, and the parent w of v becomes the new parent of x (with x replacing v as a child of w). By "independent" we mean that if v is pruned or bypassed then its parent is not bypassed. In this way tree modifications are ensured to be local and executable in parallel.

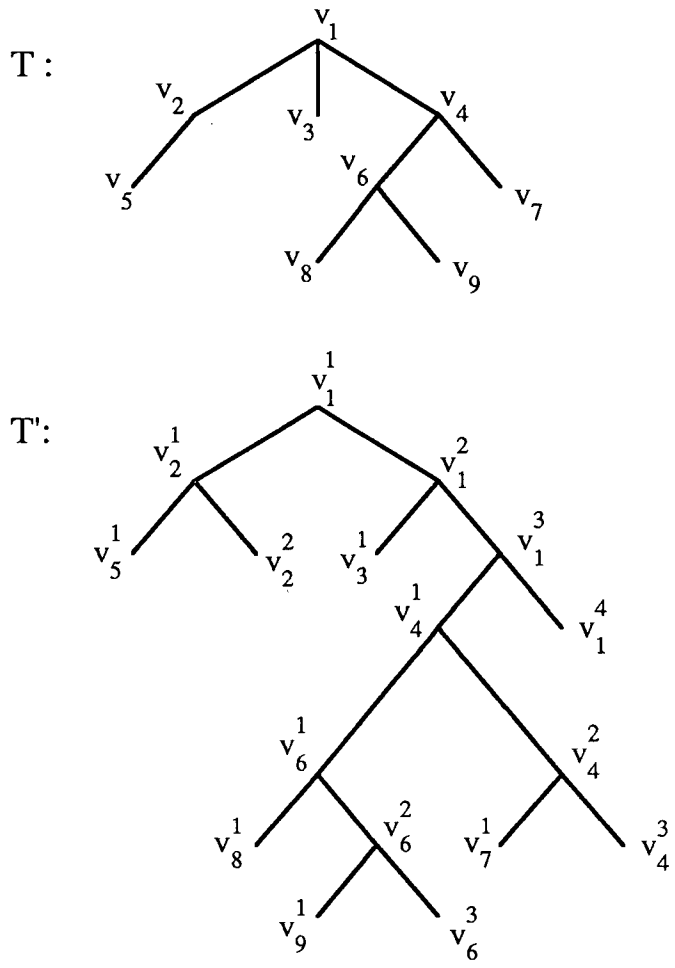


Figure 2.3. Interpretation of Trees as Binary Trees

We say that a contraction sequence is α–*compact* if it has length at most $\alpha \log n$.

The *tree contraction problem* is to construct, for an arbitrary tree T, an α-compact tree contraction sequence for T where α is some universal constant.

For our applications it is not necessary to construct the sequence explicitly; it suffices to associate with each node v the index i of the highest indexed tree containing v in the sequence, together with pointers to the parent and child (if any) of v in T_i. Since the tree contraction problem for a nonbinary tree T can be naturally reduced to a tree contraction problem for a binary tree T' obtained from T by a binarization step (see Figure 2.3) we

concentrate on binary trees only. We should also note that the operations RAKE and COMPRESS used in the tree contraction algorithm of Miller and Reif [MilRei85] can be implemented with the help of the Prune and Bypass operations.

The pair of operations Prune(v) followed by Bypass(parent(v)) (where v is any leaf) form a conceptual unit in our algorithm. The algorithm proceeds in phases, each of which consists of a batch of these basic contractions performed in parallel. The independence of the underlying operations is guaranteed by a simple global schedule for leaf removal. Let the leaves be numbered in left to right order. A leaf is removed in phase t if the rightmost 1 in its leaf index is in position t.

Our binary tree contraction algorithm has the following simple description:

```
procedure contract (T)
(* Assign leaf indices from 0 to n - 1 *)
 for each leaf v in parallel
      index(v) ← left_to_right leaf index of v
(* Contraction iterations. *)
 repeat ⌈log n⌉ times
      for each leaf v in parallel

           w ← parent (v)

           if index(v) is odd and w ≠ root
                then  if v is a left child
                           then  prune (v)
                                 bypass (w)

                      if v is a right child
                           then prune (v)
                                bypass (w)

           else index(v) ← index (v) /2
```

Note that the innermost **if** statements, though they have opposite conditions, are intended to be executed in sequence, with appropriate synchronization in between. Thus,

each iteration of the **repeat** loop has four slots in which prune or bypass operations may be executed. Accordingly, we associate four elements of the tree contraction sequence with each iteration of the **repeat** loop, describing the tree after each of the four slots. It is also helpful to view the behavior of the algorithm at two other levels. It is immediate from the description that each prune operation is immediately followed by a bypass. Hence in each successive pair of slots a number of composite prune-bypass operations are executed in parallel. Each pair of these composite slots (making up an entire iteration of the **repeat** loop) serves to eliminate all of the leaves with odd parity in the current tree together with their parents. An example of a tree contraction sequence produced by procedure **contract** is given on Figure 2.4.

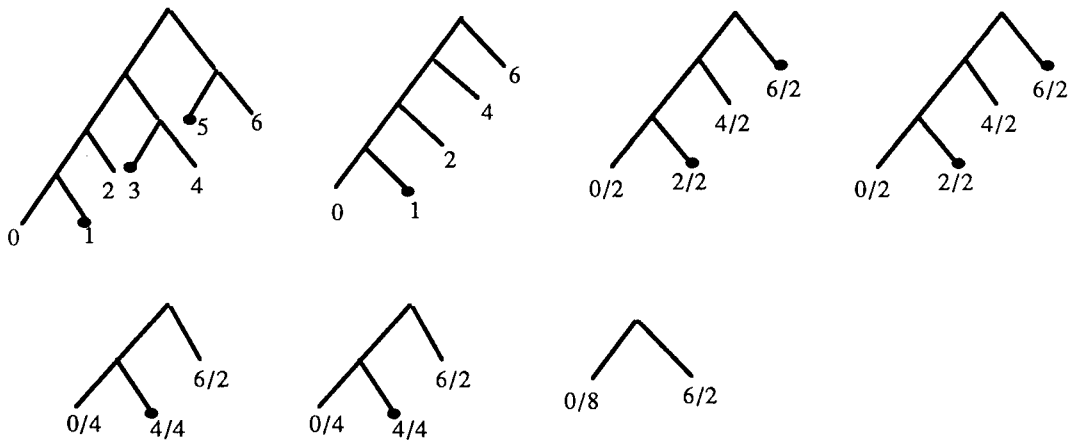


Figure 2.4. A tree contraction sequence

Lemma 2.2. Procedure **contract** constructs a 2-compact tree contraction sequence.

Proof: It suffices to demonstrate that the prunes and bypasses are performed independently. Since prunes and bypasses are never executed simultaneously, it need only be demonstrated that no vertex v and its parent w are ever bypassed simultaneously. Suppose this is not the case. Without loss of generality, v is the right child of w. Since they are bypassed simultaneously, they must both have leaves as left children. But since

these leaves are adjacent in the left to right order and since the index array maintains each leaf's left-to-right rank in the current contraction iteration (except, possibly, for the left child of the root), v and w must have indices of opposite parity, a contradiction. ■

Theorem 2.3. Procedure **contract** provides an $O(\log n)$-time and $O(n / \log n)$-processor EREW PRAM deterministic reduction of tree contraction to list ranking.

Proof: We can compute the left-to-right numbering of the leaves of the tree in $O(\log n)$ time with $n/\log n$ processors using the Euler tour technique (Example 2.1). Thus it suffices to prove that the iterated contraction step of procedure **contract** can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors. An $O(\log n)$ time, $O(n)$ processor implementation is immediate; if one processor is devoted to each leaf then each phase can be carried out in $O(1)$ time. To reduce the number of processors to $n/\log n$ we apply Brent's principle as in the prefix sum computation scheme (Section 2.2) ■

It follows from Theorem 2.3 that the results for list ranking carry over directly to tree contraction. We summarize the most important consequence in the following:

Corollary 2.4. The tree contraction problem can be solved deterministically in $O(\log n)$ time using $O(n / \log n)$ processors on a EREW PRAM.

2.5.2. Bottom-up Algebraic Tree Computations

A tree contraction algorithm gives a method for solving a large class of parallel tree computation problems. This class includes, for example, dynamic expression evaluation ([MilRei85], [GibRyt86]). He [He86a] and Gibbons and Rytter [GibRyt86] noted that any algebraic expression with operands from an algebra with carrier of fixed finite size can be computed in the cost of tree contraction. This result provides efficient parallel algorithms for several optimization problems, for example minimum covering set, maximum

independent set and maximum matching, when the underlying graph is a tree [He86] or has a tree representation [AbrDadKirPrz88], [RytSzy89].

In fact, we can relax the assumption that the carrier of the algebra is of fixed finite size and put some restrictions on the operations only. We can generalize He's binary tree algebraic computation problem in the following way: Let $\mathfrak{D}$ be a set and $\mathfrak{F} \in \{f \mid f: \mathfrak{D} \times \mathfrak{D} \rightarrow \mathfrak{D}\}$ a set of two-variable functions over $\mathfrak{D}$. The objective of bottom-up algebraic tree computations is to take any ordered regular binary tree T whose leaves are labelled by elements of $\mathfrak{D}$ and whose internal nodes are labelled by elements of $\mathfrak{F}$ and to evaluate the algebraic expression associated with T (where functions at internal nodes denote operators and elements labelling leaves are operands).

It is natural (and helpful) to generalize the above notion to include a set of functions $\mathfrak{G} \in \{g \mid g: \mathfrak{D} \rightarrow \mathfrak{D}\}$, including the identity function, which serve as edge labels and influence the computation in the obvious way. The triple $(\mathfrak{D}, \mathfrak{G}, \mathfrak{F})$ defines a *bottom-up algebraic tree computation* (B-ATC) *problem.*

An indexed set $\mathfrak{F}$ of functions is called *$(\mathbb{T}, \mathbb{P})$ - universal* if there exists a universal algorithm to compute the value of any function from $\mathfrak{F}$ in any point of its domain in time $\mathbb{T}$ using $\mathbb{P}$ processors.

We can formalize a class of problems which can be solved using the tree contraction method the help of the following algorithmic scheme :

$$\mathfrak{B}\_\mathfrak{A}\mathfrak{T}\mathfrak{C} = <\mathfrak{D}, \mathfrak{G}, \mathfrak{F}>$$

where:

$\mathfrak{D}$ - a domain

$\mathfrak{G} \subseteq \{g \mid g: \mathfrak{D} \rightarrow \mathfrak{D}\}$;

$\mathfrak{F} \subseteq \{f \mid f: \mathfrak{D} \times \mathfrak{D} \rightarrow \mathfrak{D}\}$;

Ax: (decomposability axioms)

(i) $\mathfrak{G}$ and $\mathfrak{F}$ are ($\mathbb{T}$, $\mathbb{P}$) - universal for some $\mathbb{T}$ and $\mathbb{P}$;

(ii) for all $g_i, g_j, \in \mathfrak{G}, f_m \in \mathfrak{F}$ and $a \in \mathfrak{D}$ the functions g_s and g_t given by

$$g_s(x) = g_i(f_m(\, g_j(x), a\,)\,) \text{ and } g_t(x) = g_i(f_m(\, a, g_j(x)\,)\,)$$

both belong to $\mathfrak{G}$ and their indices s and t can be computed from i, j, m and a by an algorithm which runs in $\mathbb{T}$ time using $\mathbb{P}$ processors

$\mathfrak{P}$: Input: A binary tree T with internal nodes labelled with functions $\mathfrak{F}$ from and edges labelled with functions from $\mathfrak{G}$ and leaves labelled with elements from $\mathfrak{D}$.

Output: The value of the expression defined by the parse tree T.

Scheme: Run the procedure **contract**(T) labelling the new tree edges as follows: Let e be an edge whose one endpoint is a leaf labelled a and the other endpoint is internal node v. Let g_j be the label of the edge leading from v to its nonleaf son and g_i be the label of the edge leading from v to its parent. The edge introduced by an application of Prune to the a leaf labelled a and Bypass to its parent v is given the label $g_s(x) = g_i(f_m(\, g_j(x),a\,))$ or $g_t(x) = g_i(f_m(\, a, g_j(x)\,))$ depending on whether a is the left or the right child of v.

The idea of the algorithm is perhaps most easily understood by referring to the transformations of Figure 2.5.

A B-ATC problem which satisfies the decomposability axioms (i) and (ii) defined above is called *decomposable.*

Theorem 2.5. For any interpretation of the $\mathfrak{B}\_\mathfrak{A}\mathfrak{T}\mathfrak{C}$ scheme and for any input tree the associated algebraic expression can be evaluated in time $O(\mathbb{T} \log n)$ using $\mathbb{P}\, n/\log n$ processors.

Proof: This follows immediately from the tree contraction algorithm and the decomposability axioms.

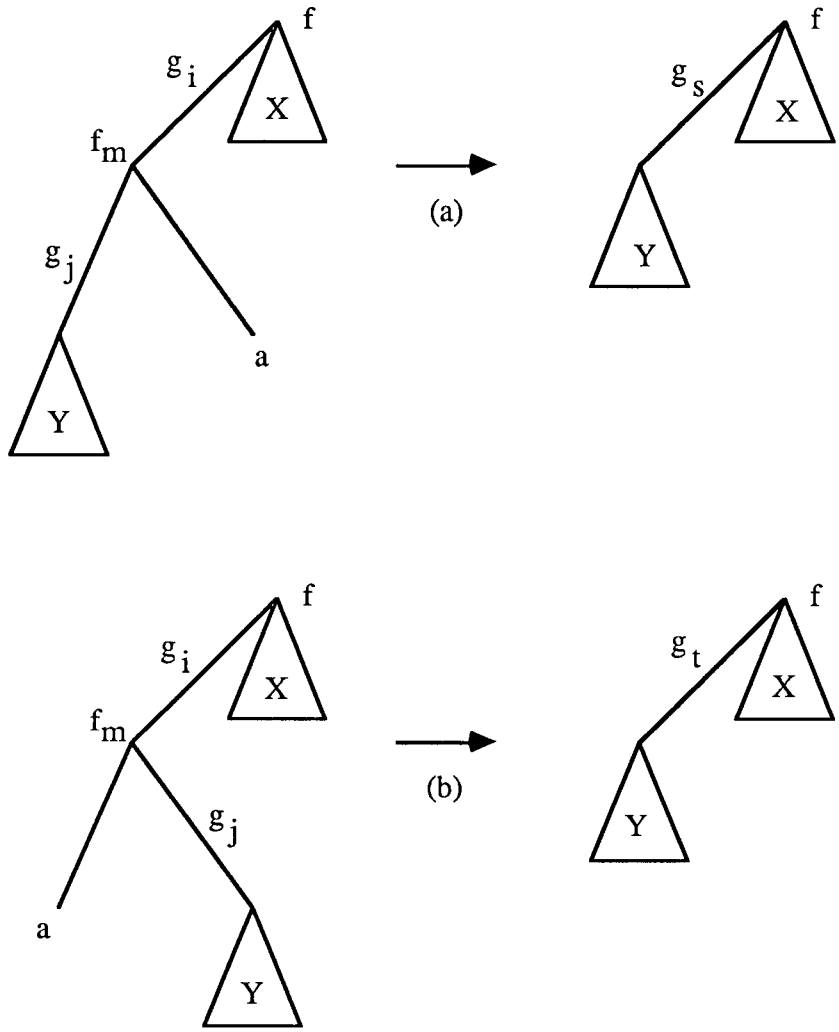


Figure 2.5. New labelling of edges after Prune and Bypass operations

Remark 2.6: We should note that we can easily modify the computation scheme in such a way that we will also compute all the functions in internal nodes. In order to do so each bypassed node maintains a pointer to the lower level endpoint of the bypassing edge (i.e. the node whose computation has to be finished in order to finish the computation of the given node). For example, in Figure 2.5 the bypassed node labelled f_m retains a pointer to the leaf labelled a and the root of the subtree Y. After finishing the computation of the function in the root we add a new phase to the algorithm. In this phase we allow all the internal nodes to finish computing the values of the function assigned to these nodes. Note

that for every internal node v removed at some iteration of the procedure **contract** one of its children remains in the tree. Furthermore to finish the computation at v it suffices to know the value of this child. In the last tree of the tree contraction sequence the values at all nodes are computed. Thus we can complete the computation at the remaining vertices by processing them in the reverse of their order of elimination in the contraction sequence.

We refer to the $\mathfrak{B}\_\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ computation scheme modified as in Remark 2.6 as the *full* $\mathfrak{B}\_\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ computation scheme.

Note that any algebra with a finite carrier has an associated instance of $\mathfrak{B}\_\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ (with $\mathbb{T}$ and $\mathbb{P}$ being constants). It is also easy to see that general {+, -, *, /} arithmetic computations can be described as an instance of the $\mathfrak{B}\_\mathfrak{Q}\mathfrak{T}\mathfrak{C}$. In the example given below we present an interpretation of the $\mathfrak{B}\_\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ which we use in our optimal algorithm to construct minimax trees (Chapter 5).

Example 2.7: Let T be a tree of n leaves such that every internal node u is labelled with an integer number λ_u. Consider the following bottom-up assignment of values to the tree nodes:

$$\text{value}(u) = \begin{cases} 1 \text{ if u is a leaf} \\ \left\lceil \dfrac{\text{value}(u_1) + \text{value}(u_2)}{\lambda_u} \right\rceil \text{if u has children } u_1 \text{ and } u_2. \end{cases}$$

To compute this labelling one can use the following interpretation of the $\mathfrak{B}\_\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ scheme:

Define the following indexed families of functions ($\mathbb{N}$ denotes the set of natural numbers):

$$\mathfrak{G} = \{ g_{ij} : \mathbb{N} \to \mathbb{N} \mid j>0;\ g_{ij}(x) = \left\lceil \frac{x+i}{j} \right\rceil \};$$

$$\mathfrak{F}=\{ f_m:\mathbb{N} \times \mathbb{N}\to\mathbb{N} \mid m>0;\ f_m(x,y)=\left\lceil \frac{x+y}{m} \right\rceil \}.$$

Associate with each edge of T the identity function $g_{0,1}$ and with each internal node u the function f_{λ_u}. We show that $<\mathbb{N},\mathfrak{G},\mathfrak{F}>$ is an interpretation of the $\mathfrak{B}\_\mathfrak{QTC}$.

It is obvious that the value of any function from the family $\mathfrak{G}$ or $\mathfrak{F}$ can be computed in a constant time from its argument(s). Thus to prove that $<\mathbb{N},\mathfrak{G},\mathfrak{F}>$ defines an instance of the $\mathfrak{B}\_\mathfrak{QTC}$ it suffices to prove that for any i,j,k,l,m the following holds:

(i) $g_{i,j}(f_m(g_{k,l}(x),a)) = g_{q,p}(x)$ for some q and p and q,p can be computed in O(1) time from i,j,k,l,m; and

(ii) $g_{i,j}(f_m(a,g_{k,l}(x))) = g_{q,p}(x)$ for some q and p and q,p can be computed in O(1) time from i,j,k,l,m;

We need the following fact:

Fact: For any real number z and integer n (n>0) $\left\lceil \frac{\lceil z \rceil}{n} \right\rceil = \left\lceil \frac{z}{n} \right\rceil$.

Proof: If $\lfloor z \rfloor = z$ then the equality is obvious. So assume $\lfloor z \rfloor < z$. Note that $\lfloor z \rfloor = nk+m$ for some $k,m \geq 0$ and $m<n$. So $z = \lfloor z \rfloor + \varepsilon = nk+m+\varepsilon$ for some $0 < \varepsilon < 1$ and $\lceil z \rceil = nk+m+1$.

Thus $\left\lceil \frac{\lceil z \rceil}{n} \right\rceil = \left\lceil \frac{nk+m+1}{n} \right\rceil = \left\lceil k+\frac{m+1}{n} \right\rceil = k+1.$

On the other hand $\left\lceil \frac{z}{n} \right\rceil = \left\lceil \frac{nk+m+\varepsilon}{n} \right\rceil = \left\lceil k+\frac{m+\varepsilon}{n} \right\rceil = k+1.$ ■

Since f_m is symmetric it suffices to prove only (i). But by the fact above we have:

$$g_{i,j}(f_m(g_{k,l}(x),a)) = \left\lceil \frac{\left\lceil \frac{\left\lceil \frac{x+k}{l} \right\rceil + a}{m} \right\rceil + i}{j} \right\rceil = \left\lceil \frac{\left\lceil \frac{x+(k+la)}{lm} \right\rceil + i}{j} \right\rceil = \left\lceil \frac{x+(k+la+mli)}{mlj} \right\rceil =$$

$g_{k+la+mla,lmj}(x)$.

Thus to compute value(u) for every node u we assign the element 1 to each leaf and apply the above interpretation of the full $\mathfrak{B}\_\mathfrak{A}\mathfrak{T}\mathfrak{C}$ scheme. As a consequence the problem of Example 2.7 can be evaluated in time O(log n) using n/log n processors.

2.5.2. Top-down Algebraic Tree Computations

The $\mathfrak{B}\_\mathfrak{A}\mathfrak{T}\mathfrak{C}$ provides a useful abstraction of many bottom-up tree-based computations. In a number of applications it is necessary to consider top-down (or perhaps a combination of bottom-up and top-down) computations based on trees. For this purpose we introduce $\mathfrak{T}\_\mathfrak{A}\mathfrak{T}\mathfrak{C}$ - the Top-Down Algebraic Tree Computation scheme. Formally:

$$\mathfrak{T}\_\mathfrak{A}\mathfrak{T}\mathfrak{C} = <\mathfrak{D}, \mathfrak{F}>$$

where:

$\mathfrak{D}$ - a domain

$\mathfrak{F} \subseteq \{f \mid f: \mathfrak{D} \rightarrow \mathfrak{D}\}$;

Ax: (i) $\mathfrak{F}$ is $(\mathbb{T},\mathbb{P})$ - universal for some $\mathbb{P}$ and $\mathbb{T}$;

(ii) $\mathfrak{F}$ is closed under composition and for each h_i, $h_j \in \mathfrak{F}$ the index of $h_i \circ h_j$ can be computed from i and j by an $O(\mathbb{T})$-time $\mathbb{P}$-processor universal algorithm.

$\mathfrak{P}$: Input: A binary tree T with edges labelled with functions from $\mathfrak{F}$ and the root labelled with an element, a, from $\mathfrak{D}$.

Output: For each internal node or leaf,v, evaluate the function $f_v(a)$ defined as follows: $f_v(a)=g(f_{parent(v)}(a))$ where g is the function associated with the edge (v,*parent*(v)).

Scheme: Run the tree contraction algorithm. Each edge e introduced by a bypass operation can, as part of the tree contraction process, be labelled by the composition of the functions associated with the nodes and edges on the path (in T) joining the endpoints of e. The vertices of T are considered in the reverse

order of their elimination in the tree contraction sequence. It is straightforward to confirm that the values of vertices in tree T_i can be computed in time T knowing the values of vertices in tree T_{i-1}.

Theorem 2.8: For any interpretation of $\mathfrak{T}\_\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ and for any input tree the value associated with each vertex can be computed in $\mathbb{T}\log(n)$ time with $\mathbb{P}n/\log n$ processors.

Proof: This follows immediately from the tree contraction algorithm and axioms (i)-(ii).

2.6. Cascading sampling

This section is devoted to a technique which was developed to support parallel algorithms based on a sequence of merging steps.

The idea of using a "sampling" technique for a problem related to merging was first explored by Cole [Col86] in connection with his optimal parallel merge sort algorithm. Cole's idea has been generalized further by Atallah, Cole, and Goodrich [AtaColGoo87] to a method called the *cascading divide-and-conquer technique*. Our technique differs from Cole's technique but shares certain important similarities. Both techniques allow a speedup of a computation which is naturally divided into log n levels. (For example, in a divide-and-conquer algorithm the levels corresponds to the depths of recursive calls). Usually one applies a sampling technique to allow pipelining ([Col86], [Kos89]). In our approach, the sampling elements are computed during a preprocessing step. They do not allow starting a computation on a higher level prior to finishing the computation on lower levels, but they make it possible to apply a more efficient algorithm on each of the levels.

Let f be a function which given a sorted sequence produces a sorted sequence of no longer length and let # denote the merge of two sorted sequences. Consider the following problem: given m sorted sequences $d_1,...,d_m$ ($m \leq \log n$) of total length at most n

compute the following sequences $f(d_m)$, $f(d_{m-1}\#f(d_m))$, $f(d_{m-2}\#f(d_{m-1}\#f(d_m)))$, ... The method of computing such a sequence of sequences would depend on the properties of the function f. Our technique can be helpful when the nature of the function f is such that sequences $f(d_m)$, $f(d_{m-1}\#f(d_m))$, $f(d_{m-2}\#\ f(d_{m-1}\#f(d_m)))$, ... are most naturally computed one after another using m consecutive steps. Several examples of such functions are given below.

Example 2.9 :

a) $$f(x_1,....,x_r) = \begin{cases} x_1+x_2,\ x_3+x_4,....,x_{r-1}+x_r \text{ if r is even} \\ x_1+x_2,\ x_3+x_4,....,x_{r-2}+x_{r-1}\ ,\ 2x_r \text{ if r is odd;} \end{cases}$$

b) $f(x_1,....,x_r) = x_1+0,\ x_2+x_1,....,x_r+x_{r-1}$;

c) Let g: $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow$ {true, false}. Define f as follows:

$f(x_1,....,x_r) = x_{i_1},....,x_{i_s}$ where $I=i_1,....,i_s$ is a subsequnce of the sequence 1,...,r such that j belongs to I if and only if g(j-1,j,j+1) is true.

To illustrate the cascading sampling technique in a possibly simple way we ignore function f and show a method of computing sequences d_m, $d_{m-1}\#d_m$, $d_{m-2}\#d_{m-1}\#d_m$, ... one after another in m consecutive steps. One can solve the above problem by applying Valiant's merging algorithm ([Val75], [BorHop85]) O(log *n*) times interleaved with computation of the values of the corresponding functions. This leads to an O(log *n* loglog *n*)-time *n*-processor solution to the problem.

In our approach, to speedup the algorithm, we precede the sequence of mergings by a preprocessing step. We say that sequence d_i belongs to *level* i. The goal of the preprocessing is to divide the merging problems into subproblems which can be solved independently such that at least one of the sequences occurring in a subproblem is "short".

Since the time used by Valiant's algorithm is O(loglog r), where r is the length of the shorter list, this may speedup the computation significantly.

The preprocessing step consists of log*n *sampling steps*. Informally, in the s^{th} sampling step, for every level $i < n$, every $(2^s)^{th}$ element from this level is merged into the level i+1. Then, among the elements which arrive at level i+1 every second element is merged into level i+2 and so on until level $i + \lceil \log^{(s)} n \rceil$ is reached[1] or no elements are left. Formally, let V^s_i be the sorted sequence of elements on level i after the s^{th} sampling step. Initially $V^s_i = V_i^0 = V_i$. Let *dist*(V,y) denote the distance of the element y from the beginning of the sequence V. For every sequence V^{k-1}_i define a family of sorted sequences $S^s_{i,1}, S^s_{i,2}, \ldots$ in the following way:

$$S^s_{i,t} = \{x \mid \exists y \in V^{s-1}_i \text{ s.t. } x=y \text{ and } dist(V^{s-1}_i, y) = 2^{s+t-1}m \quad \text{for some integer } m\};$$

Informally, $S^s_{i,t}$ is the sequence of elements which are originated at level i in the sampling step s and which arrive on level i+t.

Then V^s_j is defined as $V^{s-1}_j \# \left(\#S^s_{i,t} \right)$, where the count ranges over $t \le \lceil \log^{(s)} n \rceil$ and $i+t=j$.

Elements produced in the sampling process are called *sampling elements*. All other elements are called *real*. To simplify the description we assume that we add sampling elements at the beginning and at the end of every list. For each sampling element x in level i there is a unique element y in level i-1 such that x = y (we say that x = *sample*(y)) . The element y is called the *source element* for element x. A source element may also be a sampling element. For any sampling element, x, define *source*(x) to be equal to the source element of x. Sampling elements which have been generated in the same sampling step form sequences using the pointer *source*. The element (real or generated in an earlier

[1] $\log^{(i)} n$ denotes i times composition of the log function

sampling step) which is pointed at by the pointer *source* of the last element of such a sequence is said to be the *origin of the given sequence*. Let u_1, u_2, be a pair of sampling elements from the same level i such that there are no other sampling elements between them. The subsequence of elements which lies between u_1 and u_2 is called a *basic sequence*. If u_1 and u_2 bound a basic sequence then the sequence of elements from the level i-1 which lies between the source of u_1 and the source of u_2 is called the *gap* associated with this basic sequence. The number of elements in a gap is called the *size of the gap*.

Remark 2.10: To merge two sequences on successive levels i-1 and i it suffices to merge each basic sequence from level i with its corresponding gap on level i-1 (after application of the function f_{i-1} to all its elements). Thus if after the preprocessing step the size of a gap is bounded by $g(n)$ then the whole algorithm can be implemented in loglog $g(n)$ time with n processors.

Two important properties of the sampling process are given by the following lemmas.

Lemma 2.11. After $\log^* n$ sampling steps, each gap size is no more than $2^{\log^* n+2}$.

Proof: In the proof of the lemma we use the following simple facts:
Fact 1: a) $\lfloor x \rfloor \le \lfloor \frac{x}{m} \rfloor m+m$; b) $\lceil x \rceil \le \lfloor \frac{x}{m} \rfloor m+m+1$.

Consider an interval [a,b] on level j. Let us restrict our attention only to those elements whose samples may belong to this interval. Thus we consider only elements v from levels i=1,2,...,j-1 such that if $v \in V_{j-t}$ then $v \in [a,b]$. We call these elements *interval elements*. Let n^s_i be the number of interval elements on level i after s sampling steps (for simplicity we assume the convention that for i less than 1, n^s_i is equal to zero). The proof of the lemma is based on the following fact:
Fact 2: $n^s_i \le 2^s n^s_{i+1} + 3\lceil \log^{(s)} n \rceil + 2^{s+1}$.
Proof of Fact 2: We prove Fact 2 by induction on s. For s=1 we have:

$n^1_i \le n^0_i + \lceil \frac{1}{2} n^0_{i-1} \rceil + \ldots + \lceil \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil} \rceil$ and

$n^1_{i+1} \ge n^0_{i+1} + \lfloor \frac{1}{2} n^0_i \rfloor + \ldots + \lfloor \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil+1} \rfloor$

so $n^1_i - 2^1 n_{i+1} \le (n^0_i - 2\lceil \frac{1}{2} n^0_i \rceil) + (\lceil \frac{1}{2} n^0_{i-1} \rceil - 2\lfloor \frac{1}{4} n^0_{i-1} \rfloor) + \ldots + \lceil \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil} \rceil \le$

$\le (2+1)\lceil \log n \rceil + \lceil \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil} \rceil$ (by Fact 1b)

$\le 3\lceil \log n \rceil + 1.$

Assume now that $n^{s-1}_i \le 2^{s-1} n^{s-1}_{i+1} + 3\lceil \log^{(s-1)} n \rceil + 2^s$. But

$n^s_i \le n^{s-1}_i + \lceil \frac{1}{2^s} n^{s-1}_{i-1} \rceil + \ldots + \lceil \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil} \rceil$ and

$n^s_{i+1} \ge n^{s-1}_{i+1} + \lfloor \frac{1}{2^s} n^{s-1}_i \rfloor + \ldots + \lfloor \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil+1} \rfloor.$

Therefore

$n^s_i - 2^s n^s_{i+1} \le (n^{s-1}_i - 2^s \lfloor \frac{1}{2^s} n^{s-1}_i \rfloor) + (\lceil \frac{1}{2^s} n^{s-1}_{i-1} \rceil - 2^s \lfloor \frac{1}{2^{s+1}} n^{s-1}_{i-1} \rfloor) + \ldots$

$+ \lceil \frac{1}{2^{s+\lceil \log^s n \rceil - 1}} n^{s-1}_{i-\lceil \log^s n \rceil} \rceil$

$\le 2^s + 1 + (\lceil \frac{1}{2^s} n^{s-1}_{i-1} \rceil - 2\lfloor \frac{1}{2^{s+1}} n^{s-1}_{i-1} \rfloor + 2\lfloor \frac{1}{2^{s+1}} n^{s-1}_{i-1} \rfloor - 2^s \lfloor \frac{1}{2^{s+1}} n^{s-1}_{i-1} \rfloor) +$

$\ldots + \lceil \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil} \rceil$

$\le 2^s + 1 +$

$2 + 1 - (2^s - 2) \lfloor \frac{1}{2^{s+1}} n^{s-1}_{i-1} \rfloor + \ldots +$

$2 + 1 - (2^s - 2) \lfloor \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil+1} \rfloor +$

$\lceil \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil} \rceil$

$\le 2^s + 3\lceil \log^{(s)} n \rceil - 2 + \lceil \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil} \rceil -$

$2^{s-1} \lfloor \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil+1} \rfloor.$

But by the inductive hypothesis we have:

$$n^{s-1}_{i-\lceil \log^s n \rceil+1} \ge \frac{n^{s-1}_{i-\lceil \log^{(s)} n \rceil} - 3\lceil \log^{(s-1)} n \rceil - 2^s}{2^{s-1}}.$$

So $2^{s-1} \lfloor \frac{1}{2^{s+\lceil \log^{(s)} n \rceil - 1}} n^{s-1}_{i-\lceil \log^{(s)} n \rceil+1} \rfloor$

$$\geq 2^{s-1}\left\lfloor \frac{1}{2^{s+\lceil \log(s) n\rceil -1}} \frac{n^{s-1}{}_{i-\lceil \log(s) n\rceil} - 3\lceil \log^{(s-1)} n\rceil - 2^s}{2^{s-1}} \right\rfloor$$

$$\geq \left\lfloor \frac{n^{s-1}{}_{i-\lceil \log(s) n\rceil} - 3\lceil \log^{(s-1)} n\rceil - 2^s}{2^{s+\lceil \log(s) n\rceil -1}} \right\rfloor - 2^{s-1} \qquad \text{(by Fact 1(a))}$$

$$\geq \left\lceil \frac{1}{2^{s+\lceil \log(s) n\rceil -1}} n^{s-1}{}_{i-\lceil \log(s) n\rceil} \right\rceil - \left\lceil \frac{3\lceil \log^{(s-1)} n\rceil + 2^s}{2^{s+\lceil \log(s) n\rceil -1}} \right\rceil - 2^{s-1} - 2.$$

Therefore

$$n_i^s - 2^s n_{i+1}^s \leq 2^s + 3\lceil \log(s) n\rceil - 2 + \left\lceil \frac{3\lceil \log^{(s-1)} n\rceil + 2^s}{2^{s+\lceil \log(s) n\rceil -1}} \right\rceil + 2^{s-1} + 2 \leq 2^s + 3\lceil \log(s) n\rceil + 1 + 2^{s-1} \leq$$

$3\lceil \log(s) n\rceil + 2^{s+1}$. ■

Now we can continue the proof of the Lemma 2.11. Let a,b be the real numbers which bound weights of elements in a gap, say on level j. Consider the last sampling step. In this step level j receives only elements from level j-1. After the last sampling step in any gap on level j there are at most $2^{\log^* n}$ elements from level j. By Fact 2, level j-1 has at most $2^{2\log^* n} + 3\lceil \log(\log^* n) n\rceil + 2^{\log^* n + 1} < 2^{2\log^* n + 1}$ interval elements. Every $2^{\log^* n}$ of them are sent to level j. Thus in a gap on level j there are at most $2^{\log^* n + 1}$ elements originated at level j-1 and $2^{\log^* n}$ elements from level j. This proves that the number of elements in the gap is bounded by $2^{\log^* n + 2}$. ■

Lemma 2.12: After the sampling process the total number of elements is bounded by $3n$.

Proof: Let a_s be the number of elements after s^{th} sampling step. Obviously:

$$a_0 = n \quad \text{and}$$

$$a_s \leq a_{s-1} + \frac{a_{s-1}}{2^s} = a_{s-1}\left(1 + \frac{1}{2^s}\right)$$

But it is easy to check (by induction) that $a_s \leq 3n\left(1 - \frac{1}{2^s}\right)$. ■

Now we show how to implement the sampling process in $O(\log^* n \log n)$ time with n processors.

IMPLEMENTATION OF THE SAMPLING PROCESS: Since, by Lemma 2.12, in any sampling step we have at most $3n$ elements we have one processor per constant number of elements. However, we must perform some computation which assigns elements to processors. Initially we have one processor per real element. Inductively assume that processors 1,...,j have been assigned r elements and processors j+1,...,n have been assigned r-1 elements. In the s^{th} sampling step the processor associated with a given element checks whether this element is the origin of a sequence of sampling elements and if so, how long this sequence is. (Note that in the s^{th} sampling step the l^{th} element in a sequence is the origin of a chain of length $\max\{i \mid l \equiv 0 \bmod 2^{s+i-1}\}$). Since the length of a sampling sequence is bounded by log n the generation of such a sequence can be done in O(log n) time using one processor per sequence.

To assign processors to new sampling elements we number these elements in such a way that if v and u are two sampling elements and the origin of v is at a level higher than the origin of u, or origins of v and u are at the same level but the origin of v precedes the origin of u, then v receives a smaller number than u. Note that every "old" element (i.e. an element which is not introduced as a sampling element in the given sampling step) knows the number of new sampling elements it has originated. So using the prefix sum computation we can compute for every "old" element the number of new sampling elements preceding it. Since, for every new sampling element y, its position in the sequence of new sampling elements originated by the same element, say x, is known in order to obtain the number of y, it suffices to add this position number to the number of new sampling elements preceding x. After numbering all new sampling elements the element numbered m is assigned to the processor numbered (j+m)mod n. This solves the problem of processor allocation.

It remains to insert new sampling elements into their proper positions in the sequences produced in the previous sampling step. This can be done by a global sorting

algorithm. Note that it is important that the sorting procedure which we are using is stable (i.e. it preserves the order of equal elements). If it is not, we can number all elements (in a way similar to the way we numbered all new sampling elements) and sort pairs (number of the element, its value) lexicographically. Since the number of sampling steps is O(log*n), the whole sampling process can be implemented in O(log*n log n) time with n processors.

In the description of the cascading sampling method we assumed that f is the identity function. Now we can discuss properties of f which are needed to apply the technique. The basic property which we need is, what we call, the *distribution by sampling*, and can be described as follows (compare Figure 2.6) .

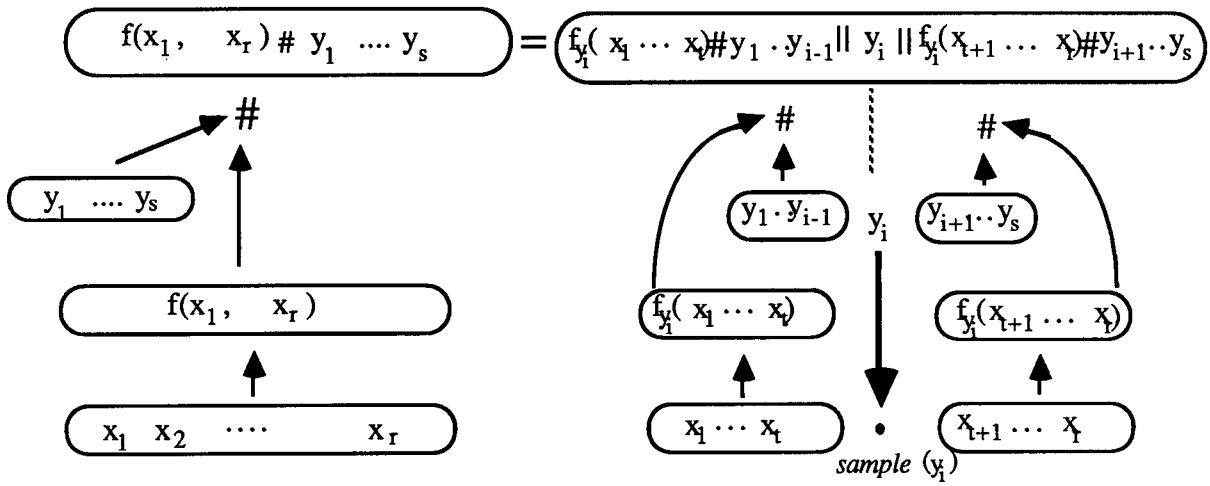


Figure 2.6. Distribution by sampling

Let $x_1,....,x_r$ and $y_1,....,y_s$ be two sorted sequences and let || denotes the concatenation of sequences. Function f converting a sorted sequence into a sorted sequence is said to be *distributive by sampling* if for any y_i we can compute value *sample*(y_i) and function f_{y_i} whose complexities are bounded by the complexity of f and such that for any y_i if $x_1,....,x_t \le$ *sample*(y_i) $\le x_{t+1},....,x_r$ then

$f_{y_i}(x_1,....,x_t)\#y_1,....,y_{i-1} \parallel y_i \parallel f_{y_i}(x_{t+1},....,x_r)\#y_{i+1},...,y_s = f(x_1,....,x_r)\ \# y_1,....,y_s$.

For the function given in Example 2.9a *sample*(y) is simply y/2. Function f_{y_i} is a modification of the function f and insures a special care of the boundary elements x_t and x_{t+1} so that if t is odd then the resulting sequence $f_{y_i}(x_1,....,x_t)\#y_1,....,y_{i-1} \parallel y_i \parallel f_{y_i}(x_{t+1},....,x_r)\#y_{i+1},...,y_s$ contains elements x_t+x_{t+1}, $x_{t+2}+x_{t+3}$, ... rather than $2x_r$, $x_{r+1}+x_{r+2}$, $x_{r+3}+x_{r+4}$,... (which would occur when we simply apply f to subsequnces $x_1,....,x_t$ and $x_{t+1},....,x_r$). A more detailed description of the application of the cascading sampling technique to a problem very similar to the one presented in Example 2.9 a is given in Chapter 4 (section 4.4.2).

2.7. Parallel divide and conquer and the All Dominating Neighbors problem

In the divide-and-conquer technique a problem is divided into independent subproblems which are solved recursively. The solution to the problem is obtained from the solutions to its subproblems. In parallel implementations of this technique, we solve the subproblems in parallel. There are two aspects in which this technique is related to tree constructions. First, it can be used for constructing certain types of trees (for example the visibility tree discussed later in this chapter). Second, this technique can be naturally visualized with the help of a tree. Namely, we can represent the recursive partition of the problem with the help of a tree T such that every internal node of T corresponds to one subproblem, and its children correspond to the partition of the subproblem associated with the given node into further subproblems. To solve a problem one has to compute solutions to all subproblems in a bottom up fashion. This computation can be implemented by performing h iterations (where h is the height of the tree) such that at the i^{th} iteration we

compute the solutions to the problems defined by the internal nodes which are on level h-i+1 of the tree T. In particular, if T has height $\lceil \log n \rceil$ then such an implementation requires $\Omega(\log n)$ time and $\Omega(n)$ processors (and therefore $\Omega(n \log n)$ total work), even if the problem admits a linear time sequential algorithm. The usual way of dealing with this problem is to reduce the number of processors using Brent's scheduling principle, or to apply a sequential algorithm for the subproblems of logarithmic size (i.e. subproblems associated with internal nodes from level $\lceil \log\log n \rceil$) and a parallel algorithm for all other subproblems. A natural generalization of the latter approach is to divide the computation into a finite number of phases such that at each phase we compute, using different algorithms, solutions to the problems associated with internal nodes from a certain range of levels. In this chapter we present such a 3-phase algorithm which leads to an optimal solution for the following problem: given a sequence of real numbers find, for every element x in the sequence, the dominating neighbors of x, that is, the closest predecessor and successor of x which are at least as large as x. We refer to this problem as to the *All Dominating Neighbors* (ADN) problem. This problem is a natural generalization of list merging (simply concatenate one list in descending order and the other in ascending order and solve the problem for the merged list), and appears to have a number of interesting applications. The ADN problem has a natural geometric interpretation which we describe in the next section. An optimal algorithm for the ADN problem leads to simple optimal algorithms for the following problems: triangulation of monotone polygons [BrSchVis88], all closest neighbors problem for a convex polygon [SchVis88], triangulation of a sorted point set, and computing horizontal neighbors for vertices of a (horizontally) monotone polygon. The solution to the ADN problem is also used as an important building block of our optimal algorithm to construct minimax trees which we present in Chapter 5.

Merks [Mer86] considered a problem which can be easily translated to the ADN problem as a part of his algorithm for point set triangulation. Merks' algorithm runs in

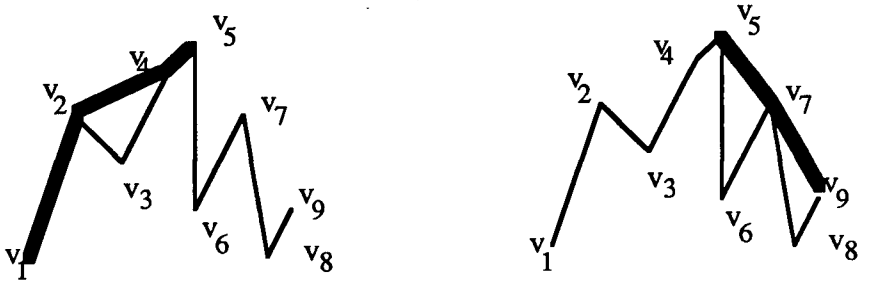
O(log n) time using n processors on a CREW PRAM. Subsequently, an optimal O(log n)-time n/log n-processor CREW PRAM algorithm (as well as an O(loglog n) time n/loglog n-processors CRCW PRAM algorithm) for the ADN problem was proposed by Berkman, Breslauer, Galil, Schieber, and Vishkin [BeBrGalSchVis89]. We present here a different CREW PRAM algorithm which achieves the same complexity bounds. This algorithm illustrates the 3-phase divide-and-conquer strategy and is needed for completeness of the results in Chapter 5. Both the algorithm of Berkman et. al. and our algorithm reduce the problem to merging certain pairs of sorted sequences. We introduce a data structure called the *visibility tree*, which is exploited by our algorithm. A slight modification of our algorithm (including a slight modification of the visibility tree) leads to a simple optimal algorithm for computing horizontal visibility in a horizontally monotone polygon.

2.7.1. The ADN problem and the visibility tree

Assume that we are given a set of points S in the plane such that no two points have the same x-coordinate. For a point $(x,y) \in S$ we define its *left (resp., right) closest greater element* as the element $(x',y') \in S$ such that $y' \geq y$, $x' < x$ (resp., $x' > x$) and the distance $|x-x'|$ is minimized (if such an element exist). Since the problem is at least as difficult as sorting the point set on their x-coordinates, we assume that elements of V are sorted by x-coordinate. Only the order of the elements is of importance (not the exact values of their x-coordinates) and therefore sorting reduces our initial problem to the following: Given a sequence $y_1,...,y_n$ of reals, find for every element y_i in the sequence, the *dominating neighbors*, that is, the closest predecessor and successor of y_i greater than or equal to y_i. We refer to this problem as to the *All Dominating Neighbors (ADN)* problem.

Consider a graph theoretical *chain* (i.e. a connected graph all but two of whose vertices have degree two and the two remaining vertices have degree one). A planar embedding of such a chain is called a *polygonal line*. A polygonal line, C, is said to be *monotone* with respect to a straight line l if every line orthogonal to l intersects C in at most one point. If l is a horizontal line then C is said to be *horizontally monotone*. Assume that C is horizontally monotone (in particular, no two vertices have the same x-coordinate). There is a natural ordering of vertices of C, namely the ordering corresponding to the increasing order of x-coordinate. For any vertex v of a polygonal line C its *horizontal neighbors* are the points on the intersection of C with the horizontal line through v which are closest to v on both sides. It is easy to see that if C is a horizontally monotone polygonal line then the problem of finding horizontal neighbors can be reduced to two instances of the ADN problem.

We now introduce basic notions whose properties are used by our parallel algorithm for the ADN problem.



a) a right visibility sequence (bold) b) a left visibility sequence (bold)

Figure 2.7. Visibility sequences

Let $V=v_0,v_1,...,v_n,v_{n+1}$ where $v_1,...,v_n$ is a sequence of reals and $v_0=v_{n+1}=\infty$. For each i, $1\le i\le n$ define $dp(i) = \max\{j<i \mid v_j \ge v_i\}$(dominating predecessor) and $ds(i)=\min\{j>i \mid v_j \ge v_i\}$ (dominating successor). A maximal subsequence $v_{i_1},...,v_{i_r}$ of a sequence $v_i,...,v_{i+k}$ such that $i_1=i$ and $i_{j+1} = ds(i_j)$ (resp., $i_{j-1} = dp(i_j)$) is called the *right* (resp., the *left) visibility sequence* for the sequence $v_i,...,v_{i+k}$ (see Figure 2.7). Note that

a right visibility sequence is a nondecreasing sequence and a left visibility sequence is a nonincreasing sequence.

As an immediate consequence of the above definition we obtain:

Lemma 2.13. Let $s_1=v_i,...,v_{i+k}$ and $s_2 = v_{i+k+1},...,v_{i+k+l}$. Let $U_1=u_1^1,...,u_r^1$ be the right (resp., the left) visibility sequence for the sequence s_1, and $U_2=u_1^2,...,u_t^2$ be the right (resp. the left) visibility sequence for the sequence s_2. Then the sequence $u_1^1,...,u_r^1,u_j^2,...u_t^2$ (resp., $u_1^1,...,u_j^1,u_1^2,...u_t^2$), where u_j^2 (resp., u_j^1) is the first element of U_2 (resp., the last element of U_1) which is greater than or equal to u_r^1 (resp., u_1^2) is the right (resp., the left) visibility sequence for the sequence $v_i,...v_{i+k},v_{i+k+1},...v_{i+k+l}$.

As we have mentioned before, the ADN problem is a natural generalization of list merging. An optimal log *n*-time *n*/log *n* processor parallel algorithm for list merging was presented by Shiloach and Vishkin [ShiVis81]. Like list merging, the ADN problem has a simple linear sequential solution. Consider, for example, the function *ds*. We process the elements of the sequence one after another. At stage *i* of the algorithm the elements which do not yet have the value *ds* computed and whose index is less than *i* are stored on a stack. The elements stored on the stack form a decreasing sequence. Initially the stack contains element v_1. At stage *i* we process element v_i. Note that i is equal to *ds*(j) for all elements v_j on the stack which are less than or equal to v_i. We remove these elements from the stack and put v_i on the stack. Note that the elements stored on the stack at this stage of the algorithm form the left visibility sequence for the sequence $V_i=v_1,...,v_{i-1}$.

In our parallel algorithm visibility sequences are used in a different way. We divide the input sequence into two subsequences of approximately equal size, then each of them into two subsequences and so on. This recursive partition is represented by an almost balanced binary tree T such that elements of V are in the leaves of T and a node u of T is

associated with the subsequence of V spanned by the leaves of the subtree of T rooted at u. Let $s_1=v_i,...,v_{i+k}$ be the sequence spanned by the leaves of the left child of u and $s_2 = v_{i+k+1},...,v_{i+k+l}$, be the sequence spanned by the leaves of the right child of u. Let $U_1=u_1^1,...,u_r^1$ be the left visibility sequence for the sequence s_1, and $U_2=u_1^2,...,u_t^2$ be the right visibility sequence for the sequence s_2. Assume that $u_1^1 \leq u_t^2$ (the case when $u_1^1 > u_t^2$ is symmetric). Then the sequence $u_1^2,...,u_{j-1}^2$ where u_j^2 is the first element of U_2 which is greater than or equal to u_1^2 is called *the right hidden sequence associated with node u* and the sequence $u_2^1,...,u_r^1$ is called *the left hidden sequence associated with node u* (see Figure 2.8). Tree T together with the visibility sequences and hidden sequences associated with every internal node is called *the visibility tree.*

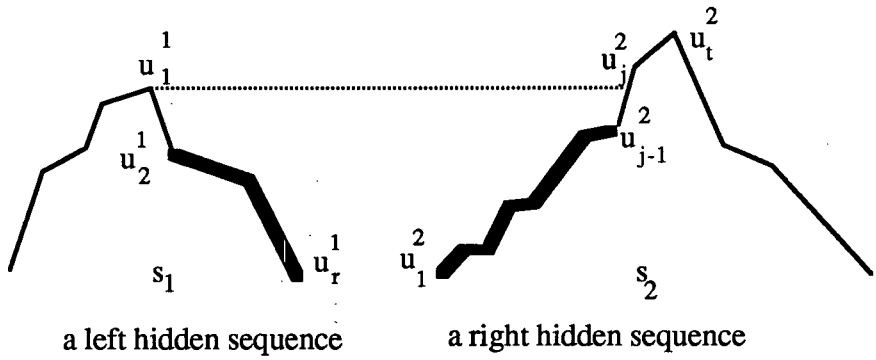


Figure 2.8. Hidden sequences (bold)

2.7.2. An optimal algorithm for the ADN problem

Assume that we are given an input sequence V and its visibility tree. (We associate a visibility/hidden sequence with a node by giving pointers to the first and the last elements of the list). Note that every input element (except for the end elements) belongs to exactly one hidden sequence. Thus it suffices to compute *dp*(x) and *ds*(x) for all elements v_x from a hidden sequence. In the following description we assume that v_x belongs to the left hidden sequence associated with an internal node u (the case when v_x belongs to the right hidden sequence is symmetric). In order to compute *dp*(x) it suffices to find the first

element in the left visibility sequence associated with the left child of u which is greater than or equal to v_x. The value *ds*(x) is equal to the index of the first element of the right visibility sequence associated with the right child of u which is greater than v_x. This element either belongs to the right hidden sequence associated with v or is equal to u_j^2 (recall Figure 2.8). It can be decided in constant time which of the two cases holds. The only case which involves more than constant time per element is finding *ds*(x) in the case when it is one of the elements of the right hidden sequence associated with u (recall our assumption that v_x belongs to a left hidden sequence). We find *ds*(x) for all elements to which this case applies with the help of a merging algorithm. We merge the right hidden sequence with the corresponding left hidden sequence in such a way that in the case of equality the elements from the right sequence follow the elements from the left sequence. Then for every element from the left sequence we search for the closest element from the right sequence following it. This reduces our problem to the problem of merging at most *n* pairs of sequences such that each pair is associated with one internal node of the visibility tree. The total number of elements in all such sequences is bounded by *n*. These merging problems can be batched into one merging problem of size *n*. To do this we number the internal nodes of the visibility tree, say in the postfix order (using for example the Euler tour technique described in section 2.4), and associate with each element in a right or left hidden sequence the sequential number equal to the number of the internal node with which the given sequence is associated. Then we concatenate left and right sequences separately in the order of the sequential numbers of their elements. This gives us two sequences each lexicographically sorted according to the triple (sequential number, value of the element, side) where side=left for elements from a left hidden sequence and side=right for the elements from a right hidden sequence. We define left<right and simply merge the sequence in O(log *n*) time using *n*/log *n* processors ([ShiVis81]). The third element of a triple ensures that in the case of equality the elements from a right sequence follow the elements from the corresponding left sequence.

It remains to show how to construct the visibility tree in O(log n) time using n/log n processors. We construct the tree in bottom up fashion using Lemma 2.13. Let u be an internal node and let U_1, U_2, and u_j^2 be defined as in the previous section (recall Figure 2.8.). Assume again that $u_1^1 \leq u_t^2$ (the case $u_1^1 \geq u_t^2$ is symmetric). Then:

- the right visibility sequence associated with u is the concatenation of the right visibility sequence of the left child of u and the sequence $u_j^2,...,u_t^2$;
- the left visibility sequence associated with u is equal to the left visibility sequence of its right child;
- the left hidden sequence associated with u is $u_2^1,...,u_r^1$;
- the right hidden sequence associated with u is $u_1^2,...,u_{j-1}^2$;

Thus the computation of the visibility and the hidden sequences for an internal node reduces to finding the vertex u_j^2. We assume in the description that $u_1^1 \geq u_t^2$; the case $u_1^1 \leq u_t^2$ is symmetric. The algorithm is divided into three phases (the levels of tree nodes are numbered in the leaves-to-root order):

PHASE 1: Computing sequences associated with internal nodes on levels 0 to $\lceil \text{loglog } n \rceil$.
Each of the n/log n processors computes the sequences associated with internal nodes of one subtree rooted at an internal node from level $\lceil \text{loglog } n \rceil$ using a sequential linear algorithm.

During the remaining phases the visibility sequences are represented as follows:

(i) Each sequence is subdivided into subsequences of size at most log n and each of them is represented by an array,

(ii) Each subsequence is assigned to one processor and each processor is assigned to a constant number of subsequences.

(iii) The processor assigned to a subsequence keeps the length of the subsequence, the address of the first element, and a pointer to the first (for the left visibility sequence) or

the last (for the right visibility sequence) subsequence of the sequence in its local memory .

This representation can be computed during the first phase of the algorithm and then preserved through the rest of the algorithm in the following way: Assume that we compute a right visibility sequence for an internal node and assume that the element u_j^2 has been computed. Let b be the subsequence containing u_j^2. Then the new sequence contains all subsequences of the right visibility sequence of the left child, the part of the subsequence b containing all elements greater than or equal to u_j^2, and all subsequences of the right visibility sequence which follow b. We assign the processor previously associated with b to the part of the sequence b which is in the new visibility sequence. The rest of the processors are associated with the same subsequences as before. The information described in point (iii) can be updated for each subsequence in constant time.

PHASE 2: Computing sequences for internal nodes on levels $\lceil \log\log n \rceil+1$ to $2\lceil \log\log n \rceil$.
We process the nodes on these levels in a bottom-up fashion. For every internal node v every processor associated with a subsequence of the right visibility sequence associated with the left child of u tests if u_1^1 lies between the first and the last element of the subsequence to which it is assigned. If so it applies a binary search algorithm to compute u_j^2. So the total amount of time spent in the second phase is $O((\log\log n)^2)$.

PHASE 3: Computing sequences internal nodes on levels $2\lceil \log\log n \rceil+1$ to $\lceil \log n \rceil$.
We assign $\lfloor \log n \rfloor$ processors to each internal node on level $2\lceil \log\log n \rceil+1$ and define these processors as *special processors associated with the given internal node*. The special processors for an internal node on a level greater than $2\lceil \log\log n \rceil+1$ are defined as the special processors of its right child. We perform $\lceil \log n \rceil-2\lceil \log\log n \rceil$ parallel steps. As in the second phase, for every internal node u from the currently processed level, every processor associated with a subsequence of the right visibility sequence associated with the

left child of v tests whether u_1^1 lies between the first and the last element of the subsequence it is assigned to. If so we assign the special processors assigned to u to the elements of this subsequence and find u_j^2 in one step taking constant time. Thus this phase of the algorithm can be implemented in O(log n) time with n/log n processors.

This gives a divide-and-conquer algorithm which provides an alternative proof of the following theorem:

Theorem 2.14. [BeBrGalSchVis89]: The ADN problem can be solved in O(log n) time using n/log n processors on a CREW PRAM.

2.7.3. Applications

As we mentioned at the beginning of this chapter, an optimal algorithm for the ADN problem has a number of interesting applications. We present here two such applications. First we present a reduction of the All Strictly Dominating Neighbors (ASDN) problem to the ADN problem. The solution to the ASDN is also used in our optimal algorithm to construct minimax trees presented in the next chapter. Then we present an optimal algorithm to compute horizontal visibility in a horizontally monotone polygon. Our algorithm uses a modified version of the visibility tree.

2.7.3.1. The All Strictly Dominating Neighbors (ASDN) problem

We present a simple optimal algorithm which computes, for every element in an input sequence, the *strictly dominating neighbors*, that is, the closest predecessor and successor which are strictly greater than the given element (if such an elements exists). We refer to this problem as the *All Strictly Dominating Neighbors (ASDN) problem.* We show a simple solution to the ASDN problem which is based on the solution of the ADN problem. Consider, for example, the problem of computing for every element v_i of the input

sequence its strictly dominating successor (*sds*(i)). Let T be the forest formed by elements of the input sequence and pointers *ds*. For all elements v_i such that $v_i<v_{ds(i)}$ we set *sds*(i)=*ds*(i) and remove from T edges (i,*ds*(i)). This splits T into a number of lists of equal elements and with equal values of *sds*. The last element of each such list has its *sds* already computed. So the values of *sds* for the remaining elements can be found by an application of a list ranking algorithm ([AndMil88],[ColVis86]).

2.7.3.2. Computing horizontal neighbors for vertices of a monotone polygon

A simple modification of the ADN algorithm presented in section 2.7.2 leads to an optimal algorithm for computing horizontal visibility (i.e. determining the horizontal neighbors of every point of a polygon) in a horizontally monotone polygon.

Let P be a monotone polygon. Assume, without loss of generality, that it is decomposed into the *upper polygonal line* and the *lower polygonal line*, i.e. two polygonal lines monotone with respect to a horizontal line. Let Y= $y_1,...,y_n$ be the sequence of y-coordinates of vertices of P listed in the order of increasing x-coordinate. (Note that given the two polygonal lines which comprise P one can compute Y in O(log *n*) time with *n*/log *n* processors by a merging algorithm).

In order to find for any vertex v its horizontal neighbors it suffices to find the edges of the polygon which contain those neighbors. Our algorithm to find horizontal neighbors is based on the same idea as the algorithm to solve the ADN problem described in section 2.7.2. We also build a visibility tree (with leaves labelled by elements of Y) and find horizontal neighbors by merging hidden sequences. However the visibility information stored in the internal nodes of the visibility tree is more complex. This only leads to difficulties of a technical nature which, as one will easily observe, can be solved by considering a number of simple cases.

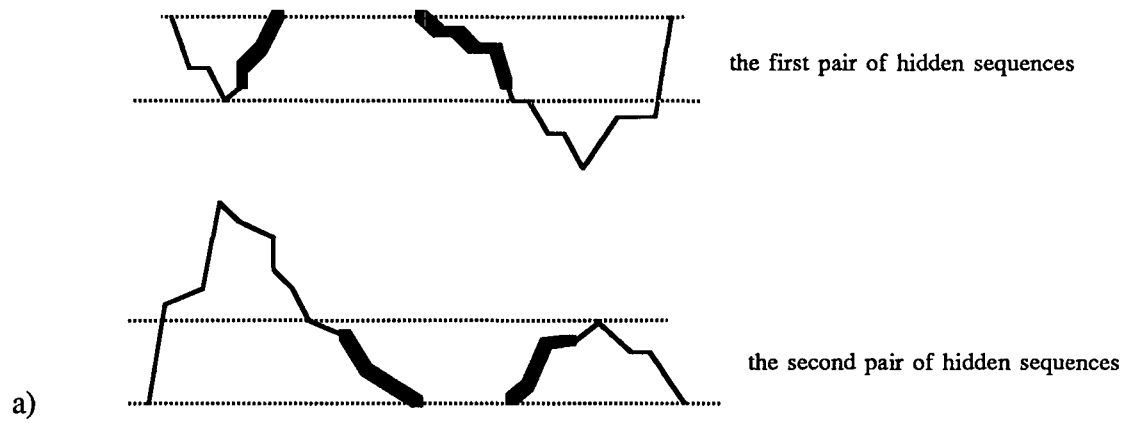
Let S be a set of edges. A point p from an edge e of S is called *left (resp., right) horizontally visible in S* if there exists a horizontal line through p such that e is the first (resp., the last) edge cut by this line. An edge which contains a point which is left (resp., right) horizontally visible in S is said to be a *left (resp., right) horizontally visible edge.* Let T be an almost balanced binary tree with leaves labelled in left to right order by elements of Y and let v be an internal node of T. Let E_v be the set of edges with at least one endpoint belonging to the set of vertices spanned by the leaves of the subtree of T rooted at v. We associate with each such node v, its left and right visibility information which consists of two, possible empty, parts :

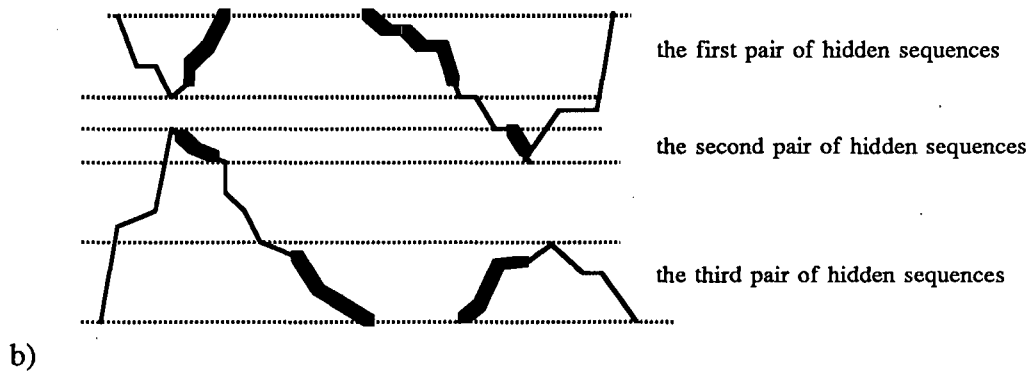
(i) the subset of edges from E_v which belong to the upper polygonal line and are horizontally visible in E_v;

(ii) the subset of edges from E_v which belong to the lower polygonal line and are horizontally visible in E_v.

It is not difficult to see that the computation of the visibility information of an internal node from the visibility information of its children can be carried out using the same method as used for the ADN problem, with an additional constant factor in the time complexity.

In a natural way we associate with each internal node at most three pairs of hidden sequences. (The three general cases are presented in Figure 2.9.) Hidden sequences can be computed using a method similar to the method described in section 2.7.2 for the ADN problem with only additional constant factor in the time complexity. As in the case of the ADN problem, every vertex of the polygon belongs to exactly one hidden sequence. Given a vertex in a hidden sequence we can find its horizontal neighbors using a method similar to the method described in section 2.7.2. Thus we reduce the problem to merging of $O(n)$ sequences of total length equal to n. As we have pointed out in the previous section this can be done in $O(\log n)$ time with $n/\log n$ CREW processors.





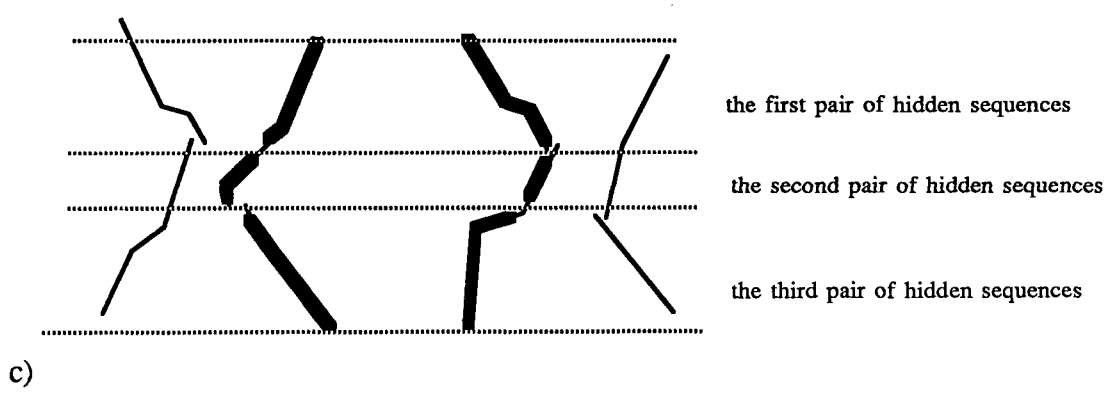


Figure 2.9. Hidden sequences (bold lines)

2.8. Summary

This chapter introduced general parallel techniques and algorithms which support our tree construction techniques. In Sections 2.1 trough 2.4, we described well-known parallel techniques: Brent's scheduling principle, list ranking, and the Euler Tour technique.

In Section 2.5, we discussed the tree contraction technique. The simple tree contraction algorithm presented in this section is joint work by Abrahamson, Dadoun, Kirkpatrick, and Przytycka [AbrDadKirPrzy87]. This work, together with other results ([ColVis86c], [GibRyt86], [KosDel88], [He86a], [MilTen87], [GibRyt88]) provides a step towards a simplification and a formalization of the tree contraction technique introduced by Miller and Reif [MilRei85] and independently by Rytter [Ryt85]. In Sections 2.5.2 and 2.5.3, we introduced (based on [AbrDadKirPrzy87], [He86a], and [MilTen87]) the Bottom-up and Top-down Algebraic Tree Computation algorithmic schemes. In example 2.7, we showed a nontrivial interpretation of the scheme. In that example, we presented an algorithm to compute a certain tree labelling which is used later in Chapter 5. In our solution we made an essential use of all components provided by our 𝔅_𝔄𝔗𝔈 scheme: functions associated with internal nodes, functions associated with edges, and nontrivial indexing of these functions.

In Section 2.6, we introduced the cascading sampling technique. The technique can support parallel algorithms based on a sequence of merging steps. As we pointed out the technique is loosely related to the cascade merging paradigm introduced by Cole [Col86] and generalized by Atallah, Cole, and Goodrich [AtaColGoo89]. This technique is used in Chapter 4 of the thesis to obtain one of the basic results of this chapter.

Finally, Section 2.7, was devoted to the parallel divide and conquer technique and to an optimal parallel algorithm for the All Dominating Neighbors (ADN) problem. The

solution to this problem is used in Chapter 5. The first optimal algorithm to solve the ADN problem was presented by Berkman et. al. ([BeBrGalSchVis89]). In Section 2.7, we presented an alternative solution based on a divide and conquer strategy. Often one can naturally parallelize a sequential divide and conquer algorithm. Thus, finding a divide and conquer algorithm for a given problem can be an important step towards the design of an efficient parallel algorithm for the problem. To obtain an optimal parallel algorithm the divide and conquer strategy has to be usually supported with other parallel techniques (e.g. Brent's principle, cascading divide and conquer [AtaColGoo87], multi-way divide-and-conquer [Goo87]). In order to obtain an optimal solution to our problem we used an interesting 3-phase divide-and-conquer strategy.

CHAPTER 3: PARALLEL TREE EXPANSION - CONSTRUCTION OF TREE REPRESENTATION FOR COGRAPHS

In this chapter we present the parallel tree expansion technique and its application to the design of a parallel algorithm to construct a tree representation for graphs belonging to the family known as cographs.

In the parallel tree expansion technique we construct a tree starting from a single node by the iterative addition of sets of vertices of degree one (leaves) or two (subdividing edges). The method is closely related to parallel tree contraction described in Section 2.5. Let T_i be the tree obtained in the i^{th} iteration. We say that T_i is an expansion of T_{i-1}. Let N be the number of steps used to construct the tree. Note that the sequence $T_N, T_{N-1}..., T_0$ is a tree contraction sequence. In this sense the tree expansion can be viewed as the reverse of the tree contraction method. We generalize further the tree expansion technique by replacing nodes and/or edges by connected subtrees.

It is well known that many problems on graphs are NP-hard [GarJoh79], however these problems can often be computed efficiently for some restricted families of graphs. It is often the case that graphs belonging to such families can be represented with the help of a tree. Such a tree representation may follow (for example for cographs and series parallel graphs) from a natural recursive definition of the graphs in the family. A different idea lies behind the tree representation for the graphs from the family of graphs known as chordal

graphs. A chordal graph (i.e. a graph without chordless cycles) can be represented with the help of the clique tree. The nodes of the clique tree are in one-to-one correspondence with the maximal cliques of the corresponding graph, and for every vertex in the graph the maximal cliques containing the given vertex form a connected subtree in the clique tree. An interesting subfamily of the family of chordal graphs is provided by k-trees. A k-tree is a graph which can be recursively constructed from a k-complete graph by adding new vertices such that each new vertex is adjacent to all vertices of an existing k-complete subgraph. A partial k-tree is a graph embeddable in a k-tree with the same vertex set. Obviously, trees are 1-trees. Series parallel graphs are partial 2-trees. Observe that the recursive definition of k-trees defines a tree structure on graphs from this family.

Given a tree representation of a graph, we may use parallel tree computation techniques (for example the tree contraction) to efficiently compute some graph properties which seem to be very difficult to compute for general graphs. This motivates the problem of designing efficient parallel algorithms to recognize membership in the given class and to construct the corresponding parse tree. He [He86b] solved this problem for the class of two terminal series parallel graphs (TTSP graphs). His algorithm constructs a binary decomposition tree if a given graph is a TTSP graph. Given a multigraph with n vertices and m edges the algorithm runs on a CRCW PRAM in $O(\log^2 n + \log m)$ parallel time using $O(n+m)$ processors. Naor, Naor and Shaffer [NaoNaoSc87] proposed parallel algorithms to construct a clique tree representation for chordal graphs and for computing properties of chordal graphs using this representation[1]. Rytter and Szymacha [RytSzy89] presented an NC algorithm for the recognition and construction of a parse tree for graphs which can be generated recursively using a context-free grammar. They also proposed

[1] Parallel algorithms for chordal graphs ware also discussed by Edenbrant ([Ede87]) and Dahlhaus and Karpinski ([DahKa86], [DahKa87]). Currently the best performance achieve the algorithms presented by Klein ([Kle89]). Dahlhaus and Karpinski ([DahKa89]) also addressed a related problem of computing Minimal Elimination Order for an arbitrary graph.

parallel algorithms for such problems as edge coloring, vertex coloring, and hamiltonian cycle, which take the parse tree as the input.

A parallel algorithm for the construction of cograph tree representation described below was first proposed by Kirkpatrick and Przytycka [KirPrz87]. This algorithm serves here as an illustration of the parallel tree expansion method. The algorithm runs in $O(\log^2 n)$ parallel time using $O(n^3/\log^2 n)$ processors on a CREW PRAM, where n is the number of vertices. In the next four sections we outline an implementation using $O(n^3/\log n)$ processors. Section 3.7 describes the reduction to $O(n^3/\log^2 n)$ processors. Independently, several other algorithms have been proposed for this problem ([Shy88], [Nov89], [AdhPen89]). Each of the algorithms uses a different method and performs a construction using time and processors resources which are comparable to those used by our algorithm.[2] Przytycka and Corneil [PrzCor89] presented an algorithm for the recognition of parity graphs which can be easily converted to an algorithm which produces a parse tree for graphs from this family. We sketch this algorithm in section 3.7.

3.1. Definitions and basic properties

A *complement reducible graph,* also called a *cograph,* is defined recursively in the following way:

(i) A graph on a single vertex is a cograph;

(ii) If G_1, G_2 are cographs, then so is their union ; and

(iii) If G is a cograph, then so is its complement.

Cographs are easily seen to satisfy the following property (cf. [CorLerSte81]) :

---

[2]Recently Novick [Nov90a] claimed to be able to reduce the processor cost to O(n2).

Property 3.1. An induced subgraph of a cograph is a cograph.

Cographs are precisely the class of graphs which do not contain P_4 as an induced subgraph (P_4 is a path of four vertices). This characterization suggests a simple parallel algorithm for the *recognition* of cographs that operates in O(1) time using $O(n^4)$ CRCW processors. Such an algorithm, however, will not necessarily reveal the simple recursive structure that is imposed by the cograph definition and exploited in many cograph algorithms.

Cographs were first introduced and studied by Lerch in [Ler71],[Ler72]. As a family of graphs, cographs have arisen independently in connection with several different mathematical problems. In the work of Sumner [Sum74], motivated by empirical logic, cographs are defined as Hereditary Dacey graphs (HD-graphs). In Burlet and Uhry [BurUhr84], cographs are referred to as 2-parity graphs and are related to a broader class of graphs called parity graphs. A family of graphs equivalent to cographs is also defined by Jung in [Jun74] and called D*-graphs. In this same paper is shown that D*-graphs are equivalent to comparability graphs of multitrees. Corneil, Lerch and Stewart [CorLerSte81] show that cographs are the underlying graphs of the family of digraphs known as transitive series parallel graphs. For more information about cographs (including various equivalent characterizations) see [CorLerSte81].

The definition of a cograph suggests a natural parse tree representation. However this way of presenting a cograph may not be unique. A unique representation is provided by the so-called *cotree* [CorLerSte81]. A cotree, T_G , is the tree presenting the parsing structure of a cograph G in the following way :

- The leaves of T_G are the vertices of G.

- The internal nodes of T_G represent the operation complement-union (that is, the graph associated with an internal node is the complement of the union of the graphs associated with its descendent nodes).
- Each internal node except possibly the root has two or more children. The root has a single child if and only if the graph is disconnected.

This representation can be constructed in $O(n+m)$ sequential time (including testing if the given graph is a cograph) [CorPerSte85]. The cotree representation is used in the formulation of linear algorithms for determining the maximum independent set, chromatic number, graph isomorphism, number of cliques and other properties of cographs (see [CorLerSte81]). Abrahamson, Dadoun, Kirkpatrick and Przytycka [AbrDadKirPrz87] and Adhar and Peng [AdhPen89] present NC algorithms for a number of such problems. These algorithms also make use of the cotree representation of cographs.

In order to simplify the description of algorithms which use the cotree representation of a cograph, each node x of a cotree T is assigned a label, label(x), in the following way:
- label(root) = 1; and
- if y is a child of x then label(y) = 1 - label(x).

Figure 3.1 illustrates a cograph G and its labelled cotree T_G. The labelling of a node x records the parity of the number of complement-union operations on the path between x and the root.

To minimize confusion, we talk about *vertices* when we refer to a graph and about *nodes* when we refer to a cotree.

The nodes of a cotree labelled by 0 are called *0-nodes* and those labelled by 1 are called *1-nodes*. We also use the following notation: n denotes the number of vertices in G, Γ_v denotes the set of neighbors of the vertex v in G, and $lca_T(v_1,v_2)$ denotes the lowest

common ancestor of nodes v_1 and v_2 in the tree T (the subscripts G and T are omitted if it is obvious to which graph or tree we refer).

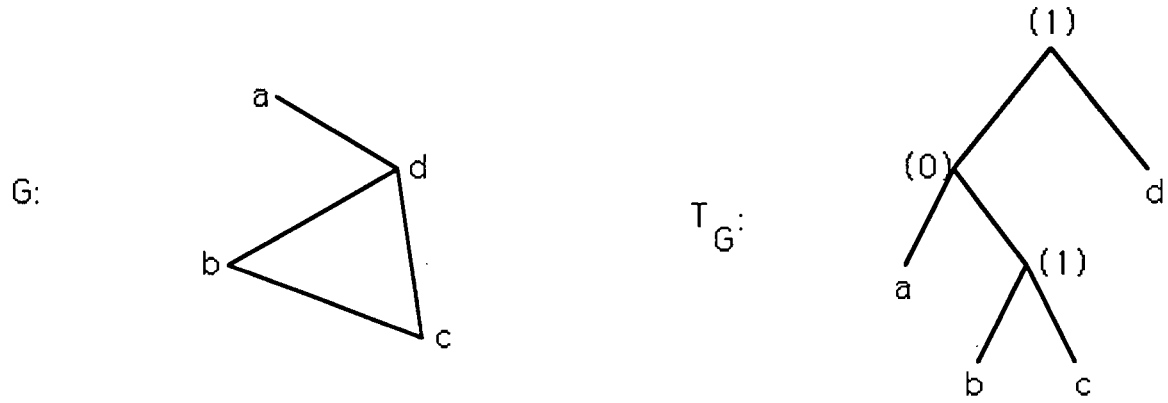


Figure 3.1. A cograph and its cotree

It is easy to confirm that :

Property 3.2. Two vertices u and v in a cograph G are adjacent iff in the cotree T defining G, the $\mathrm{lca}_T(u,v)$ is a 1-node.

To ensure readability of our figures we use the following notation:

Arbitrary subtree
(possibly empty except for the root)

Nonempty subtree

Figure 3.2. Notation concerning subtrees

3.2. Bunches and lines in a cotree

In this section we study properties of cotrees which are interesting and useful for parallel computation. In particular, we examine how the relative position of a given node in

the cotree can be determined from information about the neighborhood of the corresponding vertex in the associated cograph. We introduce notions of bunches and lines which describe collections of vertices in a cograph. These are used by our algorithm as basic building blocks for cotrees.

Define the following relations between vertices of a graph G:

(1) $S_0(u,v) \Leftrightarrow \Gamma_v - \{u\} = \Gamma_u - \{v\}$ and u,v are not adjacent,

(2) $S_1(u,v) \Leftrightarrow \Gamma_v - \{u\} = \Gamma_u - \{v\}$ and u,v are adjacent,

(3) $Z_0(v,u,w) \Leftrightarrow$ (i) $\Gamma_v \oplus \Gamma_u = \{u\}$, where $\oplus$ denotes the symmetric difference

(ii) $\Gamma_u - v \neq \Gamma_v - u$, and

(iii) w is not adjacent to either of v or to u,

(4) $Z_1(v,u,w) \Leftrightarrow$ (i) $\Gamma_v \oplus \Gamma_w = \{u\}$,

(ii) $\Gamma_v - v \neq \Gamma_u - u$, and

(iii) w is adjacent to both v and u.

The vertices satisfying relation S_i (i = 0,1) are called *siblings*. The vertices satisfying S_0 are called *weak siblings* and the vertices satisfying S_1 are called *strong siblings*. The following theorem [CorLerSte81] provides another characterization of cographs:

Theorem 3.3. G is a cograph if and only if any nontrivial induced subgraph of G has at least one pair of siblings.

Lemma 3.4. Two leaf nodes v_1 and v_2 have the same parent in the cotree T_G iff the corresponding vertices v_1 and v_2 are siblings in G.

Proof : (=>) Assume that v_1 and v_2 have the same parent in the cotree T. Note that for any vertex v such that $v \neq v_1$ and $v \neq v_2$, $\text{lca}(v,v_1) = \text{lca}(v,v_2)$. Hence, by Property 3.2, any vertex adjacent to v_1 is adjacent to v_2 and $\Gamma_{v_1} - \{v_2\} = \Gamma_{v_2} - \{v_1\}$.

(<=) Assume that v_1 , v_2 have different parents. Let $v = \text{lca}(v_1,v_2)$. At least one of the paths from v to v_i (i = 1,2) in cotree T is longer than one. Assume w.l.o.g. that the path from v to v_1 is longer then one. We can find an internal node u on this path which has a label different from v. Thus, there exists a node w ($w \neq v_1$ and $w \neq v_2$) such that $\text{lca}(w,v_1) = u$ and $\text{lca}(w,v_2) = v$. But this means that w is connected to exactly one of v_1, v_2 , so neither $S_0(v_1,v_2)$ nor $S_1(v_1,v_2)$ holds.■

A maximal set of weak siblings is called a *0-bunch set* and a maximal set of strong siblings is called a *1-bunch set*. A smallest connected subgraph of the cotree T containing a 0-bunch set is called a *0-bunch* and a smallest connected subgraph of the cotree containing a 1-bunch set is called a *1-bunch* (see Figure 3.3). The vertex with the smallest index among the vertices in a bunch set is called the *representative* of this set.

0-bunch 1-bunch

Figure 3.3. A 0-bunch and a 1-bunch

If we replace a bunch set in a cograph G by its representative, say v, then, by Property 3.1, the graph G' = (V',E') obtained in such a way is also a cograph. Consider the following construction of a tree T' from the tree T:

(1) If vertices in the bunch set are the only children of some internal node then substitute the representative of the bunch set for the bunch to which they belong (see Figure 3.4a)).

(2) If the vertices in the bunch set are not the only children of some internal node then remove from T all vertices in this set except for the representative (see Figure 3.4b)).

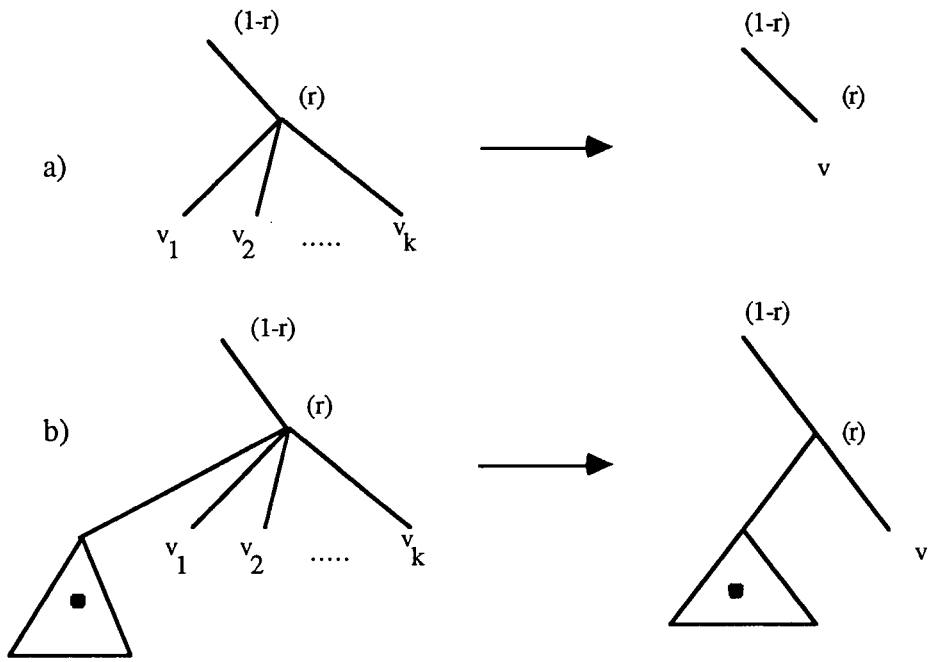


Figure 3.4. Replacing a bunch by its representative

Note: The labels of the bunch set's representative and its parent are different ensuring that this substitution is reversible.

Lemma 3.5. The tree T' obtained from the tree T in the way described above is the cotree of the cograph G'.

Proof: Note that for $u,w \in V'-\{v\}$, $label(lca_T(u,w)) = label(lca_{T'}(u,w))$. Also $label(lca_T(u,v)) = label\ (lca_{T'}(u,v))$. So, by the definition of G' and Property 3.2, the tree T' is the cotree of G'. ■

The vertices in relation Z_i (i = 0,1) also occupy special positions in the cotree. They are specified by the following lemma :

Lemma 3.6: The relation $Z_0(v,u,w)$ holds iff $v,u,$ and w are positioned in the cotree as illustrated in Figure 3.5a. Similarly, the relation $Z_1(v,u,w)$ holds iff $v,u,$ and w are positioned in the cotree as illustrated in Figure 3.5b.

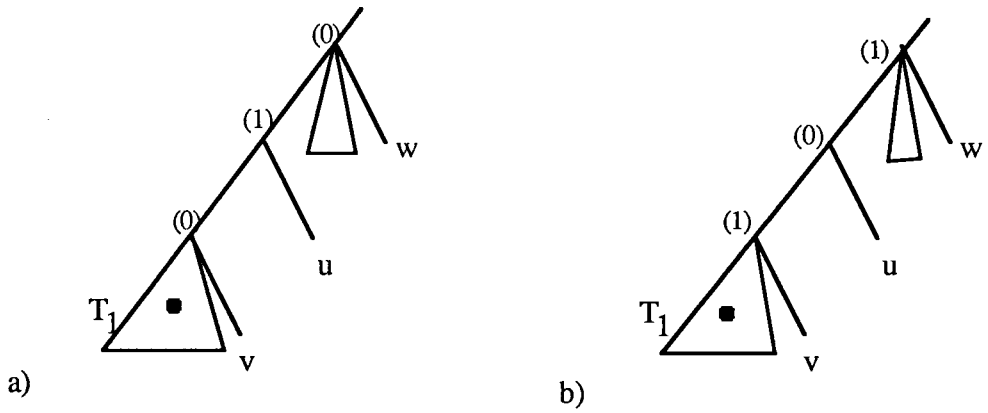


Figure 3.5. Relations $Z_0(v,u,w)$ and $Z_1(v,u,w)$

Proof : We will prove the lemma for the relation Z_0 only. The proof for the relation Z_1 is similar.

(<=) Assume that $v,u,$ and w are positioned as illustrated in Figure 3.5a. By Property 3.2, v and w have no neighbors in T_1. Assume $t \notin T_1$ and $t \neq v,u,w$ then $lca(v,t) = lca(w,t)$ and (i) follows. As immediate consequences of the cotree definition we have (ii) and (iii).

(=>) From (i) and (iii) we have that v and w are not adjacent, v and u are adjacent, u and w are not adjacent. This implies the position of the nodes as in Figure 3.6a. From (i) we know that there are no nodes between a and b, u and a , w and b, and from (ii) we have additionally $\Gamma_v \neq \Gamma_u$. This implies the more restricted position of the nodes shown in Figure 3.6b. Finally, point (i) restricts us to the positions shown in Figure 3.5a. ■

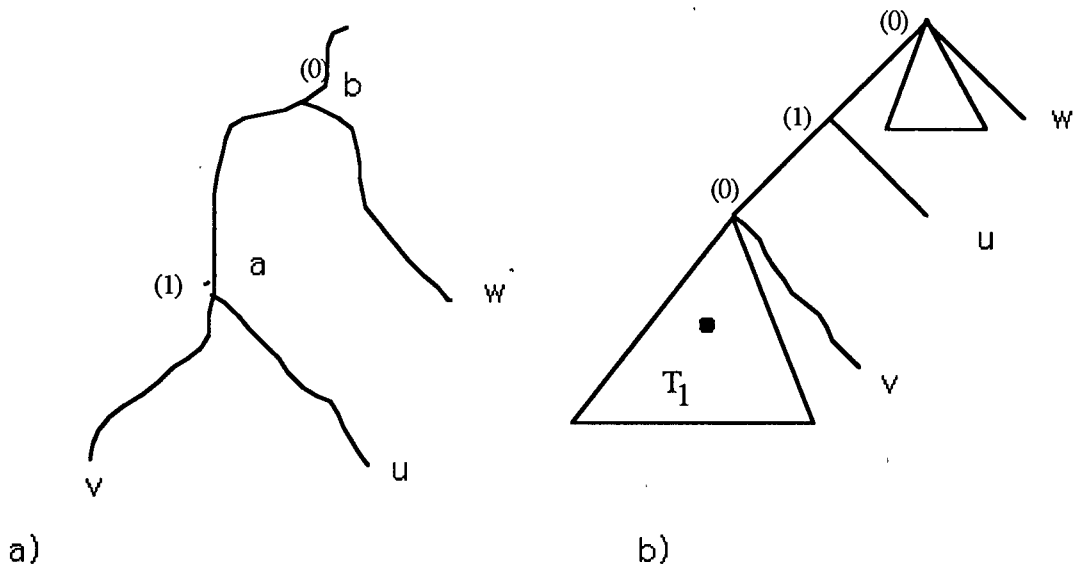


Figure 3.6. Restricting of the possible positions of nodes v,u,w satisfying $Z_0(v,u,w)$

A vertex u for which there exist vertices v,w such that $Z_0(v,u,w)$ or $Z_1(v,u,w)$ is called a *contractible vertex*. The corresponding leaf in a cotree is a *contractible leaf*. If a node has a contractible leaf as a child then it has exactly two children, one of them being a leaf and the other being a nonleaf.

A *contractible sequence* is a maximal sequence of distinct vertices (leaves in the cotree) $u_1,u_2,...,u_k$ such that there exist two vertices v,w for which $Z_i(v,u_1,u_2)$, $Z_{1-i}(u_1,u_2,u_3)$, ..., $Z_i(u_{k-1},u_k,w)$ all hold, and there does not exist an x such that $Z_{1-i}(x,v,u_1)$ holds.

Define a *branching node* as an internal node having more than two children or having more than one leaf as a child. Note that any nonbranching node appears in the cotree as node v in Figure 3.7.

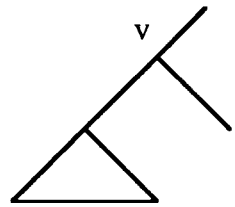
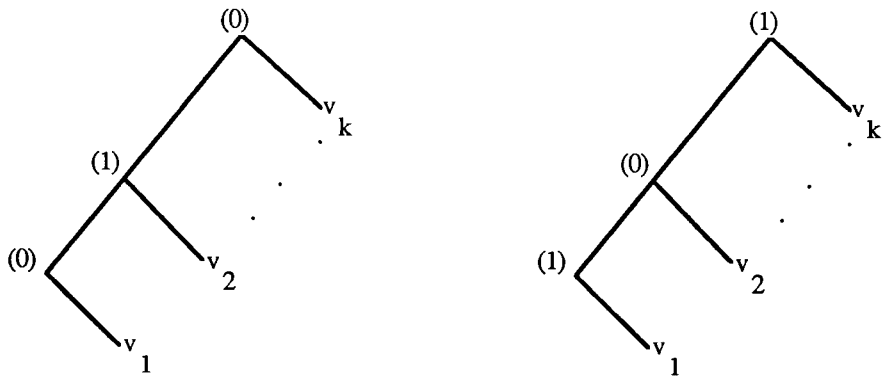


Figure 3.7. A nonbranching node

A smallest connected induced subgraph of a cotree containing a contractible sequence is called a *line* . A line is a *0-line* if the lowest level internal node is a 0-node and a *1-line* otherwise. By Lemma 3.6, lines have the form presented in Figure 3.8. The set of vertices associated with a 0-line is called a *0-line set* and the set of vertices associated with a 1-line is called a *1-line set.* The leaf which has a lowest level in the cotree among other vertices in a line (i.e. vertex v_1 in Figure 3.8) is called the *representative* of the given line set.



a) 0-line b) 1-line

Figure 3.8. A 0-line and a 1-line

Let G' be the cograph obtained from G by replacing a line set by its representative. Let T' be the tree obtained by removing from T all elements of a line set and their parents except the representative and its parent. The parent of the representative takes as its new parent the (former) parent of the highest level vertex in the line set (see Figure 3.9).

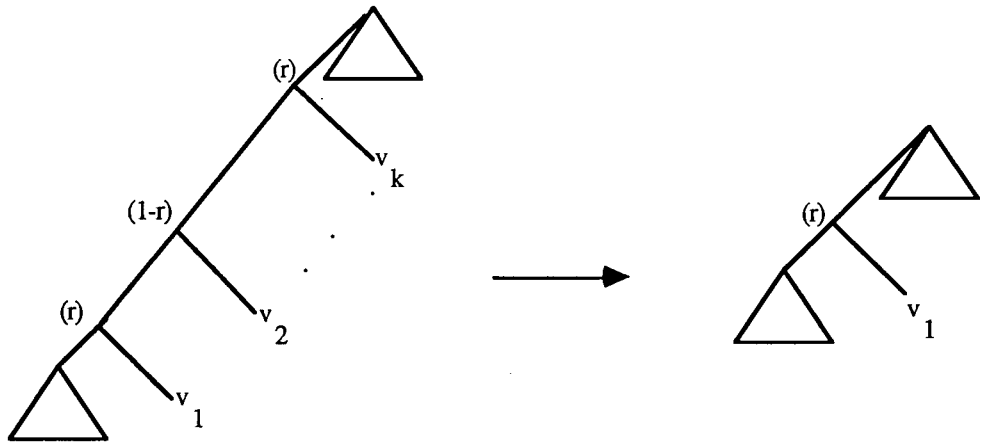


Figure 3.9. Replacing a line by its representative

Lemma 3.7. The tree T' obtained from the tree T in the way described above is the cotree of the cograph G'.

Proof : Similar to the proof of the Lemma 3.5. ■

As an example, in Figure 3.10, a,f and g are branching nodes, $\{v_1,v_2,a\}$ and $\{v_8,v_9,g\}$ induce 0–bunches and $\{v_3,v_4,v_5,b,c,d\}$ induces a 1-line. The vertex v_1 is the representative for the first bunch and the vertex v_8 for the second. The vertex v_3 is the representative for the line.

Note that in this example all line sets and bunch sets are disjoint. This is true in general as formalized by the following lemma:

Lemma 3.8. Let each of U,W be a line set or a bunch set. If $U \neq W$ then $U \cap W = \varnothing$.

Proof : By Lemmas 3.4 and 3.6, an element of a bunch set cannot belong to a line set. Note also that $S_i(w,u)$ and $S_j(u,v)$ implies $i = j$ and $S_i(v,u)$, so an element of a 0-bunch set cannot belong to a 1-bunch set. If U and W are both line sets or both bunch sets then $U \cap W \neq \varnothing$ contradicts the maximality of U and W. ■

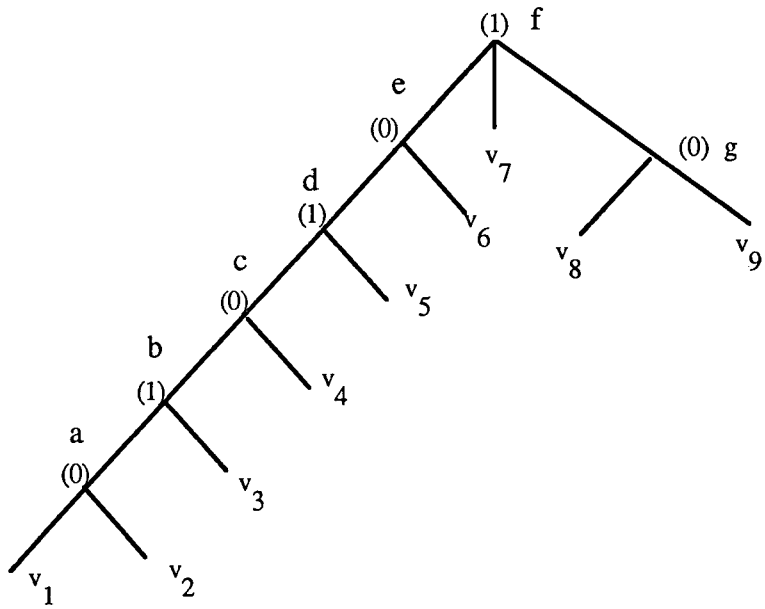


Figure 3.10. A tree containing 3 brunching nodes, 2 bunches and 1 line

3.3. A top level description of the cotree construction algorithm

We will assume that the input graph is connected. If it is not we can run a parallel algorithm for finding connected components ([HirChaSaw79]) and join the cotrees obtained for each connected component according to the cotree definition.

Let the input graph be $G_0 = (V_0,E_0)$. Assume for now that G_0 is a cograph and denote its cotree by T_0. The idea is to partition the set of vertices into subsets, remove from each subset all but one vertex (its representative), and reduce the problem to constructing the cotree for the graph induced on the diminished vertex set. Repeating this step, we obtain a sequence of graphs $G_0 = (V_0,E_0)$, $G_1 = (V_1,E_1)$,..., $G_k = (V_k,E_k)$, such that V_i is obtained from V_{i-1} by performing a partition of V_i and then removing all but one vertex in each set of the partition. The sequence should have the property that the cotree T_{i-1} for G_{i-1} can be constructed easily as an expansion of the cotree T_i for G_i. A natural approach is to partition V_i into bunch sets. Unfortunately, the length of the sequence of graphs which

is constructed in this way is proportional to the length of the longest path in the cotree T_0 which may be proportional to $|V_0|$ if T_0 is unbalanced. This is the reason why line sets must also be considered.

By Lemma 3.8, the set of vertices of a cograph can be partitioned into 0-bunch sets, 1-bunch sets, 0-line sets, 1-line sets and single vertex sets. For any set U from a partition, we can consider the smallest connected subtree of the cotree T which contains elements of this set as leaves. This subtree will be called the *fragment of T induced by* U. Note that in the proposed partition the only possible fragments are bunches, lines or single vertices.

The algorithm proceeds in stages. In stage i, it produces a triple (G_i,U_i,F_i) such that the sequence of triples produced by the algorithm satisfy the following conditions :

(i) The first element of the sequence is the triple $(G_0,\{\{v\},v\in V_0\},\{V_0\})$,

(ii) $U_i = \{U_i^1,....,U_i^t\}$ is a partition of V_{i-1},

(iii) $V_{i+1} = \{u_i^1,...,u_i^t\}$ where u_i^j is the representative of U_i^j,

(vi) $G_i=(V_i,E_i)$ is the subgraph induced by V_i,

(v) $F_i = \{F_i^1,..., F_i^t\}$ where F_i^j is the fragment of T_{i-1} induced by U_i^j,

(vi) The last element in the sequence is the first triple (G_k,U_k,F_k) for which $|U_k| = 1$.

Note that the cotree T_k is just the only fragment in F_k. For i = k,...,1, we construct cotree T_{i-1} from cotree T_i.

We define the operation *reduce* which i) partitions the vertex set into bunch sets, line sets and single vertex sets, ii) finds representatives for those sets, iii) constructs corresponding fragments, and iv) constructs the graph induced by representatives of the partition. In the next section we show how to implement this operation in polylogarithmic parallel time. In section 3.6, we outline an algorithm for constructing the adjacency matrix of a cograph from its cotree representation. In the remaining part of this section we show

that the length of the sequence $(G_0,F_0,U_0),...,(G_k,F_k,U_k)$ constructed using the reduce operation is O(log *n*) and that having this sequence we can construct the cotree T_0 in polylogarithmic parallel time.

Consider a leaf *u* of the cotree T_i. Let *r* denote a label. Let *u* be the representative of a set U in the partition U_i. The following diagram summarizes the substitutions of fragments for representatives (sometimes together with its parent) in the tree T_i to obtain the tree T_{i-1}. Their validity follows from Lemmas 3.5 and 3.7.

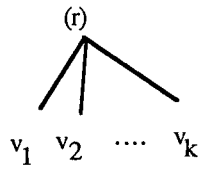
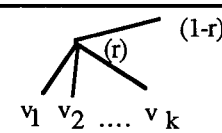
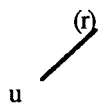
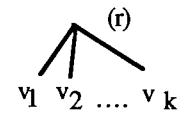
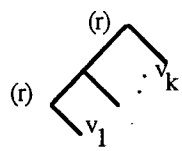
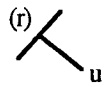
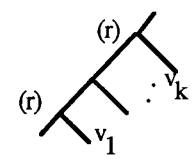
type of set represented by u	associated fragment	position of u in the cotree T_i	position of the fragment in the cotree T_{i-1}
single vertex set	. u		
r-bunch set			
			
r-line set			

Figure 3.11. Substitution of fragments for corresponding representatives

To prove that the length of the sequence is O(log *n*), we note that the operation reduce satisfies the following properties:

Property 3.9. Consider the set B of vertices in the graph (i.e. leaves in the cotree) which are in bunch sets of the partition. After a single application of the reduce operation at most $\lfloor |B|/2 \rfloor$ of these vertices remain.

Property 3.10. Let the partition have k line sets. Consider the set L of vertices in the graph which are in line sets of the partition. After a single application of the reduce operation exactly k of these vertices remain.

These properties imply the following theorem :

Theorem 3.11. After $\lceil \log_{9/10} n \rceil$ applications of the reduce operation a cograph is reduced to a single vertex.

Proof : It suffices to show that a single application of the operation reduce removes at least 1/10 of the current leaves. Suppose that the operation reduce is applied to a cograph with t vertices. Let B be the set of vertices in the cograph which are in bunch sets of the partition. Consider the following cases :

(1) $|B| \geq t/5$. Then, considering only leaves removed from bunch sets of the partition, the number of vertices left is less than or equal to $t - |B| + |B|/2 = t - |B|/2 \leq 9/10\, t$.

(2) $|B| < t/5$. Let k be the number of branching nodes in the cotree. Notice that $k \leq |B| - 1$ and the number of contractible sequences is at most k. Add the root to the set of branching nodes. With each branching node (except the root) we can associate a path of internal nodes in such a way that the first node, say v, is a branching node and the last node is the closest ancestor of v whose parent is a branching node. For every such path there are at most four leaves which are children of nodes in the path and are not contractible leaves (see Figure 3.12 for the worst case configuration). So after a reduce operation the number of leaves which are left is at most $|B|/2 + 4k + 1 \leq 9/2\,|B| - 3 < 9/10\, t$. ■

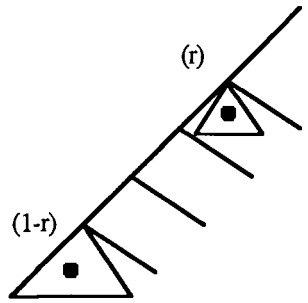


Figure 3.12. The worst case configuration for case (2)

The algorithm outlined above is based on the assumption that the given graph was a cograph. It can be modified to work without this assumption as follows:

```
(* construct the sequence of triples (Gi,Ui,Fi) *)
    M := ⌈log9/10 n⌉
    i := 0;
    U0 := { {vk} | vk∈ V};
    for 1 ≤ k ≤ n do F0k = {vk};
    while |Ui| > 1 and i ≤ M do in parallel
        i := i + 1;
        (* construct the next triple (Gi,Ui,Fi) *)
        reduce;                                   -- see section 3.4 for details
        od;

    if i > M
        then the input graph is not a cograph;

        else        (* construct the cotree *)
            Ti := Fi1;                    (* there is only one fragment in Fi *)
            while i>0 do in parallel
              i := i - 1;
              obtain Ti by substituting for each representative of the partition Ui+1 the
                    corresponding fragment from Fi+1
              od

                  (* check if correct *)
            Construct the cograph G' defined by cotree T0;   --- see section 3.5 for details
            if G'≠G0 then the input graph is not a cograph.
```

3.4. Implementation of the reduce operation

The reduce operation is used to partition the vertices of a graph into bunch sets, line sets and single vertices. It also identifies representatives of those sets and constructs corresponding fragments. We describe this operation in two phases. In the first phase, we define the main steps of the operation. In the second phase, we describe the implementation of those steps. We also note when to check conditions which might disqualify the input graph as a cograph. Some of the technical details of the implementation are left to the reader.

The reduce operation proceeds as follows :

1. For each pair of vertices v,w, check if v and w are siblings.
2. Find bunch sets, their representatives and construct corresponding bunches.
3. For each pair of vertices v,w, check for a vertex u such that $Z_i(v,u,w)$ (i=0,1). Such a vertex u is a contractible vertex. If there exists a vertex x such that $Z_{1-i}(u,w,x)$ then u,w are successive contractible vertices.
4. Find line sets, their representatives and construct corresponding lines.
5. Obtain G_{i+1} by removing from G_i all vertices not chosen as representatives.

A more detailed description of the implementation follows:

STEP 1. For each pair of vertices v,w, check if v and w are siblings.

This step can be implemented in O(log n) time with $n3$/log n processors. For each pair of vertices v,w compute the exclusive or of columns v and w of the adjacency matrix excluding rows v and w and then sum the elements of the resulting vector. This can be done for every such pair of vertices in O(log n) time with n/log n processors using the prefix sum computation technique. Store the results of this step in an n x n array A by assigning $A(v,w) = 1$ if the resulting sum is zero (i.e. v and w are siblings) and $A(v,w) =$

0 otherwise. If there exist no sibling vertices (this can be checked in O(log n) parallel time) then the graph is not a cograph.

STEP 2. Find bunch sets, their representatives and construct corresponding bunches.

Using the pointer hopping technique and the array A, each vertex can determine the vertex with the smallest index among its siblings. This can be done in O(log n) time with O($n2$/log n) processors. The unique vertex whose index is smaller than the index of its lowest indexed sibling is the representative of its bunch. The processor associated with this vertex determines its bunch-type (0-bunch or 1-bunch) and builds the parent node of the bunch. All the vertices in the bunch set construct pointers to the bunch parent (whose address is known via the representative which is known to all the vertices in the bunch set). To allow the construction of the cotree a new copy of the representative is constructed along with a pointer to its associated bunch. This copy participates in the next iteration.

STEP 3. Find successive contractible vertices.

The implementation of this step is similar to that of step 1 but in this case if the corresponding prefix sum computation produces 1 then the position at which the difference occurs indicates the vertex u. Note that the relation $\Gamma_u \neq \Gamma_v$ has been checked in the previous step and that condition iii) can be checked in constant time. First, check for the relation Z_0. Store the result in the array A by assigning $A(v,w) = u$ iff $Z_0(v,u,w)$ and $A(v,w) = 0$ if otherwise. It is possible for $A(v,w) = A(v',w') = u \neq 0$. However, if the graph is a cograph then if $A(v,w) = u \neq 0$ and $A(v,w') = u' \neq 0$ then $u = u'$. For each vertex u , it can be determined if u is an entry of A and, if so, a pair (v,w) can be chosen such that $A(v,w) = u$. (This can be done in O(log n) time using a total of O($n2$/log n) processors by exploiting the structure of A). If the pair (v,w) is chosen for vertex u then this is recorded in vector B_0 by setting $B_0(v) = u$ and $B_0(u) = w$.

In a similar way, we can check for the relation Z_1. If for vertex u the pair (v,w) satisfying $Z_1(v,u,w)$ is chosen, then it is recorded in vector B_1 by setting $B_1(v) = u$ and $B_1(u) = w$.

STEP 4. Find line sets, their representatives and construct corresponding lines.

Note that u_1,u_2 are two successive contractible vertices iff there exist vertices v and w such that $B_i(v) = u_1$, $B_i(u_1) = u_2$, $B_{1-i}(u_1) = u_2$ and $B_{1-i}(u_2) = w$. This follows from the fact that :

$$B_i(v) = u_1 \text{ , } B_i(u_1) = u_2 \Rightarrow Z_i(v,u_1,u_2) \text{ and}$$

$$B_{1-i}(u_1) = u_2, B_{1-i}(u_2) = w \Rightarrow Z_{1-i}(u_1,u_2,w).$$

Thus we can construct a table B such that $B(u_1) = u_2$ iff u_1 and u_2 are two successive contractible vertices. From this we can construct the corresponding contractible sequence. This step can be implemented in O(log n) time with O($n2$/log$n$) processors using standard pointer hopping techniques. As in the case of a bunch, we construct copies of representatives. Each copy keeps pointers to the beginning and to the end of its associated line.

Summarizing the discussion above, we have the following lemma:

Lemma 3.12. If the input graph is a cograph then we can construct its cotree in $O(\log^2 n)$ parallel time, using $O(n^3/\log n)$ processors.

3.5. Adjacency matrix construction from the cotree representation of a cograph

The algorithm to construct the adjacency matrix [AhoHopUll74] from the cotree representation of a cograph is based on Property 3.2 and is an interpretation of the $\mathfrak{T}-\mathfrak{QTC}$ algorithmic scheme.

Replace the cotree T by a binary tree T' such that newly introduced internal nodes have the same label as the node whose split led to the given node (see Figure 3.13).

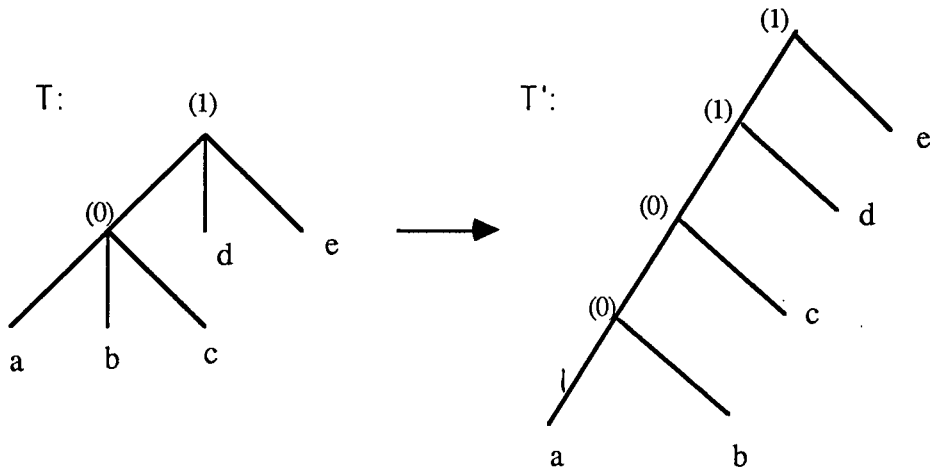


Figure 3.13. Converting a cotree to a binary tree

To construct the i-th row of the adjacency matrix we apply the instance of $\mathfrak{T}-\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ algorithmic scheme where $\mathfrak{G}$ = {id, zero, one | id(x)=x; zero(x)=0; one(x)=1} and the input tree is the tree T' with the labelling of the edges in which an edge is labelled by the constant function equal to the label of its parent vertex if the parent vertex is marked, and the identity function otherwise.

After the computation defined by this $\mathfrak{T}-\mathfrak{Q}\mathfrak{T}\mathfrak{C}$, the value computed in a leaf j ($j \neq i$) is equal to one iff i and j are adjacent in the cograph represented by T. It is easy to confirm that $\mathfrak{G}$ satisfies the decomposability axioms (i)-(ii) of $\mathfrak{T}-\mathfrak{Q}\mathfrak{T}\mathfrak{C}$, so by Theorem 2.8 the i-th row of the adjacency matrix can be constructed in O(log n) time with n/log n processors. To construct the entire adjacency matrix, O(n^2/log n) processors are used to implement this procedure for all rows in parallel.

The construction above, together with Lemma 3.13, implies the following lemma:

Lemma 3.13. We can test if an arbitrary input graph is a cograph and, if so, construct its cotree in $O(\log^2 n)$ parallel time, using $O(n^3/\log n)$ processors.

3.6. Reduction of the processor requirements

In this section, we describe a general technique which can be used to reduce the number of processors used by a CREW PRAM algorithm which computes entries of some vector and satisfies some properties stated in the following lemma. This idea is a minor generalization of a processor allocation technique used by Vishkin in [Vis84].

Lemma 3.14. Let v and w be n-vectors and let *Alg* be an algorithm consisting of $M = O(\log n)$ parallel phases, where

(i) in each phase *Alg* updates entries $v[j]$, for all j satisfying $w[j] \neq 0$, and

(ii) *Alg* sets some fixed fraction $c > 0$ of the non-zero entries of w to zero.

Suppose that each entry of v can be independently updated in $t(n)$ time using $p(n)$ processors. Then *Alg* can be implemented to run in $O((t(n)+\log\log n)\log n)$ time using $O(p(n)\, n/\log n)$ CREW processors.

Proof: An implementation using $O(t(n)\log n)$ time and $O(p(n)n)$ processors is obvious. To achieve the desired processor reduction, it is helpful to introduce a variable b which gives the number of non-zero elements of w and an array C whose ith value, for $1 \leq i \leq b$, gives the index of the ith non-zero element of w. With the help of C, it is straightforward to implement one phase of *Alg* in $O(t(n)\lceil b(\log n)/n \rceil)$time using $O(p(n)\, n/\log n)$ processors. Now, C can be updated, following the update of w, in $O(\log n)$ time using $O(n / \log n)$ processors using standard parallel prefix techniques. Hence the first $a\log\log n$ steps, where $a = 1/(\log(1/(1-c)))$, can be implemented at a total cost of $O(t(n)+ \log\log n)$ time using $O(p(n)/n \log n)$ processors. Thereafter, each step can be implemented in $O(t(n))$ time

using $O(p(n)\ n/\log n)$ processors by the straightforward implementation. Hence, the total cost is $O((t(n)+\log\log n)\log n)$ time using $O(p(n)\ n/\log n)$ CREW processors. ■

In our cotree construction algorithm, the most expensive operation is to compute relations S_i and Z_i. To compute these relations we compute the entries of array A. Note that the computations in each column of A are performed independently. Thus we can treat A as a vector of vectors and apply Lemma 3.14, assuming $p(n)=n^2/\log n$, $t(n) = \log n$ and $c = 9/10$. This construction together with Lemma 3.13 gives the following:

Theorem 3.15. We can test if an arbitrary input graph is a cograph and, if so, construct its cotree in $O(\log^2 n)$ parallel time, using $O(n^3/\log^2 n)$ processors.

The high processor cost of our algorithm follows from the cost of computing the relations S_i and Z_i. One can observe that it is possible to reduce the number of processors used in the computing of S_i by applying a sorting algorithm to rows of the adjacency matrix. Reducing the number of processors involved in computing the relation Z_i remains an open question.

3.7. An application of the cotree construction algorithm - construction of a tree representation for a parity graph

The cotree construction algorithm can be used as a fundamental building block in the construction of a tree representation for a richer family of graphs called parity graphs. A graph is called a *parity graph* if and only if for every pair of vertices all minimal chains joining them have the same parity. Parity graphs are perfect [Sac70] (i.e. for every induced

subgraph H of a parity graph the size the maximum independent set in H equals the size a minimum clique cover in H), and are a subclass of the Meyniel graphs [Mey84]. The class of parity graphs includes bipartite graphs and cographs. Burlet and Uhry [BurUhr84] noticed that parity graphs can be obtained from a single node by certain construction rules, namely the creation of weak or strong siblings and the operation called extension by a bipartite graph. Thus we can associate a parse tree with every parity graph; this parse tree, however, need not be unique.

Burlet and Uhry [BurUhr84] presented a polynomial time sequential algorithm to recognize parity graphs and construct the corresponding parse tree. They show how to reduce a given parity graph to a single vertex by operations which are reverses of the three creation operations. Their algorithm explores the BFS (Breadth First Search) layering structure of parity graphs. They proved that if $N_0,N_1,...,N_p$ are BFS layers, then the induced subgraph on each of the N_i forms a cograph. (In [BurUhr84] cographs are called 2–parity graphs.) Furthermore the relationships between vertices in neighboring layers are not arbitrary. Consider the level N_i. One can distinguish certain families of nested sets of vertices in each BFS layer with the property that elements from each set S of the family have the same set of neighbors in both the layer N_{i-1} and in the subgraphs induced by N_i-S. The relationship between elements from levels N_i and N_{i+1} is more complex but one can observe that if S is a maximally nested set then elements from S have the same set of neighbors in N_{i+1}. Thus the graph induced by such a set can only be reduced with the help of cograph operations, i.e. by the sequential removal of siblings. Furthermore such a graph is reduced either to an empty graph or to a set of independent vertices which do not have connections with the rest of the elements in the layer. The sequential algorithm of Burlet and Uhry considers the BFS layers starting from the most distant one from the root of the BFS tree, and within each layer constructs the corresponding family of subsets, and then processes them starting from a minimal one.

A nontrivial parallelization of this approach has been presented by Przytycka and Corneil [PrzCor88]. It turns out that not only can all BFS layers be processed in parallel, but also within each layer all sets in the corresponding family can be processed in parallel, treating all sets contained in a given set as "black boxes". Within each set we use only cograph operations. Thus we can construct the corresponding part of a parse tree using the techniques described in the previous sections. From the properties of the operation of extension by a bipartite graph it follows that if such a "black box" has a connection to other vertices in the level, then we are able to reduce it to the empty set with the help of an operation which is the inverse of an extension by a bipartite graph. Otherwise the elements to which the "black box" is reduced form a part of a bipartite graph which is detected and reduced (with the help of the operation which is the inverse of an extension by a bipartite graph) on other level. The algorithm given by Przytycka and Corneil runs in $O(\log^2 n)$ time with $n^4/\log^2 n$ processors on a CREW PRAM model.

The idea of using the BFS layering structure of a graph for performing parallel computation has been generalized by Novick [Nov90b] to obtain a parallel algorithm for the split decomposition of an arbitrary graph. Przytycka and Corneil [PrzCor88] showed how the BFS layering structure of a parity graph can be used to find a maximum clique in $O(\log^2 n)$ parallel time with $n^2/\log^2 n$ processors.

3.8. Summary

This chapter we presented a parallel tree expansion technique. In a parallel tree expansion technique, beginning with a single node we construct a tree by an iterative replacement of nodes and/or edges by connected subtrees. With this technique we developed the first NC algorithm for construction of the cotree representation of graphs from family of graphs known as cographs (see also [KirPrz87]). First, we identified

properties of cographs which make it possible to reduce a cograph to a single vertex such that each intermediate graph is a cograph in at most $c \log n$ steps (where c is a constant). Furthermore, the cotree of the cograph which results from a reduction step can be efficiently expanded to the cotree of the reduced cograph.

In our algorithm, we first reduce the input graph to a single vertex, then construct the cotree representation for this vertex, and finally, we interatively expand this cotree to the cotree representation of the input graph. Alternative solutions to the parallel cotree construction problem were independently obtained by several other researchers ([Shy88], [Nov89], [AdhPen89]).

In general, the basic limitation of the tree expansion method is that it may be difficult to determine whether (and how) to expand a particular edge or a vertex in a given iteration of the tree expansion process. However in the case of the cotree construction problem this method gives a simple and elegant solution.

In the last section of this chapter, we sketched an application of a cotree construction algorithm to the problem of computing in parallel a tree representation for a graph from a family of graphs known as parity graphs. In our construction, we relied on the specific structure of BFS layers of a parity graph ([BurUhr84], [PrzCor89]). Our ideas have been extended by Novick [Nov90b] to obtain a parallel algorithm for the split decomposition of an arbitrary graph.

CHAPTER 4 : PARALLEL LEVEL-BY-LEVEL TREE SYNTHESIS: CONSTRUCTION OF BINARY TREES WITH ALMOST OPTIMAL WEIGHTED PATH LENGTH

In this chapter we explore a parallel bottom-up level-by-level tree synthesis technique in connection with a family of parallel algorithms to construct trees with almost optimal weighted path length.

The idea of constructing trees which give an approximate solution to the problem of constructing of an optimal tree has been widely explored in the sequential setting. Early approximation algorithms for the optimal binary search tree problem ([Meh75], [Bay75], [Fre75]) applied a top-down technique. Allen [All82] showed that the cost of some common classes of approximately optimal binary search trees relying on a top-down approach could not be bounded above by the cost of an optimal tree plus a constant. Larmore [Lar86] presented a bottom-up subquadratic algorithm which produces a binary search tree whose cost is bounded above by the cost of an optimal tree plus a constant. Bottom-up techniques also appear to be very useful in a parallel setting. Atallah et. al. [AtaKosLarMilTen89] gave an $O(\log(1/\varepsilon)\log^2 n)$-time $n^2/\log^2 n$ processor algorithm which

finds a binary search tree whose weighted path length is within ε of that of the optimal tree. Their algorithm uses a dynamic programming technique.

Part of the motivation of looking for approximate solutions is to understand the tradeoff between the accuracy of a solution and its cost. There is an additional motivation, namely that a slight relaxation of an optimization problem, in which only an approximately optimal solution is sought, can be solved efficiently by restricting attention to trees from a certain family. In the parallel case families of trees of polylogarithmic height seem to be of special interest. Within such a family it is possible to obtain efficient parallel algorithms which produce a tree in a level-by-level fashion.

A level-by-level bottom-up technique is strictly applicable if we know the depths of the leaves of the constructed tree. In this case, we can construct the tree in d iterations, where d is the height of the tree, such that in the i^{th} iteration we construct the parents of nodes on level d-i+1. On the other hand if we know the depths of the leaves of a tree we can construct the tree in O(log n) time with n/log n processors using the accelerated valley filling technique described in Chapter 5. However we can also consider a level-by-level approach if we do not know the depths of the leaves but instead for every leaf v we can compute a value rank(v) such that rank(v)$\geq l_T$(v) (i.e. rank(v) bounds the depth of the leaf v in the constructed tree T). Then we can spread the vertices into levels according to the rank function. Now, during the i^{th} iteration of such a level-by-level construction each vertex which is on level d-i+1 either obtains a parent, which goes to level d-i, or is itself promoted to level d-i. In this chapter we apply this level-by-level construction to obtain various algorithms for producing binary trees with approximately optimal weighted path length.

The problem of finding a tree with optimal weighted path length is related to the problem of constructing an optimal code. Let $V=\{v_1,...,v_n\}$ be a set of letters and let $w(v_i)$ be the frequency of the occurrences of letter v_i in a word. The goal is to find a binary code

for every letter of V such that no code is a prefix of any other code and minimizes the average word length, defined as $\sum_{v \in V} l(v)w(v)$ (where $l(v)$ is the length of the code of v). Equivalently, one can search for a binary tree T whose set of leaves is equal to V and which minimizes the cost function $c(T)= \sum_{v \in V} l_T(v)w(v)$, where $l_T(v)$ denotes the length of the path from the root to the leaf v in the tree T. For the remainder of this chapter we call such a tree an *optimal tree*. The notion of an optimal tree extends in an obvious way to the case where w is a *weight function* such that w: V->R^+. However, in this case the problem can be reduced to the normalized problem by dividing the weight of each element v, $w(v)$, by $\mathbb{W} = \sum_{v \in V} w(v)$. Thus we assume $\mathbb{W} =1$. Note that this does not imply that the cost of an optimal tree is bounded by a constant. In fact it may achieve $\log n$ (for lower bounds on a weighted tree path see for example [Meh85]).

An optimal tree can be constructed in $O(n \log n)$ sequential time by an algorithm due to Huffman ([Huf52]). The tree produced by Huffman algorithm is called *Huffman tree*. It can be shown (see [Huf73]) that an optimal tree (whose leaves may appear in an arbitrary order) can be realized with a tree whose leaves appear in the sorted order. This observation together with the parallel dynamic programming algorithm of Miller, Ramachandran and Kaltofen [MilRefKal85] leads to an $O(\log^2 n)$-time n^6-processor parallel algorithm for construction of an optimal binary tree (see [Ten87] for a detailed description). An improved algorithm has been proposed by Atallah, Kosaraju, Larmore, Miller, and Teng [AtaKosLarMilTen89]. In their algorithm the special structure of the dynamic programming problem has been used to produce a polylogarithmic time algorithm using n^2 processors. This algorithm still does not achieve an optimal speedup over the Huffman algorithm. The question (*cf.* [AtaKosLarMilTen89]) of whether there exists a parallel algorithm which constructs an optimal tree in polylogarithmic time using $n^{2-\varepsilon}$ processors remains open. In

the same paper, an approximate solution to the problem is proposed. Let T^*_V be an optimal tree for set V and let T be an arbitrary tree with leaves equal to V. The *error of tree T,* ΔT, is defined as $c(T)-c(T^*_V)$. A tree whose error is small is called almost optimal. Atallah, Kosaraju, Larmore, Miller, and Teng [AtaKosLarMilTen89] presented an algorithm which produces a tree T with $\Delta T \leq 1$ in O(log n) time using n/log n processors if the input sequence is sorted according to the weights. Also the parallel algorithm to produce an almost optimal binary search tree presented in the same paper can be used to construct a tree T with $\Delta T \leq 1/n^k$ in $O(k\log^2 n)$ time and with $n^2/\log^2 n$ processors.

We present a family of parallel and sequential algorithms to construct an almost optimal binary tree. Each of our algorithms is an interpretation of the algorithmic scheme, called the General Construction Scheme ($\mathfrak{GCS}$), defined in Section 4.3. The use of an algorithmic scheme allows us to state a general theorem which estimates the error obtained when constructing a tree using one of our algorithms.

In Section 4.1, we present the Basic Construction Scheme ($\mathfrak{BCS}$) which defines a family of bottom-up tree constructions. This scheme leads to efficient algorithms provided that there do not exist elements of very small weight. We prove that an algorithm which is an interpretation of $\mathfrak{BCS}$ cannot produce a tree with error greater than 1. We also present a modification of $\mathfrak{BCS}$ which reduces the maximal error to 0.172. The algorithmic scheme $\mathfrak{GCS}$ defined in Section 4.3 is a modification of $\mathfrak{BCS}$. This scheme leads to efficient algorithms for sets including elements of arbitrarily small weight. In Section 4.4, we present a number of parallel interpretations of $\mathfrak{GCS}$. In particular, we give an O(log n)- time and $n\frac{\log\log n}{\log n}$-processor EREW algorithm which constructs a tree with error at most 0.172, an $O(k\log n \log^* n)$-time and n-processor CREW algorithm which produces a tree with error at most $\frac{1}{n^k}$, and an $O(k^2\log n)$-time n^2-processor CREW algorithm which produces a tree with error at most $\frac{1}{n^k}$. The algorithm obtained as the result of the second

parallel interpretation of $\mathfrak{GCA}$ achieves almost optimal speedup over the Huffman algorithm and produces a tree with a very small error. This result has been achieved by applying the cascading sampling technique presented in Section 2.6. In the Section 4.6, we present two sequential algorithms which are also interpretations of $\mathfrak{GCA}$. The first of them produces a tree with error at most $\frac{1}{n^{2k}}$ and runs in O(kn) time assuming a RAM model with bounded register capacity. The second algorithm produces a tree with error at most $\frac{1}{2^{n^{2k}}}$ and also runs in O(kn) time but it uses an integer sorting algorithm which assumes a RAM model of computation with unbounded register capacity. Section 4.7 contains some concluding remarks together with a table summarizing our results and related work.

One should be aware of another source of error which we have not addressed in this work, namely the error resulting from representation of real numbers on a computer. Our algorithms use only comparison, addition, division by 2,mod, and $\lceil\ \rceil$, with the exception of the third parallel algorithm which uses also division by n. One can show that Huffman's algorithm is numerically stable. Similarly our O(klog n log*n) algorithm is numerically stable. It is also worth noting that if the input sequence is given as a sequence of integers representing relative frequencies rather than probabilities then we can reformulate our algorithms (with the exception of the second integer sorting), so that they will perform only integer operations using words of size comparable to the size of maximal input element.

4.1. Basic Construction Scheme ($\mathfrak{BCA}$)

An optimal tree can be constructed by the following simple algorithm due to Huffman:

THE HUFFMAN ALGORITHM:

While $|V| > 1$ do
Let v_1, v_2 be the pair of elements from V of smallest weight. Construct internal node u with v_1 and v_2 as children and define $w(u)=w(v_1)+w(v_2)$.
$V:=V-\{v_1,v_2\}\cup\{u\}$.

Huffman's algorithm can be implemented to run in $O(n \log n)$ time. However, if the input sequence,V, is sorted according to weights then the Huffman tree can be constructed in linear time. Let Q be (initially empty) queue of internal nodes of the constructed tree nodes. We also treat V as a queue. Then the following algorithm constructs the Huffman tree in linear time :

CONSTRUCTION OF THE HUFFMAN TREE FROM A SORTED SEQUENCE OF WEIGHTS:

While $|Q \cup V| > 1$ do
let v_1 be a node such that $w(v_1)=\min\{w(\text{first}(Q)), w(\text{first}(V))\}$,
delete v_1 from the corresponding queue;
let v_2 be a node such that $w(v_2)=\min\{w(\text{first}(Q)), w(\text{first}(V))\}$,
delete v_2 from the corresponding queue;
Construct internal node u with v_1 and v_2 as children and define $w(u)=w(v_1)+w(v_2)$;
append u to Q.

Both algorithms presented above are highly sequential. We start by presenting an alternative way of constructing a tree isomorphic to the Huffman tree. Like Huffman's algorithm our algorithm produces the tree in a bottom-up fashion. At each stage of the algorithm we are dealing with sequences of roots of disjoint subtrees of the constructed tree. Each subtree has associated weight equal to the sum of weights of the elements in its leaves. If an element v belongs to a sequence X then *pred*(X,v) (resp., *succ*(X,v)) denotes

the element which precedes v (resp., follows v) in the sequence X and *dist*(X,v) denotes the number of elements (including v) which precede v in the sequence X. We use *first*(X) (resp. *last*(X)) to denote the first (resp. the last) element of the sequence X.

Assume that the input sequence is sorted. Observe that Huffman's algorithm can be divided into the following phases: At each phase we pair two currently smallest elements, insert the resulting element, say v, into the sequence, and then pair in sequence all (but possibly one) elements whose weight are smaller than the weight of v. Finally we merge the sequence resulting from this pairing step with the rest of the sequence. It is not difficult to implement this algorithm in O(kloglog *n*) time with n processors where k is the number of phases needed to finish the algorithm. A further modification of this approach allows for a more efficient implementation and makes it possible to obtain a family of approximation algorithms.

For every $v \in V$ we define *rank(v)* $= \lceil \log(1/w(v)) \rceil$. If rank(v) = i then we also say that *element v belongs to level i.* By convention, we think of a tree as having its root at the top and leaves at the bottom so a higher level is a level of elements of smaller rank. We divide the input sequence into subsequences $V_1, V_2, \ldots$ according to ranks. Let $I=\max\{i \mid |V_i|>0\}$. Elements of each of V_i are paired in a different step. The i^{th} step now consists of pairing the elements resulting from the previous step (the sequence U_{2i}) and merging the resulting sequence with the sequence V_{i-1}. Inductively we maintain the invariant that the elements in the sequence U_{2i} have rank at least i (i.e. they are not too big) and that the sum of the two first elements has rank at most i (i.e. it is not too small). Furthermore we design our algorithm in such a way that the cardinalities of all sequences constructed by the algorithm depends only on the cardinalities of the initial partition into sets V_i (see Lemma 4.3) This appears to be a very useful property both for the designing of approximate algorithms based on this approach and for their analysis.

Let **sort** be an increasing sorting procedure, and **merge** be a procedure which given two sorted sequences produces a sorted sequence containing all elements from both sequences. Let **pair_elements** be a procedure which given a sequence $U= u_1,...,u_{2l}$ produces a sequence $C=c_1,...,c_l$ defined as follows. For $i=1,...,l$, create a common father c_i for the pair of elements u_{2i-1}, u_{2i} defining $w(c_i)=w(u_{2i-1})+w(u_{2i})$. We use # to denote the concatenation operation on sequences, and - to denote a deletion of a subsequence from a sequence. As we prove later, the following algorithm constructs a tree isomorphic to the Huffman tree:

1. Divide elements of V into sets $V_1,V_2,....$such that $v \in V_i$ iff rank(v)=i;
2. For every i do **sort**(V_i);
3. Let $V_{i_1},...,V_{i_L}$ ($i_r<i_{r+1}$, i_L=I) be the list of nonempty sets; i:=i_L ; U_{2i}:=V_I; k:=L-1;
   - - - *i is the index of currently processed level, i_k is the index of*
   - - - *the closest nonempty level to be processed*
4. while k>0 or $|U_{2i}|$ >1 do
5.1. if $|U_{2i}|$ = 1 then U_{2i_k}:=U_{2i}#V_{i_k}; i:=i_k; k:=k-1;
   - - - *put the only element of U_{2i} into the closest nonempty level*
5.2. else c:=**pair_elements**(*first*(U_{2i}),*succ*(*first*(U_{2i}));
5.3. U_{2i-1}:=**merge**(c,U_{2i}-{*first*(U_{2i}),*succ*(*first*(U_{2i})}};
   - - - *the parent, c, of two first elements may have rank equal to i or i-1*
   - - - *so it is initially inserted into the sequence of elements of rank i;*
5.4. if $|U_{2i-1}|$ is even then c:=*pred*(*last*(U_{2i-1}))#*last*(U_{2i-1}) else c:=*last*(U_{2i-1});
5.5. C_{i-1}:=**merge**(**pair_elements**(U_{2i-1}-c),V_{i-1});
   - - - *pair elements of U_{2i-1} except for the last element (or last two*
   - - - *elements); the last element may have rank equal i-1 (compare*
   - - - *step 5.3) so should not be paired at this point;*
   - - - *merge the resulting sequence with the elements of V_{i-1}*
5.6 U_{2i-2}:=**merge**(C_{i-1},c);
   - - - *merge the remaining (one or two) elements form sequence U_{2i-1}*
   - - - *into sequence C_{i-1};*
5.7. i:=i-1; if $|V_i|$ > 0 then k:=k-1.

Lemma 4.1. The above algorithm produces a tree isomorphic to the Huffman tree.

Proof: Assume that in the above algorithm we replace the procedure **pair_elements** with a sequential procedure which pairs elements of an even length from beginning to end. It suffices to prove that during such a pairing step we always pair two smallest elements (i.e. the roots of the two subtrees of smallest weight are given a common parent). Note that all sequences occurring in the algorithm are sorted and that, for any i<k, elements in V_i are greater than elements in V_k. Note also that the following statement is an invariant of the"while" loop: elements in V_{i-1} are greater than elements in U_{2i}. This is certainly true before the first iteration. So in step 5.2 we pair two smallest elements. After step 5.3, all elements of U_{2i-1} except, possibly, the last one are smaller than any element of V_{i-1}. Also for any i the last element of U_{2i-1} is smaller than any element of V_{i-2}. Since the last element does not take part in the pairing step in line 5.5, in this step the smallest possible pair of elements is also always paired. After step 5.5 elements of C_{i-1} are smaller than elements of V_{i-2}. So elements of the sequence U_{2i-2} created in step 5.6 are also smaller than elements of V_{i-2}. From this the invariant of the "while" loop follows and consequently the fact that we always pair the smallest possible pair of elements. ■

Consider the above algorithm from a more general point of view. Replace the sorting procedure by a procedure ORDER which defines some (not necessarily sorted) order and the procedure **merge** by a procedure MERGE which given two sequences V, C produces a sequence of elements in V∪C with the property that it is sorted according to rank (but not necessarily within each rank). This produces an algorithmic scheme called the Basic Construction Scheme ($\mathfrak{BCA}$). Our initial algorithm is an interpretation of $\mathfrak{BCA}$. We call this interpretation *the Huffman tree algorithm* and denote it by H. Note that although **pair_elements** has a fixed meaning, its implementation depends on the representation of the sequences so, for consistency, we replace **pair_elements** by a procedure PAIR_ELEMENTS depending on the interpretation. When we refer to a sequence (V_i, C_i or U_i) obtained by performing $\mathfrak{BCA}$ in interpretation A, we use A as a superscript in the

name of the sequence (V_i^A, C_i^A or U_i^A respectively). Note that in fact we can use different interpretations of procedure MERGE and PAIR_ELEMENTS in different steps of an interpretation of $\mathfrak{BCA}$. If that is the case, then we indicate this by adding a subscript equal to the number of the substep of step 5 to the name of the corresponding interpretation. For example, $MERGE_3$ denotes an interpretation of the MERGE procedure used in step 5.3. So:

$\mathfrak{BCA}$ = <$\mathfrak{D}$, INITIALIZE, ORDER, MERGE, PAIR_ELEMENTS)

where

$\mathfrak{D} = \{(x_1,...,x_n) \in R^+ \times R^+ \times ....\times R^+ \mid \sum_{i=1}^{n} x_i = 1\}$;

ORDER, MERGE, and PAIR_ELEMENTS satisfy conditions described above;
(we omit a formal axiomatization of the properties of this procedures)

P: Input: array $[x_1,...,x_n]$ and array $[v_1,...,v_n]$ such that $w(v_i) = x_i$;

Output: a binary tree with leaves, V, labelled by elements from $\{x_1,...,x_n\}$

Scheme:

1. Divide elements of V into sets $V_1, V_2,....$such that $v \in V_i$ iff rank(v)=i;
2. **For every** i **do** ORDER(V_i);
3. Let $V_{i_1},...,V_{i_L}$ ($i_r<i_{r+1}$, i_L=I) be the list of nonempty sets;
   i:=i_L ; U_{2i}:=V_I; k:=L-1;
4. **while** k>0 or $|U_{2i}| >1$ **do**

5.1. **if** $|U_{2i}| = 1$ **then** U_{2i_k}:=U_{2i}#V_{i_k}; i:=i_k; k:=k-1;

5.2. **else** c:=PAIR_ELEMENTS(*first*(U_{2i}),*succ*(*first*(U_{2i}));

5.3. U_{2i-1}:=$MERGE_3$(c,U_{2i}-{*first*(U_{2i}),*succ*(*first*(U_{2i})}});

5.4. **if** $|U_{2i-1}|$ is even **then** c:=*pred*(*last*(U_{2i-1}))#*last*(U_{2i-1})
else c:=*last*(U_{2i-1});

5.5. C_{i-1}:=$MERGE_5$(PAIR_ELEMENTS(U_{2i-1}-c),V_{i-1});

5.6 U_{2i-2}:=$MERGE_6$(C_{i-1},c);

5.7. i:=i-1; **if** $|V_i| > 0$ **then** k:=k-1.

We define the level of a sequence U_j to be equal to $\lceil \frac{j}{2} \rceil$. Elements of U_j whose rank is equal to the level of U_j form the *main subsequence* of the sequence U_j. Elements of U_j whose rank is less than the level of U_j form the *head* of the sequence U_j. Elements of U_j whose rank is greater than the level of U_j form the *tail* of the sequence U_j.

Lemma 4.2: For any i the following hold:

(i) $| tail(U_{2i}) | = 0$,

(ii) $| head(U_{2i-1}) | = 0$,

(iii) $| tail(U_{2i-1}) | \leq 1$,

(iv) $| head(U_{2i}) | \leq 2$,

(v) the element in the tail of a sequence has rank one smaller than the level of the sequence,

(vi) if $|U_{2i}| > 1$ then the element whose weight is equal to the sum of the weights of two first elements in the sequence has rank equal to or one smaller than the level of the sequence.

Proof: The proof follows by induction on the level of a sequence. Consider first a sequence U_j of level I such that $j = 2I$. The sequence U_j has an empty tail and head, and all elements in U_j have rank equal to the level of U_j so (i) - (vi) are obviously true. Consider now sequence U_{2I-1}. If such a sequence is constructed then it is obtained from the sequence U_{2I} by pairing the two first elements and merging the resulting element with the rest of the elements in the sequence. It is obvious that U_{2I-1} has empty head and has a one-element tail. The rank of the element in the tail is equal to I-1. So (i) - (vi) hold also for j=2I-1.

Assume that (i) - (vi) hold for all sequences U_j of level less than i. To show point (i) note that if sequence U_{2i+1} has not been constructed then sequence U_{2i} contains elements from V_i and the only element of the last nonempty sequence U_{2k} (k>i). If U_{2i+1} has been constructed then U_j contains elements from V_i, C_i and at most two elements from U_{2i+1}. Since elements in V_i and C_i have rank equal to i and by inductive hypothesis point v)

elements in U_{2i} have rank at most i, point (i) follows. To show points (iv) and (vi) note that the elements in the head may come either from the sequence U_{2i+1} or, if U_{2i+1} has not been constructed, from the last nonempty (one element) sequence U_{2k} (k>i). In the first case (iii) follows from the fact that U_{2i+1} has no head and in the second case it follows from the fact that in this case there is only one element in the head.

Point (ii) follows from the facts (iv) and (vi) which have been proven above. Point (iii) follows from point (i) by the construction of $\mathfrak{B}\mathfrak{C}\mathcal{A}$. Point (v) follows from point (vi) by the construction of $\mathfrak{B}\mathfrak{C}\mathcal{A}$.■

Lemma 4.3 (the oblivious property) : The cardinalities of all sequences occurring in the description of $\mathfrak{B}\mathfrak{C}\mathcal{A}$ and the number of iterations performed by an interpretation of $\mathfrak{B}\mathfrak{C}\mathcal{A}$ does not depend on the interpretation.

Proof: Note that in any interpretation of $\mathfrak{B}\mathfrak{C}\mathcal{A}$ we start with the sequences $V_1,...,V_I$ such that for each i=1,...,I the cardinality of V_i does not depend on the interpretation. The cardinality of any sequence constructed by an interpretation of $\mathfrak{B}\mathfrak{C}\mathcal{A}$ depends only on the cardinalities of the sequences used for the construction and therefore, by induction, is independent of the interpretation. Similarly the number of iterations depends only on the cardinalities of the sequences and therefore is independent of the interpretation. ■

An important consequence of the above lemmas is that if T is a tree constructed by an interpretation of $\mathfrak{B}\mathfrak{C}\mathcal{A}$ then ΔT can be expressed in the following way:

Lemma 4.4: Let T be a tree obtained by some interpretation, A, of $\mathfrak{B}\mathfrak{C}\mathcal{A}$ and let

$$\Delta_{2i} = \begin{cases} 0 & \text{if } U_{2i} \text{ has not been constructed or } |U_{2i}|=1 \\ w(first(U_{2i}^{A}))+w(succ(first(U_{2i}^{A})))-w(first(U_{2i}^{H}))-w(succ(first(U_{2i}^{H})) & \text{if } |U_{2i}|>1 \end{cases}$$

$$\Delta_{2i-1} = \begin{cases} 0 & \text{if } U_{2i-1} \text{ has not been constructed or } |U_{2i-1}|=1 \\ w(last(U_{2i-1}^{H}))-w(last(U_{2i-1}^{A})) & \text{if } |U_{2i-1}| \text{ is odd and i is greater than 1} \\ w(last(U_{2i-1}^{H}))+w(pred(last(U_{2i-1}^{H})))-w(last(U_{2i-1}^{A}))-w(pred(last(U_{2i-1}^{A}))) & \text{otherwise.} \end{cases}$$

Then $\Delta T = \sum_{j=1}^{2I} \Delta_j$.

Proof: At any stage of the algorithm we are dealing with the forest consisting of the subtrees constructed so far. The roots of the subtrees belong either to the most recently constructed sequence U_j or to sequences V_i ($i<\lceil \frac{j}{2} \rceil$). Let $c(T_j^H)$ (resp., $c(T_j^A)$) be the cost of the forest such that the roots of the trees in this forest belong to the sequence U_r^H (resp., U_r^A) where U_r^H (resp., U_r^A) is the last constructed sequence such that r>j. By the definition of Δ_j we have for j<2I: $c(T_j^A)-c(T_j^H) = c(T_{j+1}^A)-c(T_{j+1}^H)+\Delta_{j+1}$. Thus $\Delta T = \sum_{j=1}^{2I} \Delta_j$. ■

We use the above lemmas to prove:

Theorem 4.5: If T is a tree obtained by an interpretation of ℬ𝒞𝒮 then $\Delta T < 1$.

Proof: Let $d_i=\Delta_{2i}+\Delta_{2i-1}$. We prove first that

$$d_i \leq \frac{1}{2^{i-1}} + \frac{1}{2^i}. \qquad (*)$$

Let $U^A_{2i} = u^A_1, u^A_2, \ldots u^A_{k_{2i}}$ be the sequence constructed by an interpretation, say A, and $U^H_{2i} = u^H_1, u^H_2, \ldots u^H_{k_{2i}}$ be the sequence produced by the interpretation H (the Huffman

tree algorithm). Note that if U_{2i} is not constructed then U_{2i-1} is also not constructed and then $d_i=0$. Alternatively consider the following four (exhaustive) cases:

1) Neither of the interpretations creates a tail. Then by Lemma 4.2 point (vi) we have $\frac{1}{2^i} \le w(u^A{}_1)+w(u^A{}_2) \le \frac{1}{2^{i-1}}$ and $\frac{1}{2^i} \le w(u^H{}_1)+w(u^H{}_2) \le \frac{1}{2^{i-1}}$. Then $\Delta_{2i} \le \frac{1}{2^i}$ and $\Delta_{2i-1} \le \frac{2}{2^i}$. So $d_i \le \frac{1}{2^{i-1}} + \frac{1}{2^i}$.

2) Both interpretations create a tail. We can interpret this case as a case when in both interpretations all but at most one (the last) of the elements are paired. Since unpaired elements have rank equal to i it follows that $d_i \le \frac{1}{2^i}$.

3) Interpretation H creates a tail and interpretation A does not create a tail. Then $\frac{1}{2^i} \le w(u^A{}_1)+w(u^A{}_2) \le \frac{1}{2^{i-1}}$ and $\frac{1}{2^{i-1}} \le w(u^H{}_1)+w(u^H{}_2) \le \frac{1}{2^{i-2}}$. Assume that $w(u^H{}_1)+w(u^H{}_2)=\frac{1}{2^{i-1}}+z$. Then $\Delta_{2i} \le -z$ and $\Delta_{2i-1} \le \frac{1}{2^i}+\frac{1}{2^i}+z$, so $d_i \le \frac{1}{2^{i-1}}$.

4) Interpretation A creates a tail and interpretation H does not create a tail. Then $\frac{1}{2^i} \le w(u^H{}_1)+w(u^H{}_2) \le \frac{1}{2^{i-1}}$ and $\frac{1}{2^{i-1}} \le w(u^A{}_1)+w(u^A{}_2) \le \frac{1}{2^{i-2}}$. Assume that $w(u^A{}_1)+w(u^A{}_2)=\frac{1}{2^{i-1}}+z$. Then $\Delta_{2i} \le \frac{1}{2^i}+z$ and $\Delta_{2i-1} \le \frac{1}{2^i}-z$, so $d_i \le \frac{1}{2^{i-1}}$.

Note that the most expensive case is case 1. However, if in this case $|U^A{}_{2i}|$ is even, then $\Delta_{2i-1} \le \frac{1}{2^i}$. If $|U^A{}_{2i}|$ is odd (and greater than 2), then this step is followed by step of type 1 such that $\Delta_{2i-1} = -\Delta_{2i-2}$. So in general we can assume that $d_i \le \frac{1}{2^{i-1}} + z_{i+1} - z_i$ where $z_i=\Delta_{2i-1}$ if U_{2i-1} has even number of elements and 1) holds and $z_i=0$ otherwise. Thus we have $\Delta T = \sum_{i=1}^{I} d_i \le \sum_{i=1}^{I} \frac{1}{2^{i-1}}$. It remains to show that $d_1=-z_2$. It is obvious that $\Delta_1=0$. If n=2 then obviously $\Delta T = 0$. Thus assume that n>2. Then V_1 has at most one element. If V_1 has one element then U_3 has at most 2 elements with neither of them in a tail

(otherwise the sum of the weights would be greater than one). Thus in this case $d_1=-z_2$. If V_1 has zero elements then U_3 either has four equal elements (and the result is obvious) or at most 3 elements. If it has 3 or fewer elements then U_2 has two elements and therefore $\Delta_2=0$. If U_3 has 2 elements then one of them must be a tail so $z_2=0$. If it has 0,1 or 3 elements then 1) does not hold and $z_2=0$ as well. ■

From the inequality (*) above, it follows that vertices of higher level may potentially contribute more to the total error. Thus it is natural to ask how far we can reduce the error if we run an approximate algorithm until we reach some level, say t, and then use the Huffman tree algorithm, i.e. when we reach level t we sort all sequences on levels t and higher, and for the remaining iterations interpret MERGE as an exact merging procedure.

To see how much we can reduce the error using this approach consider first the following problem:

Let $W=w_1,w_2,....,w_k$ where $k = n-r$, $r > 0$, and $w_1+w_2+....+w_k<1$. Let $L = 1-(w_1+w_2+....+w_k)$. A sequence $w_1,w_2,....,w_k,w_{k+1},....,w_n$ such that $w_{k+1}\geq....\geq w_n$ and $w_{k+1}+....+w_n=L$ is called an extension of W. Elements $w_{k+1},....,w_n$ are called *flexible elements*. A sequence $P=w_1,w_2,....,w_k,w_{k+1},....,w_n$ is called a *feasible extension* of the sequence W if, for every i,j such that $w_i,w_j\leq\max\{w_{k+1},....,w_n\}$, $w_i\leq 2w_j$. Denote by $\mathfrak{F}(W)$ the set of all feasible extensions of W. Assume that $\mathfrak{F}(W)$ is nonempty. Let $P\in\mathfrak{F}(W)$. We denote the cost of an optimal tree for the sequence P by C(P) . Let $\Delta W = \max_{P\,Q\in\mathfrak{F}(W)} C(P)-C(Q)$. We are going to estimate the value ΔW.

Lemma 4.6. Let f: $\mathbb{N}$ x $\mathbb{N}$->R be the function defined as $f(x,y)=\frac{L}{2x+y}$. Then there exists a pair of integers r_1,r_2 such that $r_1+r_2=n-k$ and the sequence

$$W^* = w_1, w_2, \ldots, w_k, \underbrace{2f(r_1,r_2), \ldots, 2f(r_1,r_2)}_{r_1}, \underbrace{f(r_1,r_2), \ldots, f(r_1,r_2)}_{r_2}$$

belongs to $\mathfrak{F}(W)$ and satisfies $C(W^*) = \min_{P \in \mathfrak{F}(W)} C(P)$. Furthermore there exists an optimal tree T* for the sequence W* in which all flexible leaves of weight $f(r_1,r_2)$ occur on one level, say h, and all flexible leaves of weight $2f(r_1,r_2)$ occur at level h-1.

Proof: Let $Q = w_1, w_2, \ldots, w_k, w_{k+1}, \ldots, w_n$ be a feasible extension of $W = w_1, w_2, \ldots, w_k$ which satisfies $C(Q) = \min_{P \in \mathfrak{F}(W)} C(P)$. Let T be an optimal tree for Q. Let u be the maximal flexible element of Q. First we show that there is an optimal tree in which all elements less than or equal to u occur on two consecutive levels. Assume that there are two elements $w_i, w_j \leq u$ such that w_i belongs to level h and w_j belongs to level h-s (s>1). Then the parent of w_i , say x, has weight at least twice as big as the weight of the smaller of its children. Since Q is a feasible extension of W it follows that $x \geq w_j$. Therefore we can switch w_j with x without increasing the cost of the tree. Thus there is an optimal tree for the sequence Q, say T', in which all flexible elements occur at two consecutive levels.

Let the number of flexible elements on level h-1 of tree T' be equal to r_1 and the number of flexible elements on level h be equal to r_2. We do not increase the weight of the tree if we assign the weight $2f(r_1,r_2)$ to flexible nodes on level h-1, and the weight $f(r_1,r_2)$ to flexible nodes on level h. In this way we obtain tree T* which satisfies the conditions stated in the lemma. ■

In Lemma 4.6 we constructed a sequence W* such that $C(W^*) = \min_{P \in \mathfrak{F}(W)} C(P)$. Now we are going to construct a (non-necessarily feasible) extension W' of W such that $C(W') \geq \max_{P \in \mathfrak{F}(W)} C(P)$. Let P,Q be a pair of extensions of W such that P is obtained from Q by reducing the value of a flexible element of Q, say w_i, by some value x and increasing the value of another flexible element of Q, say w_j, where $w_j \geq w_i$, by the same value x. Then we say that P is obtained from Q by an *elementary shift of weight.*

Lemma 4.7: If P is obtained from Q by an elementary shift of weight then $C(P) \leq C(Q)$.

Proof: Assume that P is obtained from Q by reducing the value of a flexible element of w, say w_i, by some value x and increasing the value of another flexible element of w, say w_j, where $w_j \geq w_i$, by the same value x. Let T be a tree which is optimal for the sequence Q. Let T' be a tree obtained from T by changing the weights of leaves corresponding to w_i and w_j by subtracting and adding x respectively. Since $l_T(w_j) \leq l_T(w_i)$ it follows that $c(T) \geq c(T')$. Furthermore the cost of T' is greater than or equal to the cost of an optimal tree for P. ■

Lemma 4.8. The sequence $W'=w_1,w_2,\ldots,w_k,\frac{L}{r},\frac{L}{r},\ldots,\frac{L}{r}$ satisfies $C(W') \geq \max_{P \in \mathfrak{F}(W)} C(P)$.

Proof: We show that for any feasible extension $P=w_1,w_2,\ldots,w_k,w_{k+1},\ldots,w_n$, there exists a sequence of elementary shifts of weight leading from W' to P. We define this sequence inductively. Let $W_i = w_1,w_2,\ldots,w_k,w_{k+1}{}^i,\ldots,w_n{}^i$ be the sequence obtained after the i^{th} elementary shift of weight $(W_0 = w' = w_1,w_2,\ldots,w_k,\frac{L}{r},\frac{L}{r},\ldots,\frac{L}{r}) = w_1,w_2,\ldots,w_k,w_{k+1}{}^0,\ldots,w_n{}^0)$. If $W_i \neq P$ then perform the following shift of weight:

Let $w_j{}^i$ be the first flexible element such that $w_j{}^i < w_j$ and $w_t{}^i$ be the last flexible element such that $w_t{}^i > w_t$. Then define W_{i+1} as follows:

for $s \neq j,t$ $w_s{}^{i+1} = w_s{}^i$;

$w_j{}^{i+1} = w_j{}^i + \min(w_j - w_j{}^i, w_t{}^i - w_t)$;

$w_t{}^{i+1} = w_t{}^i - \min(w_j - w_j{}^i, w_t{}^i - w_t)$;

(shift $\min(w_{k+1} - w_t{}^i, w_t{}^i - w_n)$ weight from $w_t{}^i$ to $w_j{}^i$). It is obvious that a finite number of such steps convert W' into P. Thus, by Lemma 4.7, $C(W') \geq C(P)$. ■

Now we are ready to prove the following theorem:

Theorem 4.9: If $W=w_1,w_2,....,w_k$ where $k<n$ and $1-(w_1+w_2+....+w_k) = L>0$ then $\Delta W \leq L(3-2\sqrt{2})$.

Proof: Let T* be the optimal tree from Lemma 4.6 and let T'' be a tree obtained from T* by replacing all flexible vertices with vertices of equal weight ($=\frac{L}{n-k}$). Of course $c(T'') \geq c(T')$ where T' is an optimal tree for the sequence where all flexible vertices are equal. So

$$\Delta W \leq c(T'')-c(T^*) = r_2(\frac{L}{r} - \frac{L}{2r_1+r_2}) = \frac{r_2 L}{n-k} \cdot \frac{n-k-r_2}{2(n-k)-r_2}.$$

Consider ΔW as a function of r_2. This function achieves its maximum, equal to $L(3-2\sqrt{2}) < 0.1716L$, for $r_2=r(2-\sqrt{2})$. Thus $\Delta W \leq L(3-2\sqrt{2})$.■

Corollary 4.10: If $\mathfrak{BCA}$ is interpreted in such a way that an approximate construction is carried out to level t-1, and the exact construction is then performed for all levels greater than or equal to t, then $\Delta T \leq \frac{1}{2^{t-2}}+0.1716$.

Proof: Let $W=w_1,...,w_k$ be the sequence of weights of elements of rank smaller than t. Then for any interpretation of $\mathfrak{BCA}$ the sequence obtained by sorting of the sequence U_{2t-1} is a feasible extension of W. By (*) and Theorem 4.9 it follows immediately that $\Delta T \leq \frac{1}{2^{t-2}} + 0.1716$.■

Note that from the above corollary it follows that if we start the exact construction from level 11 then $\Delta T \leq 0.172$.

4.2. Approximate sorting and merging of approximately sorted sequences

A sequence $u_1,u_2,...u_k$ is *ε-sorted* if and only if $\max_i(\max_{j<i}(w(u_j)-w(u_i)))\leq\varepsilon$.

A sequence $u_1,u_2,...u_k$ is an *ε-approximation* of the sequence $v_1,v_2,...v_k$ if for every i, $w(u_i)+\varepsilon \geq w(v_i) \geq w(u_i)-\varepsilon$. Note that if $u_1,u_2,...u_k$ is an ε-sorted permutation of a sorted sequence $v_1,v_2,...v_k$, then it is also an ε-approximation of the sequence $v_1,v_2,...v_k$.

A procedure performs *ε-merging* if, given two ε-sorted sequences, it produces an ε-sorted sequence.

Example 4.11. Consider the standard merging procedure applied to two ε-sorted sequences $C = c_1,c_2,...c_k$ and $V = v_1,v_2,...v_r$ (i.e. the procedure which inductively compares the first elements of the input sequences, removes the smaller of them, makes it the next element of the output sequence, and so on). Let $U=u_1,u_2,...u_{k+r}$ be the resulting sequence. To see that this procedure performs ε-merging note that for any two elements $u_i \in V$, $u_j \in C$ (resp., $u_i \in C$, $u_j \in V$) if $i<j$ then there exists an element $u_t \in C$ (resp., $u_t \in V$) such that $i < t \leq j$ and $w(u_t) \geq w(u_i)$. Since C (resp., V) is ε-sorted it follows that $w(u_t) \leq w(u_j) - \varepsilon$. So $w(u_i) \leq w(u_j) - \varepsilon$.

Example 4.12: Consider a merging procedure which first divides input sequences V and C into m subsequences $V_1,V_2,...V_m$ and $C_1,C_2,...C_m$ respectively (some of them possibly empty) such that the i^{th} subsequence contains elements whose weights are in the interval $(\frac{i}{m},\frac{i-1}{m}]$ (recall that all elements are from the interval (0,1]). The merged sequence is equal to $V_1,C_1,V_2,C_2,...,V_mC_m$. It is easy to see that this procedure first constructs $\frac{1}{m}$- sorted sequences and then performs $\frac{1}{m}$-merging of those sequences.

An important property of an ε-merging procedure is given in the following lemma:

Lemma 4.13: Let $C^A = c^A_1,c^A_2,...c^A_k$ and $V^A = v^A_1,v^A_2,...v^A_r$ be two ε-sorted sequences. Assume also that V^A is an ε-approximation of a sorted sequence V^H and that C^A is an ε-approximation of a sorted sequence C^H. Let U^A be a sequence obtained by

merging V^A and C^A by an ε-merging procedure, and let U^H be a sequence obtained by the exact merging of V^H and C^H. Then U^A is a 2ε-approximation of U^H.

Proof: Let u^A and u^H be two elements with the same index in U^A and U^H respectively. Assume without loss of generality that the number of elements from C^A in the sequence U^A which precede element u^A is less than or equal to the number of elements from C^H which precede element u^H in U^H. Then there exists j such that the j^{th} element of list V^A (v_j^A) occurs in U^A before u^A or is equal to u^A, and the j^{th} element of list V^H (v_j^H) occurs in U^H after u^H or is equal to u^H. But V^A is an ε-approximation of V^H so $w(v_j^A) \geq w(v_j^H)-\varepsilon$. However $w(v_j^H) \geq w(u^H)$ and (since U^A is ε-sorted) $w(u^A) \geq w(v_j^A)-\varepsilon$. So $w(u^H)-w(u^A) \leq 2\varepsilon$. Two prove that $w(u^A)-w(u^H) \leq 2\varepsilon$ we use a symmetric argument (with the role of C and V exchanged). ■

4.3. General Construction Scheme (𝔊ℭ𝔖)

Note that if set V contains elements of very small weights then computing ranks may become a bottleneck for any interpretation of 𝔅ℭ𝔖 . To obtain efficient implementations we divide all elements into two groups : heavy elements and light elements. More precisely let K(n) be an integer function of n. An element whose rank is less than or equal to K(n) is called a *heavy element* and an element whose rank is greater than K(n) is called a *light element*. We assume that K(n) is chosen in such a way that it is easy to decide whether a given element is light or heavy and that it is easy to compute ranks of heavy elements. Denote the set of heavy (resp., light) elements of V by V_h (resp., V_l) and let $V' = V_h \cup \{u\}$ where u is an arbitrary element of V_l.

The General Construction Scheme (𝔊ℭ𝔖) is a modification of the basic construction scheme. Roughly speaking, we use 𝔅ℭ𝔖 for the set of heavy elements and then add to

the resulting tree a subtree of light elements. (A similar approach was also taken in [AtaKosLarMilTen89] for the algorithm to construct an almost optimal binary search tree.)

GENERAL CONSTRUCTION SCHEME ($\mathfrak{GCA}$)

1. Divide set V into sets V_h and V_l.
2. Divide elements of V_h into sets $V_1, V_2, \ldots V_{K(n)}$ according to ranks
3. For every i compute ORDER(V_i).
4. Choose an arbitrary light element u and add it at the beginning of the sequence $V_{K(n)}$ (let $V'_{K(n)}$ be the resulting sequence)
5. Perform steps 3-5 of the $\mathfrak{BCA}$ for set $V' = V_h \cup \{u\}$;
6. Replace u by an almost full binary tree composed of all light elements.

Note that the error of a tree constructed by an interpretation of $\mathfrak{GCA}$ is composed of two factors: the error resulting from an interpretation of $\mathfrak{BCA}$ and the error from the light elements. We call the first component *construction error* and the second component *truncation error*. The total error can be approximated with the help of the following lemma:

Lemma 4.14: Let T' be an approximation of an optimal tree for V' and let T be the tree obtained from T' by replacing u by any binary tree of all light elements. Then $\Delta T \le \Delta T' + \frac{n^2}{2^{K(n)}}$.

Proof: Since every leaf has depth at most n in T we have:

$$c(T) \le c(T') + n \sum_{v \in V_l} w(v) \le c(T') + \frac{n^2}{2^{K(n)}} \le c(T^*_{V'}) + \Delta T' + \frac{n^2}{2^{K(n)}}$$

so $c(T) \le c(T^*_V) + \Delta T' + \frac{n^2}{2^{K(n)}}$. ■

4.4. Parallel interpretations of $\mathfrak{GCA}$

As we have mentioned before, the $\mathfrak{GCA}$ presented in the previous section can be divided into two parts: computing a tree for heavy elements and modification of the

resulting tree with a subtree of light elements. The second part can be implemented in O(log n) time with n/log n EREW PRAM processors independently of the choice of K(n). The first part involves O(K(n)) iterations of step 5 of $\mathfrak{BTA}$ so in order to obtain an efficient parallel algorithm it is natural to chose K(n)=k$\lceil \log n \rceil$ where k is some integer constant. With this definition of K(n), truncation error is bounded by $\frac{1}{n^{k-2}}$. So we concentrate on an interpretation of the first part (i.e. on an interpretation of the $\mathfrak{BTA}$ for heavy elements).

4.4.1. O(log n) time $n\frac{\text{loglog } n}{\log n}$ EREW processors parallel interpretation of $\mathfrak{gTA}$ with error bounded by 0.172

By Theorem 4.5, any interpretation of $\mathfrak{BTA}$ gives a construction error bounded by 1. By Corollary 4.9 any interpretation of $\mathfrak{BTA}$ in which starting from level t we apply the Huffman tree algorithm leads to a construction error bounded by $\frac{1}{2^{t-1}}$ + 0.1716. So, in particular, we can start with arbitrarily ordered sequences and for sequences on levels lower than t interpret MERGE as concatenation of two sequences. Then at level t, we sort all sequences and finish the construction interpreting MERGE as an exact merging procedure. We can compute ranks for heavy elements in O(loglog n) time with O(n) EREW processors by a binary search or (simulating $\frac{\log n}{\text{loglog } n}$ processors by one processor) in O(log n) time with $n\frac{\text{loglog } n}{\log n}$ EREW processors. Then, using the parallel bucket sort algorithm of Cole and Vishkin [ColVis86a] which sorts integer elements in the range [0...log n] in O(log n) time with n /log n processors, we distribute the elements to the corresponding subsets. All concatenation and pairing steps can be implemented in total of O(log n) time with n EREW processors. However, by applying Brent's scheduling principle (since we can maintain information about the position of every element within a

list, the processor allocation is not a problem) we can reduce the number of processors to $\frac{n}{\log n}$. Since there are at most 2^t elements on levels t and higher, the "exact" part of the algorithm can be implemented in $O(t \log^2 t)$ time using Cole's parallel merge sort and an EREW impelemention Valiant's parallel merging procedure. Let $t \leq \log n - \text{loglog } n$. Then the entire algorithm can be implemented in $O(\log n + t \log^2 t)$ time with $n\frac{\text{loglog } n}{\log n}$ EREW processors. In particular, if we choose t=k=11 we obtain:

Corollary 4.15: A tree whose cost differs by at most 0.172 from the cost of an optimal tree can be constructed in $O(\log n)$ time with $n\frac{\text{loglog } n}{\log n}$ EREW processors.

4.4.2 A O(klog *n* log**n*) time *n* processor parallel interpretation of 𝔊𝔗𝔖 with error bounded by $1/n^k$

We represent sequences in the form of lists such that for every element its position within the list is known. To achieve zero construction error we can implement ORDER with Cole's parallel sorting algorithm [Col86] which runs in $O(\log n)$ time using n CREW processors and MERGE with Valiant's merging algorithm ([Val75],[BorHop85]) which merges two sorted arrays of sizes n_1, n_2 $(n_1 \leq n_2)$ in $O(\text{loglog } n_1)$ time with n_1+n_2 CREW processors. This permits a straightforward implementation of step 5 of 𝔅𝔗𝔖 in $O(\text{loglog } n)$ time with n processors. By the definition of heavy elements, we have $O(k \log n)$ iterations. Thus this implementation runs in $O(k \log n \text{ loglog } n)$ time using n processors.

To reduce the running time to $O(\log n \log^* n)$ we use the cascading sampling technique described in Section 2.5. Note that we can view our solution as a sequence of parallel merging steps which, by the nature of the problem, have to be performed one after another. Thus we cannot improve the time complexity by pipelining the sequence of mergings. However, the cascading sampling techniques makes it possible to break each

merging problem into smaller merging problems which can be solved in parallel. We define *sample*(y)=y/2 and perform the preprocessing step as discussed in Section 2.6. Recall from Section 2.6 that this results in introducing new elements, called sampling elements, to each level. For each sampling element, x, there is a unique element (the source element for the element x) in the next higher level whose weight has been divided by half to obtain x. For any sampling element, x, *source*(x) is defined to be equal to the source element of x. Recall also that if u_1, u_2, is a pair of sampling elements from the same level such that there are no other sampling elements between them, then the subsequence of elements which lie between u_1 and u_2 is called a basic sequence, and the sequence of elements from the higher level which lie between the source of u_1 and the source of u_2 is called a gap. We also say that the gap defined by u_1 and u_2 corresponds to the basic sequence defined by u_1 and u_2. From Lemma 2.10 it follows that the gap size is bounded by $2^{\log^* n+2}$.

To efficiently implement one iteration , say the i^{th}, of $\mathcal{GCA}$, we have to efficiently pair real elements from level $\lceil \log n \rceil - i + 1$ and merge the resulting elements into level $\lceil \log n \rceil - i$. Similarly to the cascading functions problem, the existence of sampling elements allows us to subdivide each merging problem into subproblems such that at least one of the sequences in each merging subproblem is bounded by $2^{\log^* n+2}$. The existence of sampling elements makes the implementation of the PAIR_ELEMENTS procedure more involved. We only have to pair the real elements of a sequence. On the other hand the sampling elements (with doubled weight) must stay in the sequence after the pairing step. These elements are needed for the next merging step. Thus in our implementation of the PAIR_ELEMENTS procedure we first rearrange the sequence (not changing the relative order of real or sampling elements) in such a way that real elements which are going to be paired occur as consecutive elements. Furthermore positions of sampling elements are chosen in such a way that the list obtained by pairing real elements and doubling of weights of sampling elements is sorted. For this purpose for every element in a sequence we keep

the value *nr_r* equal to the number of real elements which precede the given element in the sequence (including the element itself), and *left_r* which is the pointer to the closest real element to the left.

To efficiently merge two sequences we have to determine, for each element in a sequence, the merging subproblem to which it belongs. For this purpose we keep, for each element, sequence pointers *left_sa* and *left_so* which define respectively the closest element to the left which is a sampling element and the closest element closest to the left which is a source element. In this way, each sampling element defines a basic sequence and its source defines the corresponding gap. We also keep, for every element in the sequence, the value *nr_so* equal to the number of sampling elements which precede the given element in the sequence (including the element itself). This information is needed to efficiently compute the functions defined above for the sequence resulting from a merging step.

This can be described in more detail as follows:

REPRESENTATION OF SEQUENCES: Each sequence is represented by a list formed by the pointers *succ*. Also, for every element of a sequence, the following information is given:

nr_sa - the number of sampling elements in the sequence, which precede the given element (including the element itself),
nr_r - the number of real elements in the sequence which precede the given element,
left_sa - the pointer to the closest sampling element to the left,
left_r - the pointer to the closest real element to the left,
left_so - the pointer to the closest element to the left which is a source element,
nr_so - the number of sampling elements in the sequence which precede the given element.

and for every sampling element we have:

source - the pointer to the source of the given element.

To simplify the description we will also assume the following pointers (these pointers can be computed in parallel using the information provided by the pointers defined above):

right_r - the pointer to the closest real element to the right,
pred - the reverse of the *succ* pointer,
$source^{-1}$ - the reverse of the *source* pointer ,
dist - the sum *nr_so+nr_r*.

INITIALIZATION: The *source* pointers are built in the sampling process. All the other information can be computed by applying the parallel prefix technique.

PAIR_ELEMENTS:

STEP 1: Rearrange the sequence in such a way that real elements which are going to be paired occur as consecutive elements.

For every sampling element x compute y1(x):=*left_r*(x); y2(x):=*right_r*(x);
A sampling element x such that *nr_r*(y1(x)) is odd is called a *skipped* element.
For every skipped element the following two tests are performed:

AFTER(x) $\Leftrightarrow$ 2w(x) $\leq$ w(y1(x))+ w(y2(x))
INSERT(x) $\Leftrightarrow$ AFTER and (w(y1(x))+ w(y2(x)) < 2w(*succ*(x))
or *succ*(x) =*right_r*(x))

The first test checks whether the two real elements y1(x), y2(x) should be inserted somewhere after x and the second test checks whether the two real elements should be inserted immediately after x. However it may happen that the pair of real elements should be inserted before all sampling elements which occur between them. To detect this case every real element y performs the following test:

FIRST(y) $\Leftrightarrow$ *nr_r*(y) is odd and w(y)+ w(*right_r*(y)) < 2w(*succ*(y))

STEP 2. For every skipped sampling element x update *left_r* and *nr_r* :
if AFTER(x) then *left_r*(x):=*left_r*(*left_r*(x)); *nr_r*(x):=*nr_r*(x)-1
otherwise *left_r*(x):=y2(x); *nr_r*(x):=*nr_r*(x)+1

STEP 3. Rearrange the elements on the list:

For every skipped element x for which INSERT(x) is true do:

succ(*pred*(y1(x))):=*succ*(y1(x)); *succ*(*pred*(y2(x))):=*succ*(y2(x));
succ(y1(x)):=y2; *succ*(y2(x)):=*succ*(x); *succ*(x):=y1(x);

For every real element y for which FIRST(y) is true do:
succ(*pred*(*right*(y))):=*succ*(*right*(y)); *succ*(*right_r*(y)):=*succ*(y); *succ*(y):=*right_r*(y)

STEP 4. For every real element which changed its place compute *left_sa, nr_sa:*

For every skipped element x for which INSERT(x) is true do:
left_sa(*succ*(x)), *left_sa*(*succ*(*succ*(x))):=x;
nr_sa(*succ*(x)), *nr_sa*(*succ*(*succ*(x))) :=*nr_sa*(x);
(we don't need pointer *left_so* for the list which is currently at the lowest level)

For every real element y for which FIRST(y) is true do
left_sa(*succ*(y)):=*left_sa*(y); *nr_sa*(*succ*(y)):=*nr_sa*(y)

STEP 5. Form a new sorted list by pairing real elements and doubling weights of sampling elements.

Elements obtained from pairing real elements are considered as real. All the functions (*nr_r, nr_sa, left_r, left_sa, succ, pred*) can be easily computed from the corresponding functions of the old list.

$MERGE_i(C,V)$ for i=3,6 is implemented as one or two insertion steps (we use the concurrent read facility to find the proper place for the inserted element).

$MERGE_5(C,V)$ is implemented as follows:

Step 1. Identify gaps and corresponding basic sequences:
- For every element of V decide (based on *left_so*) to which gap it belongs.
- For every real element of C decide (based on *left_sa*) to which basic subsequence it belongs.
- If x is a sampling element then the basic sequence defined by this element corresponds to the basic sequence defined by *source*(x).

STEP 2. Merge every basic subsequence with the corresponding gap

We use Valiant's merging algorithm. Let x be an element from a merged list and let d(x) be the number of elements in a merged list which precede x and which are from a sequence other than the sequence to which x belonged before merging. This value can be computed as the difference between the current position in the (merged) subsequence and the previous position in the gap or basic subsequence.

STEP 3. Compute the representation of the merged sequence:

- Since we know the position of every element in V we may assume that we have immediate access to every element of V. Denote the i^{th} element of sequence V by V(i).
  For every x∈ V:
  nr_sa(x), *left_sa* -remains unchanged,
  nr_r(x):=d(x)+*nr_r*(x)+*nr_r*(*source*$^{-1}$(*left_so*(x)));
  For every x∈ C:
  y:=V(*dist*(*source*(*left_sa*(x)))+d(x));
  - - - *find the closest element from V preceding x in the merged sequence*
  nr_sa(x) :=*nr_sa*(y); *left_sa*(x):=*left_sa*(y);*nr_r*(x):=*nr_r*(x)+*nr_r*(y));
- For every x: *dist*(x):=*nr_sa*(x)+*nr_r*(x)
- Computing *left_r:* Let R be an auxiliary array. For every real element x do R(*nr_r*(x)):=x (assume R(0)=null). If x is a real element then *left_r*(x):=R(*nr_r*(x)-1) otherwise *left_r*(x):=R(*nr_r*(x)).

The remaining functions can be easily computed in O(1) steps.

This finishes the description of the algorithm. We can summarize the main result of this section in the following theorem:

Theorem 4.18: A tree whose cost differs by at most $\frac{1}{n^k}$ from the cost of an optimal tree can be constructed in O(k log n log*n) time using n CREW processors.

4.4.3. $O(k^2 \log n)$ time n^2 processor parallel interpretation of $\mathfrak{GCA}$ with error bounded by $\frac{1}{n^k}$

In our first interpretation of the $\mathfrak{GCA}$ we left the elements in each of the sequences in their initial order. In our second interpretation we maintain the sorted order in all sequences. The interpretation which we present in this section lies between these two approaches. Here we approximately sort initial sequences and use an approximate merging algorithm which combines two sequences in constant time. The idea of the merging algorithm is taken from Example 4.12. First, we describe an algorithm which works in $O(k \log n)$ time with n^{6k} processors and then we show a hierarchical data structure which allows an $O(k^2 \log n)$-time n^2-processors implementation.

The idea is to partition the main subsequences (i.e. sequences with tails or heads excluded) into n^{6k} subsequences such that element x belongs to the subsequence j if and only if $w(x)-\frac{1}{2^i} \in \left[\frac{j-1}{n^{6k}}\frac{1}{2^i}, \frac{j}{n^{6k}}\frac{1}{2^i}\right)$ where i is the level of the subsequence (we treat heads and tails separately). If an element, say x, belongs to the j^{th} sublist we say that its *subrank* is equal to j (denote subrank(x)=j). In order to pair the elements in a sequence we need to know for each element the distance (or at least the parity of the distance) of a given element from the beginning of the sequence. If we pair two elements we have to compute the rank and the subrank of the resulting element. This is relatively easy if both elements have the same rank. Then the subrank of new element is approximately equal to the average of the subranks of the two initial elements. This becomes technically more involved when the paired elements have different ranks (this may happen when we pair the first two elements of a sequence). We show how this can be done in time proportional to the difference of ranks of these elements. In order to merge two sequences of the same level we concatenate corresponding subsequences (*cf.* Example 4.12). For each element in the new sequence we want to compute its distance from the beginning of the sequence. To do this it suffices, for

each element, to find an element from the other sequence which precedes the given element in the merged sequence. For this purpose, for each subsequence, we maintain the pointer to the closest nonempty subsequence following it and to the first and the last element of the given subsequence. This motivates the following representation of the sequences:

REPRESENTATION OF SEQUENCES:

For every element, x, we have:

dist(X,x) - the position of element x in the sequence X,
succ(X,x) - the successor of element x in the sequence X,

If $\mathbb{X}^j$ is a subsequence of the basic subsequence of the sequence X then

SUCC($\mathbb{X}^j$) - the closest nonempty subsequence following $\mathbb{X}^j$.
FIRST($\mathbb{X}^j$) - the first element of the subsequence $\mathbb{X}^j$.
LAST($\mathbb{X}^j$) - the last element of the subsequence $\mathbb{X}^j$.
(If a subsequence j is empty then FIRST($\mathbb{X}^j$)=LAST($\mathbb{X}^j$)=0.

INITIALIZATION: It is not difficult to construct the above data structure in O(klog *n*)time with n^{6k} processors.

PAIR_ELEMENTS: First we show how to compute the subrank of the parent, say u, of two elements u_1, u_2 in time |rank(u_2)-rank(u_1)|. We assume that for every element u we know the boundary values $\frac{1}{2^{rank(u)}}$ and $\frac{1}{2^{rank(u)-1}}$. If u_1, u_2 have the same rank then

$$\lfloor \frac{subrank(u_1)+subrank(u_2)}{2} \rfloor \leq subrank(u) \leq \lfloor \frac{subrank(u_1)+subrank(u_2)}{2} \rfloor + 1 .$$

We can compute in O(1) time the boundary values of the two possible subranks of element u and in this way determine one of two possible values. Assume that rank(u_1)-rank(u_2) = $\Delta r > 0$. In this case we normalize the subrank of the smaller element by dividing it by $2^{\Delta r}$. Note that u has rank equal to rank(u_2) or rank(u_2)-1. In the first case we have

$$\lfloor \frac{subrank(u_1)}{2^{\Delta r}} + subrank(u_2)) \rfloor \leq subrank(u) \leq \lfloor \frac{subrank(u_1)}{2^{\Delta r}} + subrank(u_2)) \rfloor + 1$$

and in the second case we have:

$$\lfloor \frac{subrank(u_1)}{2^{\Delta r+1}} + \frac{subrank(u_2)}{2} \rfloor \leq subrank(u) \leq \lfloor \frac{subrank(u_1)}{2^{\Delta r+1}} + \frac{subrank(u_2)}{2} \rfloor + 1$$

so again we have to determine one of two possible values. To be able to do this we shall compute the boundary values of the two subranks possible for u. We can compute them from the boundary values of subranks of elements u_1, u_2 using a method similar to the one described above.

Procedure PAIR_ELEMENTS$_2$ has only two elements to pair. We simply create a common parent for both of them and compute the subrank of the new element. Procedure PAIR_ELEMENTS$_5$ can be implemented as follows:

STEP 1. Create a common father u for every element u_1 whose value *dist*(U,u_1) is odd and the element u_1=*succ*(U,u_1). Let C be the resulting list.

STEP 2. For each element v, *dist*(C,v):= $\lceil$ *dist*(U,*left*(v))/2 $\rceil$.

STEP 3. Construct the representation for the list C.

For every element v from the resulting list C compute its subrank (i.e. divide C into sublists).

STEP 4.Compute the functions FIRST and LAST

To compute these functions we decide for every element of list C if it is the first and/or the last element of this subrank by comparing the subrank of the given element with the subranks of its neighbors.

STEP 5. Compute the functions PRED and SUCC:

If FIRST(C^j)≠0

then PRED(C^j):=IN(*parent*(*prec*(U,*left*(FIRST(C^j))))

else PRED(C^j):=IN(*parent*(LAST(PRED(U^j))))

where IN(x) is a function which returns the pointer to the subsequence containing element x.

Function SUCC can be computed similarly.

So the time to implement the above procedure with n^{6k} processors is $O(\Delta r_i+1)$ where Δr_i is equal to the difference of ranks of the first two elements on list U_{2i}. But $\sum_{j=1}^{I} \Delta r_i = O(K(n))$ so the total time spent on the pairing step is $O(K(n))$.

$MERGE_i(C,V)$: For i=3,6 this procedure is implemented as one or two insertion steps. If the inserted element has rank greater or smaller than the level of the sequence then it forms the tail or head of the sequence and is treated separately. For k=5 procedure $MERGE_k(C,V)$ can be implemented as follows.

STEP 1. Obtain the resulting list U
Put, for every j, elements of subrank j from list C before elements of subrank j from list V.

STEP 2. For each element v of subrank j in the list U compute $dist(U_i,v)$:

if v is an element from list C then:
dist(U,v) = $dist$(C,v) + $dist$(V, LAST(PREC(V^j))
else if FIRST(C^j)≠0
then $dist$(U,v) = $dist$(V,v) + $dist$(C,LAST(C^j))
else $dist$(U,v)= $dist$(V,v)+$dist$(C,LAST(PREC(C^j))).

STEP 3. Compute the functions FIRST, LAST, PRED, SUCC for the list U:

if FIRST(C^j)≠0 then FIRST(U^j):=FIRST(C^j)
else FIRST(U^j):=FIRST(V,j);
PRED((U^j):=max(PRED(C^j),PRED((V^j));
if LAST(C^j)=0 then LAST(U^j):=LAST(C^j)
else LAST(U^j):=LAST(V^j);
SUCC((U^j):=min(SUCC((C^j),SUCC((V^j)).

It is easy to see that procedure MERGE can be implemented in O(1) time with n^{6k} processors. This leads to the following lemma:

Lemma 4.19: A tree whose cost differs by at most $\frac{1}{n^{k-3}}$ from the cost of an optimal tree can be constructed in O(klog n) time using n^{6k} CREW processors.

Proof: The processor and time bounds follow directly from the description of the algorithm. The truncation error is $\frac{1}{n^{k-2}}$. We will show that the construction error is bounded by $\frac{8}{n^k}$. Let $d_i=\Delta_{2i}+\Delta_{2i-1}$. We prove that $d_i \le \frac{4}{2^i n^k}$ which, by Theorem 4.5, implies the result. More precisely we show, by induction, that $\Delta_{2i} \le \frac{2}{2^i n^k} \frac{2^{5(I-i)}}{2^{5k\log n}}$ and $\Delta_{2i-1} \le \frac{2}{2^i n^k} \frac{2^{5(I-i)+2}}{2^{5k\log n}}$. For every j the sequence $U^A{}_j$ constructed by the algorithm is an approximation of the corresponding sequence $U^H{}_j$ constructed by the Huffman tree algorithm. It suffices to show that $U^A{}_{2i}$ is a $\frac{1}{2^i n^k} \frac{2^{5(I-i)}}{2^{5k\log n}}$ - approximation of the sequence $U^H{}_{2i}$ and $U^A{}_{2i-1}$ is a $\frac{1}{2^i n^k} \frac{2^{5(I-i)+2}}{2^{5k\log n}}$ - approximation of the sequence $U^H{}_{2i-1}$. For $U^H{}_{2I}$ this fact is obvious. For other values of i note that, by Lemma 4.13, each application of MERGE at most doubles the approximation error. Also each application of PAIR_ELEMENTS at most doubles the approximation error. Since in $\mathfrak{GTA}$ between creation of a sequence U_{2i} and U_{2i-1} we have two calls of MERGE and PAIR_ELEMENTS, and between the creation of a sequence U_{2i-1} and U_{2i-2} we have three calls of MERGE and PAIR_ELEMENTS the result follows. ■

Note that in the above algorithm the high number of processors follows from the fact that we use one processor for each subsequence. But in any sequence there are at most n nonempty subsequences. To avoid this inefficient utilization of both space and processors, we divide a sequence into subsequences in the following recursive way. A sequence is divided into n subsequences, then every nonempty subsequence is divided into n subsubsequences, and so on (6k times). The partition of sequences is reflected by a hierarchical data structure. The number of subsequences at every level of the hierarchy is

bounded by n^2. To merge two sequences we concatenate corresponding subsequences. (Note that heads and tails of sequences have to be treated separately). The merging step is a natural simulation of the merging step described for the previous algorithm on such modified data structure. On each level of the hierarchy, we keep the pointers PRED, SUCC, FIRST, and LAST, whose meaning is similar to the n^{6k}-processor algorithm. To be able to move up and down within the hierarchical data structure we maintain, for each subsequence represented in the hierarchy, pointers UP and DOWN. For a given subsequence X the pointer UP points to the subsequence which contains X and is immediately higher in the hierarchy, and pointer DOWN points to a block of sequences defined by the next step of the recursive partition of X (see Figure 4.1).

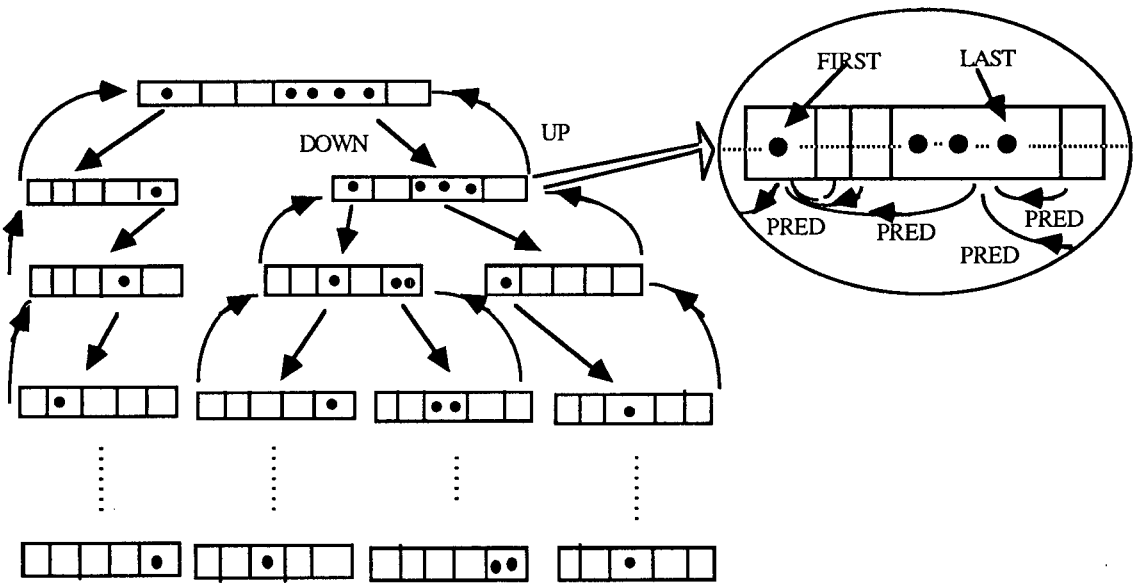


Figure 4.1. The hierarchical data structure used by the n^2-processor algorithm (not all pointers are shown)

The pairing step is more involved. To represent the sequence resulting from pairing elements of an even length sequence of elements of equal ranks, we modify the data

structure of the input sequence. During the pairing step we may obtain elements of subranks which have not been represented in the initial sequence. Thus we may be forced to extend our hierarchical structure. Some sequences may become empty and we should remove their representation from the hierarchy. However all these modifications can be carried out in time proportional to the depth of the hierarchy.

More formally:

Let X be a main subsequence of level i and let t=6k. Then

1. X is divided into n subsequences $X^1,...,X^n$ (some of which may be empty) according to the weights, such that $x \in X^j$ iff $w(x)-\frac{1}{2^i} \in [\frac{j-1}{n}\frac{1}{2^i}, \frac{j}{n}\frac{1}{2^i})$.

2. Each nonempty sequence $X^{i_1 i_2 \ldots i_r}$, where, r<t is divided into *n* subsequences

$X_1^{i_1 i_2 \ldots i_r 1}, \ldots, X_1^{i_1 i_2 \ldots i_r n}$ such that $x \in X_1^{i_1 i_2 \ldots i_r j}$ iff $w(x) - \frac{1}{2^i}(1+ \frac{i_1-1}{n} + \frac{i_2-1}{n^2} + \ldots + \frac{j-1}{n^r}) \in [\frac{j-1}{n^{r+1}}\frac{1}{2^i}, \frac{j}{n^{r+1}}\frac{1}{2^i})$.

We say that X^α is a *r^{th} order subsequence* iff α is a sequence of k indices. If $x \in X^\alpha$ and α is a sequence of r indices such that $\alpha=\beta j$ then we say that $subrank_r(x)=j$. We maintain the subsequences of each order in lexicographical order of their upper index. The sequences are represented by the following data structure:

REPRESENTATION OF SEQUENCES:

For each element, v , from the sequence X we have:

dist(X,v) - position of v in the sequence X

Let $\mathbb{X}$ be the main subsequence of the sequence X. For every subsequence $\mathbb{X}^\alpha$ we have:

FIRST($\mathbb{X}^\alpha$) - the pointer to the first element of the subsequence.

LAST($\mathbb{X}^{\alpha}$) - the pointer to the last element of the subsequence.
(If subsequence $\mathbb{X}^{\alpha}$ is empty then FIRST($\mathbb{X}^{\alpha}$)=LAST($\mathbb{X}^{\alpha}$)=0).

For a each subsequence $\mathbb{X}^{\alpha}$ of order t we have:

SUCC($\mathbb{X}^{\alpha}$) - the pointer to the closest nonempty subsequence of order t following $\mathbb{X}^{\alpha}$.

PRED($\mathbb{X}^{\alpha}$) - the pointer to the closest nonempty subsequence of order t preceding $\mathbb{X}^{\alpha}$.

The subsequences which differ only by last index are kept in an array (called a block) ordered according to the last index. For each subsequence of order smaller than t we have:

DOWN($\mathbb{X}^{\alpha}$) - the pointer to the block of subsequences into which $\mathbb{X}^{\alpha}$ is divided.

For each subsequence $\mathbb{X}^{\alpha}$ of order greater than zero (where zero is the order of whole sequence) we have:

UP($\mathbb{X}^{\alpha}$) - the pointer to the subsequence of one order lower than the order of $\mathbb{X}^{\alpha}$ which contains the given subsequence.

For every element we know its t subranks.

INITIALIZATION: Assign n processors to every element. We sort the input sequence using Cole's parallel merge sort. For every element compute all its subranks. This can be done in O(klog n) time using n processors, by k applications of binary search (performed for every element in parallel). Use the first subrank to divide sequences into first order subsequences. To compute functions FIRST and LAST it suffices to compare the subrank of every element with the subranks of its neighbors. Compute (using the prefix sum computation) the number of nonempty subsequences preceding a given subsequence. Divide every nonempty first order subsequence into second order subsequences according to the value $subrank_2$ and construct the pointer DOWN. Thus we obtain, for every nonempty subsequence, n second order subsequences (some of them possibly empty). Assign one processor for every n elements of the second order subsequence (say one of n processors associated with the first element of the subsequence of first order). For every

second level sequence construct the pointer UP (we can do it in O(1) time time with n^2 processors). Since the number of first order subsequences preceding a given subsequence is known and every first order subsequence is divided into exactly n second order subsequences, we can treat second order subsequences as consecutive elements of some array. (We can compute the position of every second level subsequence in such an array in O(1) time.) Thus we can use a prefix sum computation to compute, for every second order subsequence, the number of nonempty subsequences preceding it. Similarly we compute the subsequences of next orders, the corresponding functions FIRST, LAST, UP, DOWN, and the number of nonempty subsequences of the given order preceding given subsequence. From the last information we can compute PRED and SUCC for subsequences of order t in the following way: Use an array, say A, and assign to A(i)the i^{th} nonempty subsequence. For a subsequence X with index j in A do PRED(X):=A(j-1); SUCC(X):=A(j+1).

The initialization step can be implemented in O(klog n) time with n^2 CREW processors.

PAIR_ELEMENTS: To show the implementation of this procedure we first show how to compute in O(t |rank(u_1)-rank(u_2)|)for any two neighboring elements u_1, u_2, the subranks of their parent, say u. But it is easy to compute in O(t) time the value subrank(u) from the sequence subrank$_1$(u), subrank$_2$(u),....,subrank$_t$(u) and the opposite. Thus we can use the method presented for the n^{6k}-processor algorithm.

PAIR_ELEMENTS$_2$(U): In this case we have only two elements to pair. We simply create a common parent for both of them and compute all subranks of the new element.

PAIR_ELEMENTS$_5$(U):

STEP 1,2 : As in the n^{6k}-processor implementation

STEP 3. Construct the representat.ion for the sequence C.

Let u_1 and u_2 be a pair of elements which are given a common father, say u. To construct the data structure of the new sequence C modify the data structure of sequence U by removing elements u_1 and u_2 and inserting element u.

3.1.Compute all subranks of every newly created element u.

3.2. Find, using pointers UP, the subsequence of the highest order, say s, to which both of u_1, u_2 belong. If u belongs to a subsequence of order k+1 which was previously empty then build subsequences of orders s+2, ...,t (together with the pointers UP, DOWN). Since the sequence is a one-element sequence we can do it, for every new element, in O(s) time with n processors (using the information about subranks).

STEP 4. Compute the function FIRST and LAST for every level and every subsequence.

Use the same method as in the initialization step.Then for every subsequence check (based on the values FIRST and LAST) whether it is an empty subsequence. If yes and if the higher level subsequence containing the given subsequence is also empty then remove this subsequence from the data structure.

STEP 5. Compute the functions PRED and SUCC.

Let $IN_r(u)$ be the reference to the subsequence order r containing u. For any order r subsequence $C^{\alpha i}$ of the sequence C do

If $FIRST(C^{\alpha i})\neq 0$ then $PRED(C^{\alpha i}):=IN_t(parent(prec(U,left(FIRST(C^{\alpha i}))))$

else if the subsequence $U^{\alpha i}$ was represented in the data structure

then $PRED(C^{\alpha i}):= IN_r(parent(LAST(PRED(U^{\alpha i}))))$

else there is exactly one element, x, in the subsequence C^{α}. Let $x\in C^{\alpha j}$.

if i>j then $PRED(C^{\alpha i})=IN_r(x)$ else $PRED(C^{\alpha i})=IN_r(pred(C,x))$.

Function SUCC can be computed similarly.

$MERGE_i(C,V)$ (for i=3,6): This procedure is implemented as one or two insertion steps. If the inserted element has rank equal to the level of sequence into which it is inserted then it is added to the hierarchical data structure. Since the ranks of the element are known it can be done in O(k) time. If the rank of the inserted element is smaller or greater than the level of the sequence then the inserted element is added to the head or tail of the sequence. (Note that we never have more than two elements in a head nor more than one element in a tail).

$MERGE_5(C,V)$: Note that at this step neither of the merged sequences have a nonempty tail or head. Let U be the resulting sequence. We merge the sequences C and V in a top-down fashion:

STEP 1. Perform merging on the lowest level of the hierarchy
Using n processors, combine (for each j) V^j with C^j. If one of the subsequences is empty then U^j is represented by the nonempty one. If both V^j and C^j are nonempty then we call U^j an active subsequence of level order 1. Note that we may have at most n active subsequences.

STEP 2. Propagate the result of Step 1 to higher levels of the hierarchy
2.1.*current_order*:=1;
2.2.While *current_order*<t do
For each active subsequence of the current order recursively combine (using n processors) the subsequences of the next order (use the pointer DOWN to find the proper block of subsequences). Let V^α and C^α be the merged subsequences. If *current_order*+1 = t then put the elements of V^α before C^α. If *current_order*+1 <t and if one of the subsequences is empty then U^α is equal to the nonempty sequence. Otherwise U^α is an active subsequence of order *current_order+1*.
current_order:=*current_order*+1;

STEP 3. Compute the functions FIRST, LAST, PRED, SUCC for the lowest level.

Use the same method as in step 4 of $PAIR\_ELEMENTS_5$.

Theorem 4.20: A tree whose cost differs by at most $\frac{1}{n^k}$ from the cost of an optimal tree can be constructed in $O(k^2 \log n)$ time using n^2 CREW processors.

Proof: The algorithm is a modification of the n^{6k}-processor algorithm in which we use n^2 processors and the time of each call of MERGE or PAIR_ELEMENT is multiplied by the depth of the hierarchical data structure. Thus the time complexity is $O(k^2 \log n)$. The bound for the construction error follows from Lemma 4.19. ■

4.5. Sequential interpretations of the 𝔊𝔈𝔄

In the previous sections we presented parallel algorithms to construct an approximately optimal tree which are based on the idea of approximate sorting and merging. Here we show that the ideas developed in the previous sections can be also used to obtain linear time algorithms to construct an almost optimal tree.

A natural sequential interpretation of the general construction scheme is to obtain an ε-sorted sequence using an integer sorting algorithm and implement MERGE as standard merging procedure (recall Example 4.11). Since the cost of merging using the standard merging procedure is proportional to the sum of the lengths of the merged sequences, the total time which is spent on merging is $O(n)$. To obtain a linear time implementation of the general construction scheme we must be able to compute ranks in linear time. Since a computation of the rank of a single element requires more than constant time this certainly cannot be done by computing the ranks of the elements of the sequence one after another. Assume that the ranks are bounded by $cf(n)$ and that in linear time we can sort integers in the range $[0,2^{cf(n)}]$. Then we associate the value $\lceil w(v_i)2^{cf(n)} \rceil$ with element v_i and sort the elements of the sequence according to this value. Now we are going to merge into this sequence, a sequence of so called *boundary elements*. Boundary elements are defined in such a way that elements between two consecutive boundary elements have the same rank. However we have to keep the number of boundary elements linear in the size of the input

sequence. On the other hand we would like to choose the boundary elements in such way that the difference between them is as small as is needed to distinguish two elements with different values $\lceil w(v_i)2^{cf(n)} \rceil$. To satisfy both these condition we use a hierarchical approach similar to the one used in the previous section for the n^2-processor algorithm. We first merge the sequence with a "sparse" subset of boundary elements and interatively add more boundary elements by merging more additional boundary elements into nonempty subsequences bounded by two boundary elements. Following this idea we can compute ranks in $O(n\log\frac{cf(n)}{n})$ time by the following algorithm:

1. Sort the sequence according to $\lceil w(v_i)2^{cf(n)} \rceil$,
2. Merge the resulting sequence with the following sequence of boundary elements 1, $2^{1\lceil \frac{f(n)}{n} \rceil c}$, $2^{2\lceil \frac{f(n)}{n} \rceil c}$, ..., $2^{n\lceil \frac{f(n)}{n} \rceil c}$. Let X_1 be the resulting sequence.
3. i:=1;
4. while $i \le \log(\lceil \frac{f(n)}{n} \rceil c)$:
   4.1. Let X'_{i+1} be the sequence obtained from X_i by removing those boundary elements which have boundary elements immediately before and immediately after them;
   4.2. Each boundary element belonging to X'_{i+1} which has a real element immediately before it, defines a new boundary element equal to the geometric average of its value and the value of the closest boundary element which occurs in X'_{i+1} before the given boundary element. Denote by B_{i+1} the sorted sequence of these new boundary elements;
   4.3. Obtain X_{i+1} by merging X'_{i+1} and B_{i+1};
   4.4. i:=i+1;
5. Now every non-boundary element is between two boundary elements which are consecutive powers of two. Elements which are between 2^i and 2^{i+1} have rank cf(*n*) - i.

If we assume f(*n*)=log *n* and $c=11\cdot 2^k$ we can use the integer sorting algorithm of Kirkpatrick and Reisch [KirRe84] which sorts integers in the range $[0,n^c]$ in $O(n(1+\log c))$ time. As a result we obtain a $\frac{1}{n^c}$-sorted sequence. If we apply ℑℭ𝒮 with $K(n)=2^k\log n$ then we can compute the ranks of heavy elements in O(k*n*) time with the help of the algorithm described above. This interpretation leads to the following theorem:

Theorem 4.21: A tree T such that $\Delta T \leq \frac{1}{n^{2^k}}$ can be constructed in O(kn) time.

Proof: It is obvious that the algorithm presented above runs in O(kn) time. The truncation error is at most $\frac{1}{n^{2^k-2}}$. It suffices to show that the construction error in this algorithm is at most $\frac{8}{n^{2^k}}$. Using the same technique as in the proof of Lemma 4.19 we can show that $\Delta_{2i} \leq \frac{2}{2^i n^{2^k}} \frac{2^{5(I-i)}}{2^{10 \cdot 2^k \log n}}$ and $\Delta_{2i-1} \leq \frac{2}{2^i n^{2^k}} \frac{2^{5(I-i)+2}}{2^{10 \cdot 2^k \log n}}$ which implies the result. ■

If we assume a RAM with unbounded register capacity as our computation model then we can sort n integers in the range $[0,2^{cn})$ in $O(n(1+\log c))$ time ([KirRe84]). If we assume $c=7 \cdot 2^k$, $f(n)=n$, and $K(n)=2^k n$, then we can use the algorithm described at the beginning of this section to obtain a $\frac{1}{2^{cn}}$-sorted sequence of heavy elements and to compute ranks of the heavy elements in O(kn) time. This construction leads to the following theorem:

Theorem 4.22: A tree T such that $\Delta T \leq \frac{1}{2^{n2^k}}$ can be constructed in O(kn) time on a RAM with unbounded register capacity.

Proof: This uses exactly same techniques as in the proof of Theorem 4.21. ■

Like the parallel algorithms presented in previous sections, our sequential algorithms have been stated as explicit interpretations of the $\mathfrak{GCA}$ scheme. In the parallel setting we made an essential use of the fact that elements are divided into subsets according to their ranks. This allowed us to determine which elements can be processed in parallel. In the sequential setting the explicit computation of ranks is not necessary. In this setting the fact that the computation is naturally divided into phases corresponding to consecutive iteration of the "while" loop was helpful in the analysis of error but was not essential for the computation itself. In fact, if we restrict our attention to heavy elements only, the tree

produced by our algorithm that computes ranks explicitly is identical with the tree produces by linear time algorithm described in Section 4.1 applied to the approximately sorted sequence.

4.6. Summary

Traditionally a level-by-level tree construction algorithm consists of d steps (where d is the height of the constructed tree) such that at step i the internal nodes on depth d-i are constructed. This approach, in particular, requires that we know the depths of all leaves of the constructed tree. However, if these depths are known, we can construct the corresponding tree in O(log n) time with n/log n processors using the accelerated valley filling technique presented in Chapter 5.

In this chapter, we presented a different approach to a level-by-level tree construction. In our approach, we assume that for each leaf v of the constructed tree we can compute value rank(v) which bounds from above the depth of the leaf in the constructed tree. During the i^{th} iteration of such a level-by-level construction each vertex which is on level d-i+1 either obtains a parent, which goes to level h-i, or is itself promoted to level d-i. The basic limitation of this method is that it can be used to obtain efficient parallel algorithms only for constructing trees of polylogarithmic height. However, even if the tree which we want to construct is potentially unbalanced, there exists a tree of polylogarithmic height which closely approximates the properties of the tree we are interested in.

We have concentrated on parallel algorithms for constructing binary trees with almost optimal weighted path length. Since for many applications the weights of leaves are computed with finite precision or are assigned as the outcome of some experiment, and therefore already admit some error, this approach is justified. We approximate an optimal

solution with the help of a tree of logarithmic depth. The problem of finding a binary tree with optimal weighted path length admits a simple and elegant sequential $O(n \log n)$-time solution due to Huffman [Huf52]. The best currently known parallel algorithm for the problem is due to Atallah et. al ([AtaKosLarMilTen89]). It uses $O(\log^2 n)$ time and $n^2/\log n$ processors. With the parallel level-by-level technique described above we have been able to develop a family of efficient parallel algorithms for the relaxed version of the problem, in which only an approximately optimal solution is sought.

	time t(n)	processors p(n)	error	
1.	$O(\log^2 n)$	n^2	0	+
2.	$O(\log n)$	n	1	+
3.	$O(\log n)$	$n\frac{\log\log n}{\log n}$	0.172	*
4.	$O(k\log^* n \log n)$	n	$\frac{1}{n^k}$	*
5.	$O(k^2 \log n)$	n^2	$\frac{1}{n^k}$	*
6.	$O(k\log^2 n)$	$n^2/\log^2 n$	$\frac{1}{n^k}$	+
7.	$O(kn)$	1	$\frac{1}{n^{2^k}}$	*
8.	$O(kn)$[1]	1	$\frac{1}{2^{n^{2^k}}}$	*

Figure 4.2. The summary of results of Chapter 4

We summarize the results of this chapter and the related work in Figure 4.2 (* stands for results presented in this chapter, + stands for results obtained in or following from [AtaKosLarMilTen89]). In particular, the algorithm corresponding to the fourth entry in the table produces an exact solution if the weights of all leaves are bounded from below by $\frac{1}{n^k}$.

[1]On aRAM with unbounded register capacities

All of the algorithms presented in this chapter are interpretations of one algorithmic scheme. This makes it possible to estimate the error made by each of them using one universal lemma. The same analysis is applicable to the sequential algorithms presented at the end of the chapter. Obviously, one can propose different approximation algorithms based on the idea of approximate sorting and merging which are not interpretations of our algorithmic scheme. However, in general the accurate estimation of the error associated with an approximation algorithm is significantly more challenging than the formulation of such algorithms. For this reasons we see the formulation of the approximation scheme introduced in this chapter as being at least as significant as any of its instantiations tabulated above.

CHAPTER 5 : VALLEY FILLING TECHNIQUE AND CONSTRUCTION OF MINIMAX TREES

The parallel level-by-level tree synthesis method presented in Chapter 4 can be used to obtain efficient parallel algorithms for the construction of trees of polylogarithmic height. In this chapter we present a different tree synthesis technique which is applicable to weighted trees of arbitrary height. The technique, which we call *accelerated valley filling,* can be viewed as the parallel analogue of the method used in the T-C algorithm (the Hu-Tucker algorithm) ([HuTuc71], [Hu75]) for constructing alphabetic trees with optimal weighted path length. A simpler version of this parallel technique has been used in [AtaKosLarMilTen89] and we refer to the method used there as the *basic valley filling technique.* The technique basically consists of dividing the sequence of leaf weights into so-called *valley sequences,* and computing the parts of the tree corresponding to each valley sequence independently by performing O(log n) iterations of what we call a *valley filling step.* During a valley filling step we construct a sequence of subtrees such that the sequence of weights of their roots has its number of valleys reduced by at least half. The basic idea behind the *accelerated valley filling technique* is to speed up the basic algorithm by replacing the iterative application of the valley filling step by one pipelined parallel step.

We apply our technique to obtain an optimal algorithm for the problem of constructing minimax trees. Let $V=\{v_1,...,v_n\}$ be the set of leaves of a binary tree and w:V–>N be a *weight function.* The *(non-alphabetic) minimax tree* problem takes a set of

leaves and a weight function w and constructs a binary tree which minimizes the cost function $c(T) = \max_{v \in V}(w(v)+l_T(v))$ where $l_T(v)$ denotes the depth of the leaf v in the tree T.

Similarly, the *alphabetic minimax tree problem* takes an *ordered* sequence of leaves and a weight function w and finds an alphabetic (ordered) binary tree which minimizes the value c(T). If W is a sequence of weights then we use C(W) to denote the cost of an alphabetic minimax tree whose leaves are weighted with the elements of W in the left-to-right order. Note that the weight function can be naturally extended to internal nodes by associating with an internal node the weight equal to the maximum of the weights of its children plus one. In this way, a minimax tree can be seen to minimize (over all trees with appropriate weighted leaf set) the weight of the root. We present optimal algorithms for both alphabetic and non-alphabetic minimax tree problem.

There are several ways of generalizing the binary alphabetic minimax tree problem to t-ary trees. Kirkpatrick and Klawe [KirKla85] and Coppersmith, Klawe, and Pippenger [CopKlaPip86] presented linear algorithms for various versions of the t-ary alphabetic minimax tree problem. Our alphabetic minimax tree algorithm can be generalized to an O(log n)-time n /log n-processor algorithm for the t-ary alphabetic minimax tree defined as in [CopKlaPip86].

Minimax trees have several interesting applications [Gol76], [HooKlaPip84]. The complexity of the minimax tree problem has been studied in [Gol76], [KirKla85], [CopKlaPip86]. One can observe that in the sequential setting the techniques which can be used to construct trees with optimal weighted path length can be also used to construct minimax trees. In particular, Golumbic [Gol76] presented an O($n$log n)-time sequential algorithm for the non-alphabetic minimax tree problem which is an adaptation of Huffman's algorithm [Huf85] for constructing of trees with optimal weighted path length. Considering alphabetic versions of the two problems we observe that both binary alphabetic minimax trees and binary alphabetic trees with optimal (leaf) weighted paths

length can be constructed with the help of the T-C algorithm ([HuTuc71], [Hu73]) originally designed for the second of the problems. (The T-C algorithm requires $\theta(n \log n)$ time so it does not lead to an optimal algorithm for constructing minimax trees). The two problems seem to have a different nature when a parallel setting is considered. In this chapter we present an O(log n)-time n-processors algorithm for constructing non-alphabetic minimax trees. Interestingly, as we have mentioned in Chapter 4, there does not currently exist a parallel algorithm to solve the problem of constructing trees with optimal weighted path length whose total work is $O(n^{2-\varepsilon})$. The algorithm of Atallah, Kosaraju, Larmore, Miller and Teng runs $O(\log^2 n)$ in time using $n^2/\log n$ processors on a CREW PRAM. In Chapter 4 an O(log*$n$log n)-time n-processor CREW PRAM algorithm was presented which gives an approximate solution to this problem.

We also present an O(log n)-time n/log n-processor algorithm for constructing alphabetic minimax trees. In contrast the best currently known algorithm for constructing alphabetic trees with optimal weighted path length applies the parallel dynamic programming technique of Miller, Ramachandran, and Kaltofen [MilRamKar88] and requires $O(\log^2 n)$ time with roughly n^6 processors. This algorithm solves in fact a more general problem, in which the internal nodes may also have weights. Atallah, Kosaraju, Larmore, Miller, and Teng [AtaKosLarMilTen89] presented an $O(\log^2 n)$-time $n^2/\log^2 n$ processor algorithm which gives an approximate solution to this problem. Interestingly, our optimal parallel algorithm for constructing alphabetic minimax trees was motivated in part by the T-C algorithm for constructing alphabetic trees with optimal weighted path length.

The alphabetic minimax tree problem can be also viewed as a generalization of the following problem of *constructing the tree from a sequence of leaf depths*: Given a sequence of integers $l_1,...,l_n$ representing the depths of leaves of a binary tree construct the corresponding tree. One can prove that if in a minimax tree T the value $w(v_i)+l_T(v_i)$ (where

$l_T(v_i)$ denotes the depth of the leaf v_i in T) is constant then the tree is unique. Consequently, the above problem can be reduced to that of constructing the minimax tree for the weight sequence $-l_1, ...,-l_n$. An $O(\log^2 n)$-time $n/\log n$-processor CREW PRAM algorithm for the problem of constructing the tree from a sequence of leaf depths was presented by Atallah, Kosaraju, Larmore, Miller, and Teng [AtaKosLarMilTen89]. So, as a special case, our algorithm improves the last result.

In addition to the accelerated valley filling technique mentioned above we make use of several existing parallel algorithms and design techniques to achieve our results. We start our presentation by outlining the basic ideas behind the valley filling technique. Then we present, what we call a *level tree*, a data structure which allows us to speed up the basic valley filling technique. We present an optimal algorithm which constructs such a tree. Finally we give an optimal algorithm which given a level tree constructs an alphabetic minimax tree. In the last section of this chapter we show how our results for alphabetic minimax trees can be applied to non-alphabetic minimax tree problem and the minimax tree problem for a real weight function. We also discuss t-ary minimax trees.

5.1. Valley Filling Technique

Consider the bottom-up construction of a weighted tree. At each stage of such a construction we are dealing with a sequence of weights of roots of subtrees of the constructed tree called a *construction sequence* [1]. Our initial construction sequence is $w(v_0),w(v_1),...,w(v_n),w(v_{n+1})$ where $w(v_0)=w(v_{n+1})=\infty$.[2]

---

[1] The term "construction sequence" as well as "valley sequence" is taken from [Hu75].

[2] We thing of ∞ as an integer which is at least equal to the weight of the root of a minimax tree for the sequence $w(v_1),...,w(v_n)$.

A subsequence $u_k,u_{k+1},...,u_m$ of a construction sequence is called a *valley sequence* if it is a bitonic sequence, where a bitonic sequence is defined to be a sequence which is first nonincreasing and then nondecreasing. An element u_i is called a *maximal element of a valley sequence* $u_k,u_{k+1},...,u_m$ iff $u_k = u_{k+1} =....= u_i$ or $u_i = u_{i+1} =...= u_m$. The weight of the maximal elements of a valley is called the *level of the valley*. The observation that every construction sequence can be divided into valley sequences provides the basic idea behind the basic valley filling technique:

BASIC VALLEY FILLING ALGORITHM

while the construction sequence contains more than one element of weight less than infinity **do**:

(i) divide the current construction sequence into valley sequences;

(ii) for every valley build in parallel a forest of trees such that the leaves of each forest are the elements of the subsequence of the construction sequence forming the valley and the roots have weights equal to the level of the valley.

The process of replacing a valley sequence by a sequence of roots of trees whose weight is equal to the level of the valley is called *filling the valley* and each repetition of the while loop is called a *valley filling step* (compare Figure 5.1) Since at each valley filling step the number of valleys is reduced by at least half, the algorithm performs O(log m) valley filling steps in total, where m is the number of valleys in the initial construction sequence. One can implement each valley filling step in O(log *n*) time using *n*/log *n* processors (such an algorithm for the special case when the input sequence represents a leaf depth pattern was presented in [AtaKosLarMilTen89]). This leads to an O(log m log *n*)-time *n*/log *n* -processor algorithm.

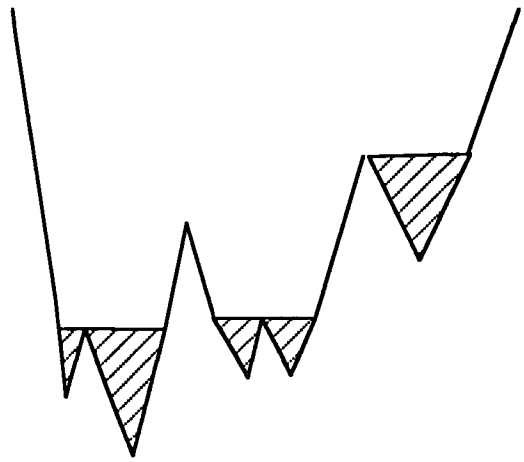


Figure 5.1. A valley filling step

The basic idea behind the *accelerated valley filling technique* is to speed up the basic algorithm by avoiding O(log m) distinct iterations. To this end we precede the construction of a minimax tree by a preprocessing step in which we construct what we refer to as the *level tree*. This tree contains, in particular, information about the number of internal nodes of the constructed tree which occur on certain distinguished levels. This information allows us to replace the iterative application of step (ii) by one pipelined parallel step.

5.2. Level tree and its construction

We start our description of the level tree with the following geometric construction (see Figure 5.2) : Represent the sequence of weights corresponding to the left-to-right order of the leaves of a constructed alphabetic minimax tree by a polygonal line : for every element v_i draw on the plane the point $(i,w(v_i))$, and for every i=1,..., n-1 connect the points $(i,w(v_i))$ and $(i+1,w(v_{i+1}))$. For every v_i such $v_i > v_{i+1}$ (resp., $v_i > v_{i-1}$) draw a horizontal line going from $(i,w(v_i))$ to its right (resp., left) until it hits the polygonal line. The intervals defined in such a way are called *level intervals*. We also consider the interval (∞,∞) and the degenerate intervals $((i,w(v_i),(i,w(v_i))$ as level intervals. Let e be a level interval. Note that at least one of e's endpoints is equal to $(i,w(v_i))$ for some index i. If

only one endpoint coincides with such a point then we call the leaf v_i the *defining leaf* for e. If both endpoints of e coincides with such points, then we choose the leaf corresponding to the left end of e as the *defining leaf* for e. Thus for every level interval e there exists a unique defining leaf which we denote by *def*(e). We define the *level of a level interval* to be equal to the weight of its defining leaf.

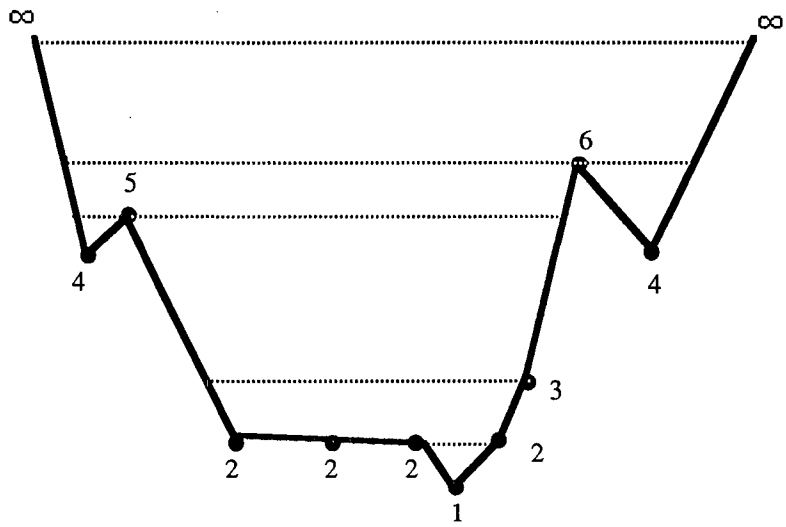


Figure 5.2. Level intervals

Note that a minimax tree can be embedded in the plane in such a way that the root of the tree belongs to the level interval (∞, ∞) and that internal vertices whose weights are equal to the weight of one of the leaves lie on the horizontal line going through this leaf. Furthermore, if there is a tree edge cutting a level interval, then adding a vertex subdividing this edge to the minimax tree does not increase the weight of the root. By this observation we can consider minimax trees which can be embedded on the plane such that no edge intersects a level interval other than at an endpoint (see Figure 5.3).

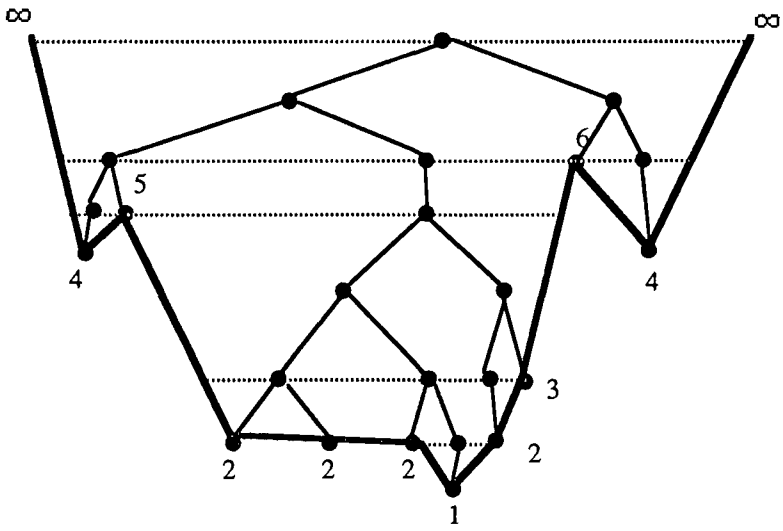


Figure 5.3. Embedded minimax tree

A level tree (see Figure 5.4) for a given sequence of weights is an ordered tree whose nodes are in one-to-one correspondence with the level intervals defined as above. For a node v define its parent to be the internal node which corresponds to the closest level interval which lies above the level interval corresponding to v. The left-to-right order of children of an internal node corresponds to the left-to-right order of the corresponding level intervals in the plane.

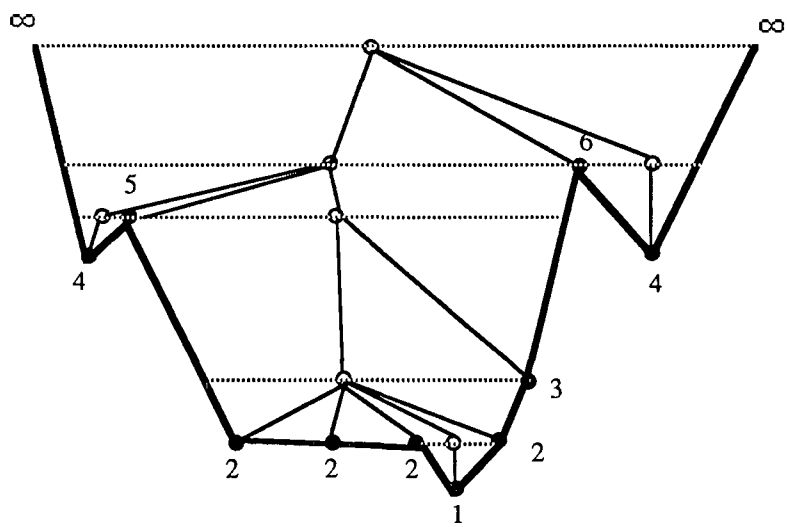


Figure 5.4. A level tree

We use the level tree to compute, for every level interval, of the number of nodes of the constructed minimax tree which belong to this level interval (assuming the embedding as described above). To this end, for every node u of a level tree we define *load*(u) to be equal to the number of nodes of the constructed minimax tree which belong to the level interval corresponding to u. We show how to compute this function using the tree contraction method.

5.2.1. Construction of level tree

To construct the level tree we have to establish the "parent" relation and the ordering of every internal node's children. It is easy to observe that to find the parent of a node corresponding to a level interval e, it suffices to find the strictly dominating successor and predecessor of *def*(e). Recall that the strictly dominating successor (resp., strictly dominating predecessor) of an element v_i in the sequence $w(v_0),w(v_i),...,w(v_{n+1})$ is the element v_j satisfying $w(v_i)<w(v_j)$ that is closest to the right (resp. to the left) of v_i. Observe that the parent of the node associated with a level interval e which is different than (∞,∞), is the node associated with a level interval defined by the dominating predecessor or successor of *def*(e) (whichever has smaller weight). Thus the "parent" relation can be established using the solution to the ASDN problem presented in section 2.6.

To construct lists of children of internal nodes, every node has to decide whether it is the first or the last node on a list of children and (if it is not the last node on a list) find its successor on the corresponding list. To solve this problem we use the solution to the ADN problem presented in the previous chapter. Observe that to construct lists of children it suffices to compare, for every node u, the weight of *def*(u) with the weight of the dominating successor of *def*(u) (see step 3 of the algorithm).

Let us present now a more detailed description of the algorithm to construct the level tree for a given input sequence of weights. Assume first that we have n processors and that the sequence of the weights of the leaves (in left-to-right order) is given in an n-element array W. We assign one processor to each element of the array W (and therefore to each leaf). We also assign the processor assigned to $def(e)$ to the level interval e. So each processor is assigned to at most three nodes of the constructed level tree.

STEP 1: Solve the ADN and ASDN problems for the sequence W.

We use the optimal algorithms for the ADN and ASDN problems described in the previous chapter. Let $ds(v_i)$, $dp(v_i)$, $sds(v_i)$, and $sdp(v_i)$ be the dominating successor, dominating predecessor, strictly dominating successor, and strictly dominating predecessor of the i^{th} element, respectively.

STEP 2: Establish the parent relation

- For every level interval e_i let $y_i = sdp(def(e_i))$ if $sdp(def(e_i)) \leq sds(def(e_i))$ and $y_i = sds(def(e_i))$ otherwise.

- Define the parent of the level tree node which corresponds to the level interval e_i to be the node which corresponds to the (nondegenerate) level interval defined by y_i which lies above e_i.

STEP 3: Define the ordering of the children

For every level interval check whether the tree node associated with it is the first (or the last) child of its parent (in the left-to-right order).

Observe that u_j is the first child of some internal node if one of the following holds:

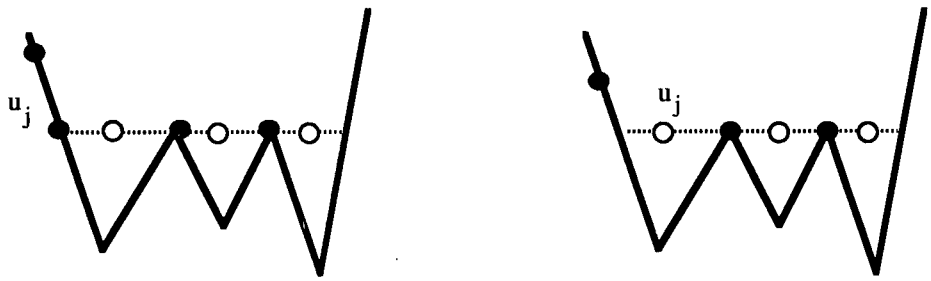
a) u_j corresponds to a leaf v_i (a degenerated level interval) such that $w(v_{i-1}) > w(v_i)$ (see Figure 5.5 a),

b) u_j corresponds to a nondegenerate level interval e such that $w(dp(def(e))) > w(def(e))$ (see Figure 5.5 b).

The last element of the sequence can be determined in a similar way.

For any element u_j which is not the last element of a list of children, the next element on the list is defined in the following way:

- if u_j is a leaf and it is not the last element of a list of children then u_j is the left end of a level interval and the internal node corresponding to this level interval follows u_j on the list;
- if u_j is an internal node and it is not the last element of a list of children then the right endpoint of the level interval to which u_j corresponds coincides with a leaf node. This leaf node follows u_j on the list of children.



a) b)

Figure 5.5. Introducing the ordering of children

By Theorem 2.14, step 2 of the algorithm can be implemented in O(log n) time with n/log n processors. The remaining steps can be performed in constant time with n processors. Thus they can be computed in O(log n) time with n/log n processors by a straightforward simulation of log n processors by one processor.

5.2.2. Computing the *load* function

Now, for every internal node u we want to compute the value *load*(u) equal to the number of nodes of the minimax tree which belong to the level interval corresponding to u.

Lemma 5.2 : The function *load*(u) can be computed for every internal node u in O(log *n*) time with *n*/log *n* processors.

Proof : If u is a leaf then *load*(u) = 1. Assume that u is an internal node. Let $u_1,...,u_k$ be the children of u. Let Δu denote the minimum of the value $\lceil \log n \rceil$ and the difference between the level of the level interval corresponding to node u and the level of the intervals corresponding to its children. It is easy to confirm that

$$load(u) = \left\lceil \frac{load(u_1)+...+load(u_k)}{2^{\Delta u}} \right\rceil .$$

We compute the function *load*(u) for every internal node with the help of an algorithm which is an interpretation of a 𝔅−𝔔𝔗𝔖 scheme. In order to apply the scheme we have to replace the level tree by a binary tree. Let T' be a tree obtained from the level tree by replacing each internal node, u, by a sequence of nodes $u',u'_1,...,u'_{k-1}$ as on Figure 5.6 ($u_1,...,u_k$ are children of u). We assign $\Delta u'_i=0$. Then it is easy to see that *load*(u)=*load*(u').

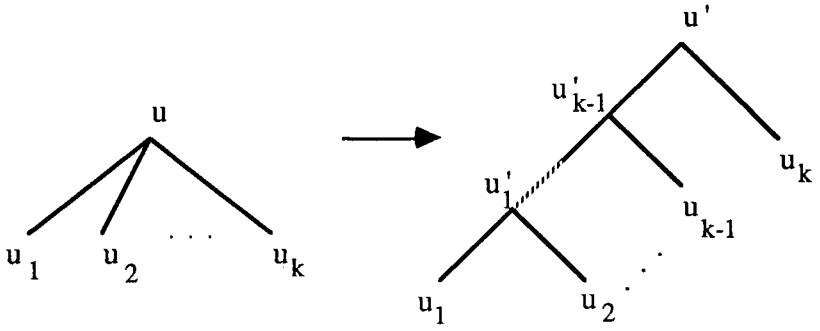


Figure 5.6. Binarization of a level tree

We associate the function $f_{2^{\Delta u}}(x,y) = \left\lceil \frac{x+y}{2^{\Delta u}} \right\rceil$ with every internal node u of the binary tree. To do this we build a table of size $\lceil \log n \rceil$ which keeps the value 2^j for every j = 1, ..., $\lceil \log n \rceil$. Such a table can be build in O(log *n*) time sequentially. Then, for every internal node, the index of the function associated with it can be computed in constant time. In this

way we have obtained the bottom-up tree computation problem considered in Example 2.7. As was shown in section 2.5.2 this can be solved by an interpretation of the $\mathfrak{B}-\mathfrak{Q}\mathfrak{T}\mathfrak{C}$ scheme in O(log n) time with n/log n processors ■

5.3. An optimal algorithm for alphabetic minimax tree problem

Consider a node u of a level tree. With each such a node we have *load*(u) associated nodes of the minimax tree which we want to construct. Let us call those *load*(u) nodes the *group of nodes associated with u.* If u is an internal node of the level tree then the sequence of groups of nodes associated with children of u (in left-to-right order) is called the *block of nodes associated with u.* Let g(u) be the number of the elements in the block associated with u.

A *balanced forest with n nodes and m* roots is a forest of m trees and total number of leaves equal to n which has the minimal depth (where by the depth of a forest we mean the maximum of the depths of the trees in the forest).

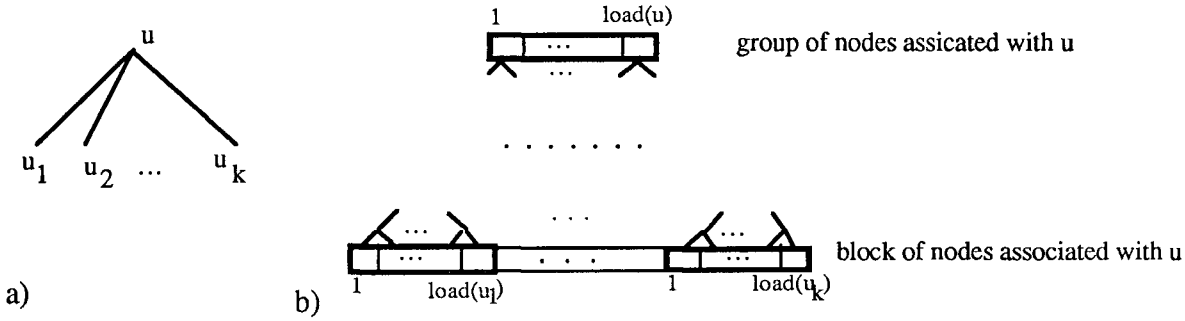


Figure 5.7. Construction of a balanced forest

a) node u in a level tree

b) corresponding part of the constructed minimax tree

In order to construct the minimax tree for every internal node u we have to construct the group of *load*(u) nodes, determine the block of g(n) nodes associated with u, and build a balanced forest with g(u) leaves and *load*(u) roots (compare Figure 5.7). To minimize the number of internal nodes of the constructed minimax tree we remove internal nodes of degree one. This can be summarized in the following algorithm:

STEP 1: Construct a table NODES such that each element of the table corresponds to one node of the constructed tree. Furthermore nodes from the same block have associated consecutive positions in the table.

For any node v from the level tree let *group_index(v)* denote the index in the table NODES of the first element of the group associated with v. Similarly define the *block_index(v)* as the index of the first element of the block associated with v.

1.1. Recall that every internal node of the level tree keeps the lists of its children. Concatenate these lists in any order (say using the Euler Tour Technique [TarVis84]). Let L be the resulting list. The order of the elements in the list defines the order of storing blocks in the table NODES.

1.2. For every node v in L compute its *group_index*(v) by summing (using a prefix sum computation) the values *load*(u) for all nodes u preceding v on the list L. The sum over all elements u in L of values *load*(u) gives the size of the table NODES.

1.3. For every node v compute its *block_index* (v) that is the group_index of the first child of v.

STEP 2: For every element of the table NODES compute the index of its parent and for every node corresponding to a leaf of the minimax tree construct the pointer to the element of the input sequence labelling the given node.

2.1. Divide the table NODES into $n/\log n$ segments of length $\log n$ each, and associate one processor to each segment. Note that each segment can contain a number of blocks. Within every segment use a sequential linear algorithm to build a balanced forest for

each block of size at most log n which is entirely contained in the given segment or intersects the right boundary of the segment.

2.2. All remaining blocks have length $m_i > \log n$ and as a consequence of step 2.1. each block has assigned $\lfloor m_i/\log n \rfloor$ processors. We can decide which processor is assigned to which block as follows: Every processor decides by a sequential search whether it is assigned to an interval containing the beginning of a block. If so it writes one in its local memory. Then a prefix sum computation gives, for every , the number of the block it is assigned to. With this processor assignment we can finish the computation in O(log n) time using standard techniques (i.e. for each block use a sequential linear time algorithm to construct at most m subtrees of $\lceil \log n \rceil$ leaves and a parallel algorithm to construct the remaining nodes of the tree).

STEP 3: Remove the internal nodes with only one child using an interpretation of $\mathfrak{B}$–$\mathfrak{A}\mathfrak{T}\mathfrak{C}$ scheme.

Identify maximal chains of internal nodes with only one child. We consider the lowest level vertex in a chain as the beginning of the chain. Given n processors it is easy to determine in constant time for every internal u whether u belongs to such a chain and if so who is its predecessor and successor (if any). Since one can simulate log n processors by one in using O(log n) time this can be implemented in O(log n) time with n/log n processors. To remove internal nodes of degree one we have to compute for the first element on every chain the corresponding last element. This can be done using the following interpretation of the (full) $\mathfrak{B}$–$\mathfrak{A}\mathfrak{T}\mathfrak{C}$ scheme. We assume that the only child of an internal node of degree one is the left child.We add a right child to each such node. Let $\mathfrak{G}=\{id \mid id(x)=x\} \cup \{label_i \mid label_i(x)=i\}$, $\mathfrak{F}=\{id' \mid id'(x,y)=x\} \cup \{zero \mid zero(x,y)=0\}$. It is easy to confirm that $\mathfrak{G}$ and $\mathfrak{F}$ satisfy axioms (i)-(ii) of $\mathfrak{B}$–$\mathfrak{A}\mathfrak{T}\mathfrak{C}$ scheme with P=1 and T=O(1). We associate with every edge (v,*parent*(v)) such that v has degree two in the original tree or is a leaf the function $label_v$ and the function id with all other edges. With every internal node of degree 2 in the original tree we associate the function zero and with internal nodes whose degree in the original tree is one we associate the function id'. The result of this bottom-up Algebraic Tree Computation is that for every node which has degree two in the original tree, the values associated with the edges leading to its children define the addresses of its new children.

Since all steps of the algorithm can be implemented in O(log n) time with n/log n processors, this yields the following theorem:

Theorem 5.3 : The binary alphabetic minimax tree problem can be solved in O(log n) time with n/log n processors.

5.4. Other versions of the minimax tree problem

5.4.1. t-ary alphabetic minimax trees

There are several ways of generalizing the binary alphabetic minimax tree problem to t-ary trees. Kirkpatrick and Klawe [KirKla85] considered regular t-ary alphabetic minimax trees (assuming that the number of leaves is equal to 1 mod (t-1). Coppersmith, Klawe, and Pippenger [CopKlaPip86] considered alphabetic minimax trees of degree (at most) t. As is shown in these papers, both types of trees admit linear time sequential construction algorithms. It is easy to confirm (see Figure 5.8) that the best regular t-ary alphabetic minimax tree may have a higher cost than some at most t-ary alphabetic minimax tree for the same input.

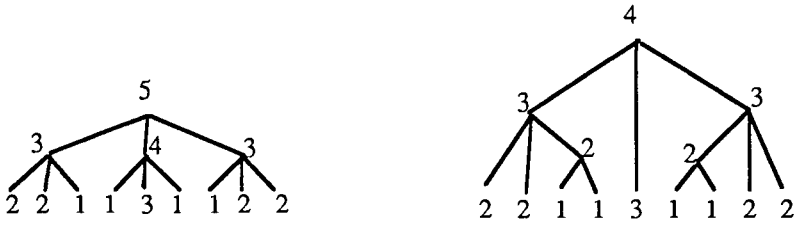


Figure 5.8. A regular t-ary alphabetic minimax tree and a t-ary alphabetic minimax tree

Coppersmith, Klawe, and Pippenger [CopKlaPip86] also addressed the question of minimizing the number of internal nodes in an alphabetic t-ary tree. The number of internal

nodes of a t-ary tree is bounded from below by $\lceil (n\text{-}1)(t\text{-}1) \rceil$. Coppersmith, Klawe, and Pippenger [CopKlaPip86] gave a linear time algorithm which converts an alphabetic t-ary minimax tree into an alphabetic t-ary minimax tree with the number of internal nodes bounded by $\lfloor (n\text{-}1)(t\text{-}1)+(t\text{-}2)(n+1)/3(t\text{-}1) \rfloor$. However the tree obtained by this compaction process is still not guaranteed to minimize the number of internal nodes.

It is easy to convert our alphabetic binary minimax tree algorithm to an algorithm which constructs an alphabetic t-ary tree. It suffices to replace the value $2^{\Delta u}$ in the proof of Lemma 5.2 (the recurrence for the *load* function) by $t^{\Delta u}$ and the construction of a binary balanced forest in step 1 of the tree construction algorithm by a balanced t-ary forest. Here we sketch a parallel minimax tree compaction algorithm which converts a minimax tree into tree of the same cost with a possibly smaller number of internal nodes. The resulting tree satisfies the same properties as the tree which results from the compaction algorithm of Coppersmith, Klawe, and Pippenger [CopKlaPip86]. As in [CopKlaPip86] we define a *leaflet* to be an internal node that has only leaves as children and has degree less than t. The tree produced by the compaction algorithm satisfies the following conditions:

(1) Each internal node either has degree t or is a leaflet

(2) No two adjacent leaves are the children of different leaflets.

(3) Each leaflet has degree at least two.

The upper bound on the number internal nodes of a t-ary tree satisfying conditions (1)-(3) is given in the following lemma:

Lemma 5.6 [CopKlaPip86]: If T is a tree with n leaves satisfying conditions (1), (2), and (3), then t has at most $\lfloor (n\text{-}1)(t\text{-}1)+(t\text{-}2)(n+1)/3(t\text{-}1) \rfloor$ internal vertices.

Similar to the sequential algorithm of Coppersmith et.al., our parallel algorithm compacts the tree in two steps. We first present a high level description of these two steps and then present them in more detail.

During the first step we compact internal nodes which are not leaflets. The sequential algorithm of Coppersmith et. al. compacts such nodes by traversing them one after another in preorder. To allow for parallel computation, we cover the tree by disjoint paths and compact the nodes on each path independently. Let $p=u_1,...u_r$ be a path of internal nodes in the tree T (u_1 is the highest level vertex on the path). To compact p means to replace it by a path p' such that the following conditions are satisfied:

(p1) the children of nodes of elements on a path p becomes children of nodes on path p';

(p2) all but the last element of p' have t-1 children which are not on p' and the last element of p' has at most t and at least 2 children;

(p3) the cost of the tree T' obtained from the tree T by replacing by the path p' the path p is at most equal to the cost of T;

(p4) T and T' have the same order of leaves.

In our algorithm we compact a path by moving the children of vertices on the path from the end of path towards the beginning. To ensure condition (p4) we use the following algorithm (compare Figure 5.9).

For each vertex v on the path p we define its *sequential number*, as its distance from the end of the sequence. With each child of v which does not belong to the path p, we associate its *path index* defined as follows: if a child is a left sibling of a node on the path p its path index is equal to the negative of the sequential number of its parent otherwise it is equal to the sequential number of its parent.

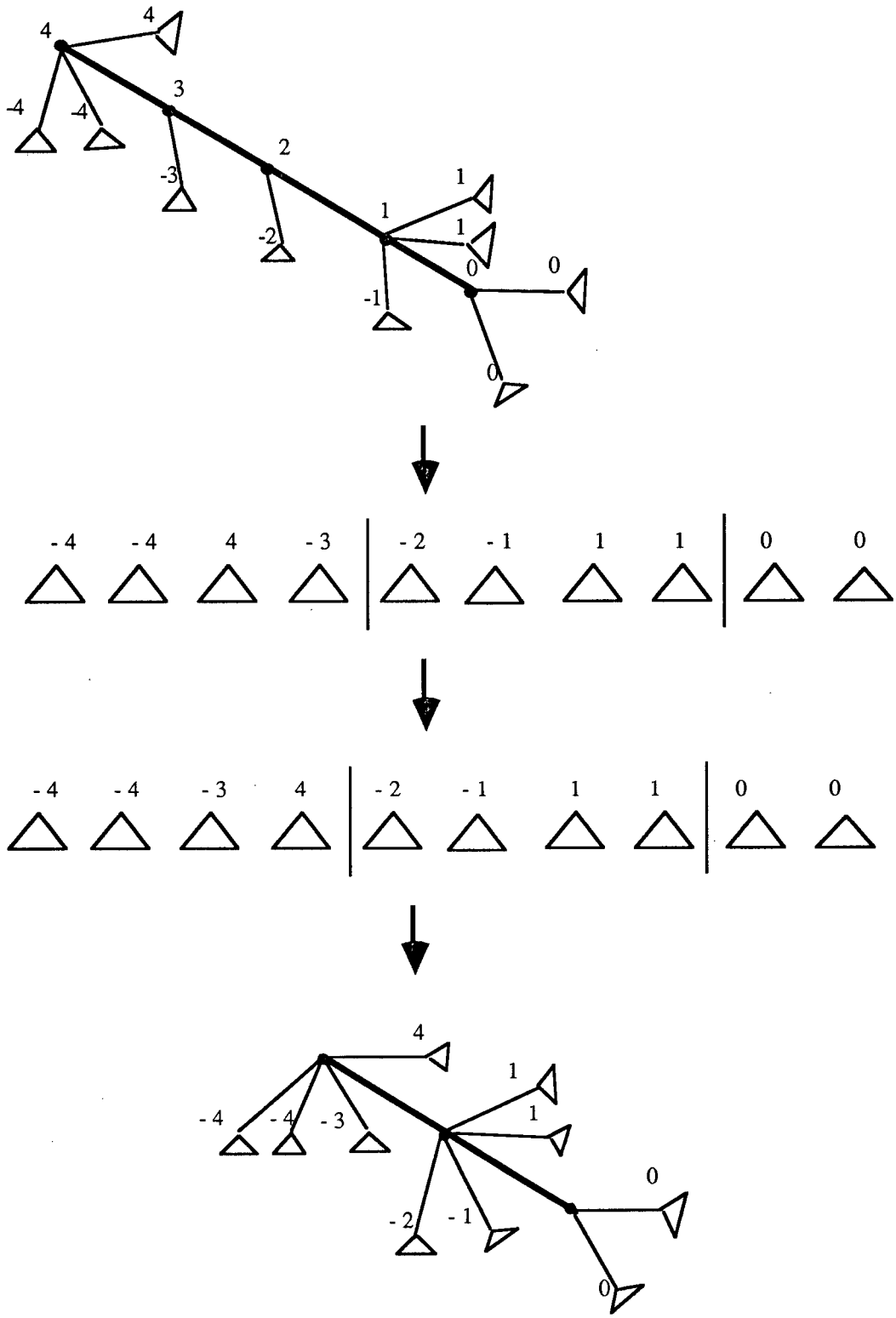


Figure 5.9. Path compaction (t = 5)

THE PATH COMPACTION ALGORITHM

STEP 1: Compute sequential numbers.
For each node v on the input path p and for each child of v which does not belong to p compute (using a prefix sum computation) its path index.

STEP 2: Concatenate.
Concatenate sequences of children of nodes from the path p, excluding nodes which belong to p. We divide the resulting sequence into blocks of size t-1 except for the last block which may be of size at most t.

STEP 3: Sort.
Within each block we sort elements according to their path index.

STEP 4: Construct the resulting path p'.
Cnstruct path the p' by creating one internal node per block.The children of a node from p' are defined as follows: If a node v corresponds to the last block then its children are the nodes in this block. Otherwise, in addition to the nodes in the corresponding block, v is given the internal node corresponding to the next block as its child. The order of children agrees with their order in the corresponding block, and the internal node corresponding to the next block is inserted between the children of negative and positive path index.

Assume that a tree T has been partitioned into vertex disjoint paths. Then the tree resulting from T by the compression of all paths satisfies conditions (1) and (3) at all internal nodes except at the nodes resulting from compacting a path to a single node. To ensure conditions (1) and (3) for all internal nodes we iterate the compaction step for different partitions of the tree into paths.

During the second step of the algorithm we compact leaflets. We construct monotone lists of leaflets and within each such list we move (if possible) all but one children from leaflets of smaller weights to leaflets of greater or equal weight. If a leaflet is left with one child this child replaces the given leaflet in the sequence of children of the parent of the leaflet. To do this we iterate a compaction step two times. During the first iteration we consider maximal nondecreasing sequences of neighboring leaflets, and compact the

leaflets on these sequences (see Figure 5.10). During the second iteration we consider maximal nonincreasing sequences of neighboring leaflets and compact the leaflets on these sequences.

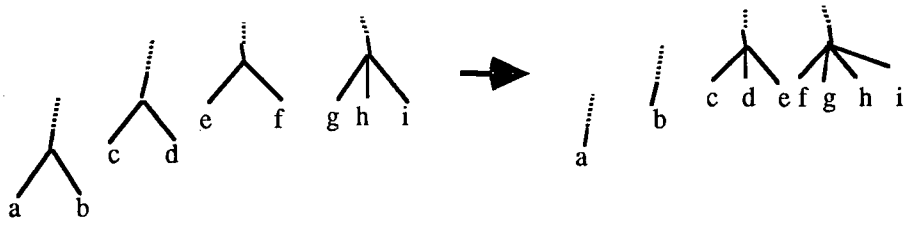


Figure 5.10: Leaflets compaction (t = 4)

Thus the tree compaction algorithm can be described as follows:

THE TREE COMPACTION ALGORITHM:

STEP 1 : compaction of internal nodes

Repeat the following compacting step t times:

1. Partition the tree into independent paths by choosing, for any node v which has a nonleaf child, its first nonleaf child as its path successor.
2. Compact in parallel all the paths computed in the previous step.

STEP 2 : compaction of leaf nodes

1. Construct lists of leaflets as follows: For each leaflet v, check whether it is adjacent to a leaflet with greater or equal weight. If so, this leaflet follows v on the list. Let l^1_i be the number of leaflets on the i^{th} list and l^2_i be the sum of their children.
2. Let $k_i = \lceil (l^2_i - l^1_i) / (t-1) \rceil$. For each list divide the last $l^2_i - l^1_i + k_i$ elements of the sequence of children of the leaflets of the list into blocks of size t (except possibly for the first block). These blocks of leaves became the new children of the last k_i leaflets. The remaining elements replace the remaining leaflets (we have one element per leaflet).
3. Repeat steps 1 and 2 replacing nondecreasing lists by nonincreasing lists and switching the meanings of adjectives "last" and "first".

Lemma 5.5 : The algorithm described above compacts the input alphabetic minimax tree to a minimax tree satisfying conditions (1)-(3).

Proof: It is obvious that after Step 1, the condition (3) is satisfied. An easy inductive argument shows that after the i-th iteration of the path compaction step each node which has less than t children and is not a leaflet has at least i children, and the first i of them (in the left-to-right order) are leaves. To see that (1) and (3) are satisfied after step 2 of the algorithm first note that step 2 never changes the number of children of a node which is not a leaflet. Step 2 consists of two compacting steps each of which consists of shifting children of leaflets of smaller weight to leaflets of greater weight. There is at most one leaflet left on each list of adjacent leaflets. Thus condition (2) is also satisfied. ■

5.4.2. Non-alphabetic t-ary minimax trees

The main result of this section is based on the following lemma:

Lemma 5.6: The cost of a non-alphabetic minimax t-ary tree is equal to the cost of the alphabetic minimax tree whose leaves form a sorted sequence.

Proof: We use the following result proven by Golumbic [Gol76] : If T is a minimax tree then

$$c(T) = \lceil \log_t \sum_{v \in V} t^{w(v)} \rceil. \qquad (5.7)$$

It follows immediately that replacing t elements of weight w by one element of weight w+1 does not change the cost of the minimax tree. Also, if the number of minimal elements is smaller than t, then replacing them by an element of the weight greater by one does not change the cost of the minimax tree. The second fact follows from the observation that if a subset of the set of elements of minimal weight share a common parent with an element of

greater weight, then we can insert an internal node which becomes a new parent of the minimal nodes and a child of their old parent. This justifies the following algorithm for the construction of a (non-alphabetic) minimax tree:

1. Sort the leaves in nonincreasing order of their weights. Let W be the resulting construction sequence.

2. While there are more than t elements in the construction sequence do:
   Let w be the weight of the last element of the sequence. If t or more elements have weight w replace the first t elements of weight w by a common parent. Otherwise replace all elements of weight w by a common parent.

3. If more than one element remains, construct a common parent for all remaining elements

It is easy to observe that the current construction sequence remains sorted as an invariant of the above algorithm. Thus the algorithm constructs an alphabetic tree (for the sorted sequence of leaf weights) whose cost is equal to the cost of a non-alphabetic minimax tree. ■

By this result and the results of the previous section we can construct a non-alphabetic (integer) minimax tree in O(log n) time with n CREW processors using the parallel sorting algorithm of Cole [Col86]. It is also worth noting that in the case of an alphabetic minimax tree with leaf weights in sorted (say nonincreasing) order, one can give a parallel algorithm which minimizes the number of internal nodes. We precede the sketch of such an algorithm with a number of definitions and observations.

A maximal subsequence of elements $u_i,u_{i+1},\ldots,u_{i+k}$ of equal weights such that $w(u_{i-1}) > w(u_i)$ is called a *tread*. A tread is called a *short tread* iff the number of elements in the tread is less than t. Otherwise it is called a *long tread*. A monotone sequence, all of whose treads have length at most t, is called a *slope*. A monotone sequence without long treads is called a *cliff*.

A maximal sequence $u_{i+1},\ldots,u_{i+t},u_{i+t+1},\ldots,u_{i+rt},\ldots,u_{i+(r-1)t+1},\ldots,u_{i+rt},u_{i+rt+1}$, such that

$w(u_{i+1})=\ldots=w(u_{i+t-1})$,

$w(u_{i+(t-1)+1})=w(u_{i+(t-1)+2})=\ldots=w(u_{i+2(t-1)})=w(u_i)-1$,

.....

$w(u_{i+(r-1)(t-1)+1})=w(u_{i+(r-1)t+2})=\ldots=w(u_{i+r(t-1)})=w(u_{i+r(t-1)+1})=w(u_{i+r(t-1)})-1$,

and $w(u_i) < w(u_{i-1})$ is called a *stair*.

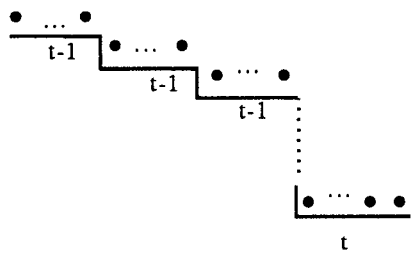


Figure 5.11. A stair

Lemma 5.8 (tread cutting lemma): Let U be a construction sequence containing a long tread $u_i,u_{i+1},\ldots,u_{i+k}$ such that $w(u_{i-1}) > w(u_i)$, and let $k+1 = st+q$ $(0\le q<t)$. Let U' be a construction sequence obtained from U by replacing each of the subsequences $u_{i+pt},u_{i+pt+1},\ldots,u_{i+pt+t-1}$, $p=0,\ldots,s-1$ by a common father u_p' of weight $w(u_i)+1$. Then $C(U) = C(U')$.

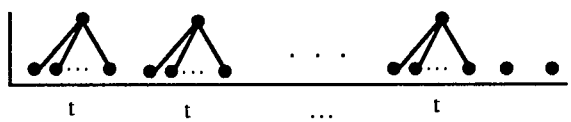


Figure 5.12. Treads cutting

Proof: Immediately from equality (5.7) ■

Lemma 5.8 (stair climbing lemma): Let U be a sequence containing the stair $w(u_{i+1}),\ldots,w(u_{i+t}),w(u_{i+t+1}),\ldots,w(u_{i+rt}),\ldots,w(u_{i+(r-1)t+1})\ldots,w(u_{i+rt}),w(u_{i+r(t-1)+1})$ and

let U' be a sequence obtained from U by replacing the stairs by the tree T presented in Figure 5.13. Then C(U)=C(U').

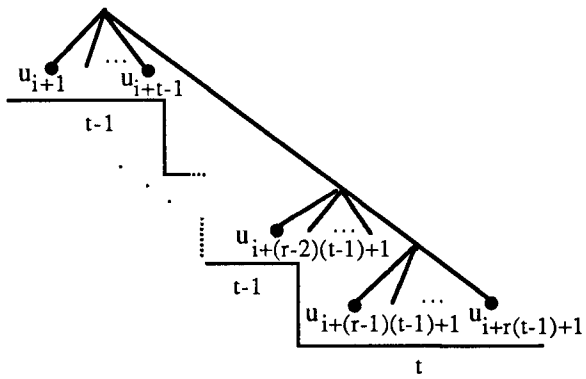


Figure 5.13. Stair climbing

Proof: This follows from r applications of the tread cutting lemma. ■

Lemma 5.9. (tread climbing lemma) Let k be the length of the longest long tread in the construction sequence. Then after $\lceil \log_t k \rceil$ applications of the tread cutting lemma the input sequence is reduced to a slope.

Proof: Let m_i be the length of a maximal tread after i applications of the tread cutting lemma. After the $i+1^{st}$ application of the tread cutting lemma we have $m_i \leq \lfloor \frac{m_{i-1}}{t} \rfloor + t-1$.

This follows from the fact that at most t-1 elements from a tread do not obtain a parent, and there are at most $\lfloor \frac{m_{i-1}}{t} \rfloor$ new nodes coming from the tread below to the tread above it. Thus after $\lceil \log_t k \rceil$ applications of the tread cutting lemma $m_{\lceil \log k \rceil} \leq t$. ■

Lemma 5.10 (slope climbing lemma). Let U be a sequence forming a slope. Then the application of the stair climbing lemma to all stairs of the slope reduces the slope to a cliff.

Proof : By the maximality condition in the definition of stairs, the root of a tree which replaces a stair cannot be a part of a long tread. Since all long treads are removed as the

result of stair climbing, and no new long treads are created, the resulting sequence forms a cliff. ■

Lemma 5.11. (cliff climbing lemma): Let $U = w(u_1)...,w(u_{r(t-1)+j})$, where $1 < j \leq t$, be a cliff. Then C(U) is equal to the cost of the tree created as in Figure 5.14.

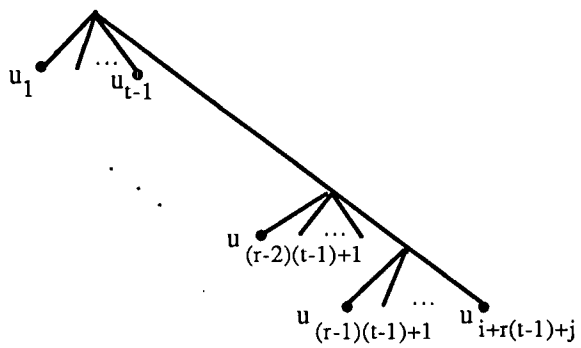


Figure 5.14. Cliff climbing

Proof: This follows from the immediate observation that the cost of this tree is equal to $w(u_1)+1$, and that this is a minimax tree for the sequence U. ■

The following algorithm for constructing a non-alphabetic minimax t-ary tree which minimizes the number of internal nodes is an immediate consequence of the above lemmas and Lemma 5.6:

1. Sort the input sequence of weights applying Cole's sorting algorithm;
2. Apply the treads climbing lemma to reduce the input sequence to a slope;
3. Apply the slope climbing lemma to reduce the slope obtained in step 2 to a cliff;
4. Apply the cliff climbing lemma to the cliff obtained in step 3.

It is straightforward to implement steps 2-4 of the algorithm in $O(\log n)$ time with $n / \log n$ processors (use Brent's pronciple to reduce the number of processors of an obvious implementation of step 2). Since all but (possibly) one internal nodes of the tree have t children the tree constructed by the above algorithm minimizes the number of internal nodes.

5.4.3. Minimax trees with real weights

Kirkpatrick and Klawe [KirKla85] showed that the alphabetic minimax tree problem for real leaf weights can be reduced to O(log n) instances of the integer version. Let $W=w_1,...,w_n$ and $a_i=w_i-\lfloor w_i \rfloor$. Define $W(a_i)$ to be the following (integer) sequence of weights $\lceil w_1 - a_i \rceil, \lceil w_2 - a_i \rceil, ..., \lceil w_n - a_i \rceil$. Let $b_1, ..., b_n$ be the number a_i rearranged into ascending order and let $b_0 = b_n$. The reduction is based on the following lemma [KirKla85]:

Lemma 5.12 [KirKla85]: Let j be such that $C(W(b_j))+b_j = \min\{ C(W(b_i))+b_i \mid 1 \le i \le n\}$. Then :

(a) $C(W) = C(W(b_j))+b_j$, and a minimax tree for $C(W(b_j))$ yields an minimax tree for $C(W)$;

(b) if $i \le j$ then $C(W(b_i)) \ge C(W(b_j))$;

(c) $C(W(b_0)) - C(W(b_n)) = 1$.

The proof technique used in [KirKla85] also applies to all the types of minimax trees considered in this chapter. This immediately leads to O($\log^2 n$)-time, n/log n-processor (or an O(log n)-time n^2/log n-processor) parallel algorithms for the alphabetic and non-alphabetic minimax tree problems.

5.5. Summary

In this chapter, we identified a parallel tree synthesis technique called the valley filling technique. The technique is applicable to the construction of weighted trees where weights are derived from a well ordered domain. The technique basically consists of the iterative application of a valley filling step in which the sequence of weights is divided into so-

called *valley sequences,* and then the parts of the tree corresponding to each valley sequence are computed independently. As the output of a valley filling step we obtain a sequence of rooted trees. The corresponding sequence of weights of the roots of these trees forms the input for the next iteration of the valley filling step. The idea of performing independent computation within a valley sequence probably originates with the T-C algorithm (the Hu and Tucker algorithm) for construction of alphabetic trees with optimal weighted path length ([HuTuc71], [Hu75]) Later Atallah et. al. [AtaKosLarMilTen89] used an independent computation within valley sequences (referred to there as "fingers") for their $O(\log^2 n)$-time $n/\log n$ -processor algorithm to construct the tree from a given sequence of depths of leaves.

The basic idea behind the accelerated valley filling technique is to speed up the basic algorithm by replacing the iterative application of the valley filling step by one pipelined parallel step. Towards this end, our algorithm contains a preprocessing step in which it precomputes the sequence of weights of the roots of the subtrees which will be used to fill any particular valley. This is done based on the result of a bottom-up Algebraic Tree Computation before corresponding subtrees are constructed. Then it remains to construct these subtrees and glue them together into the resulting tree.

The accelerated valley filling technique is applied to obtain an $O(\log n)$-time $n/\log n$ -processor algorithm to construct the (integer) alphabetic minimax tree. Since the problem of constructing the tree from a given a sequence of leaf depths can be formulated as a minimax tree problem, this improves, in particular, the result of Attalah et. al. mentioned above. We show that our optimal algorithm to construct alphabetic minimax tree can be used to obtain an $O(\log n)$-time n-processor algorithm for non-alphabetic minimax trees. In fact, if the input sequence is sorted then only $O(\log n)$ time and $n/\log n$ processors is required. (Observe that the problem is as least as difficult as integer sorting.) We present a $O(\log n)$ time n-processor algorithm to construct a t-ary non-alphabetic minimax tree

minimizing the number of internal nodes. Finally, we show an O(log n) time n /log n processor algorithm which reduces the number of internal nodes of an alphabetic t-ary minimax tree. This algorithm parallelizes the incremental sequential algorithm of Coppersmith, Klawe, and Pippenger [CopKalPip86] using a path compaction technique.

CHAPTER 6 : CONCLUDING REMARKS

In this thesis, a number of parallel techniques are presented for efficient construction of certain families of trees (*cf.* chapters 3-5). Along with these tree construction techniques, a number of parallel techniques are developed which have broader applications (*cf.* chapter 2). These techniques are illustrated with new parallel algorithms to construct various types of trees, with concrete results summarized at the end of each chapter. One objective of this final chapter is to reconsider the problems discussed in the thesis from a different perspective.

For a large part of the thesis, we have been dealing with the problem of the parallel construction of trees which are optimal with respect to some cost function. Many such optimization problems can be solved using the parallel dynamic programming technique developed by Miller, Ramachandran, and Kaltofen [MilRamKal86]. The basic limitation of this technique is its high processor cost. The technique (when applied to construction of optimal weighted trees) generally involves iterative multiplication of n^2 x n^2 matrices and thus requires $M(n^2)$ (by current knowledge roughly n^6) processors. However parallel dynamic programming technique remains a basic tool for constructing optimal trees, even for problems which can be solved optimally by sequential algorithms which do not use dynamic programming. Sometimes one can exploit the special structure of matrices associated with a given problem and in this way reduce the processor cost ([AtaKosLarMilTen89]). In our approach we have been looking for solutions which completely avoid dynamic programming.

A technique which can accomplish this is to relax an optimization problem and search for almost optimal solutions. This allows us to confine our attention to a restricted family of trees (in our case to trees of polylogarithmic depth) and design algorithms which relay on the structure of the trees in the family. This approach has been taken in Chapter 4 where we consider almost optimal solutions to the problem of constructing trees with minimal weighted path length. We present a whole family of parallel algorithms naturally related to the Huffman algorithm.

In Chapter 5, we present optimal parallel algorithms to construct minimax trees which also avoids dynamic programing, yet constructs an exact solution. As we previously mentioned, our approach has been motivated in part by the T-C algorithm - a sequential $O(n \log n)$ time algorithm to construct alphabetic trees with optimal weighted path length. We identify the valley filling technique and developed its accelerated version - the accelerated valley filling technique. Using this accelerated valley filling technique we obtain an optimal parallel algorithm for constructing minimax trees.

We next consider techniques for accelerating parallel algorithms whose natural implementation involves the iteration of some basic step which depends on the result of the previous iteration. This is one of the fundamental problems in parallel computation. We can think of the consecutive iterations as of consecutive levels of computation. We identify a few approaches:

1. Cascading sampling.

   This technique is introduced in Chapter 2 (section 2.6) and considered further in Chapter 6. The basic idea is to precede the iteration by a preprocessing step. During this preprocessing step, for each level of computation, a computation is performed on a sample subset of the data available at the given level and the results are distributed to the other levels of computation. The additional information which arrives at a given level during this preprocessing step is used to speed up the iterated step.

2. Pipelining

   This technique makes it possible to start computation on a higher level before the computation on a lower level is finished. This approach can be applied when it is possible to start computation on a higher level based on a partial results obtained from a lower level of computation. This approach has been taken in Cole's optimal parallel merging sort algorithm. A pipelining technique is usually combined with a sampling algorithm, that is with an algorithm which chooses the portion of information to be submitted to the next higher level prior to the completion of the computation at a given level ([Col85],[Kos89]).

3. Collapsing an iterative computation into one step

   This is a specific mixture of the two previous approaches. Sometimes it may be possible to obtain in a preprocessing step all information required for all levels. Then the computation on all levels can be completed by performing one parallel step. Such approach is taken in Chapter 5 to speed up the general valley filling technique to produce the accelerated valley filling technique.

We now consider the tree contraction technique. Following the work of Miller and Reif [MilRei85], applications and simplifications of this technique have been studied in a broad range of papers including some of the results reported in this thesis. One of the first steps was to abstract the technique from its applications. Another step was to systematize classes of problems which can be solved using this technique. One step in this direction are Bottom-up and Top-down Algebraic Tree Computation Schemas defined in Chapter 2.5. This formalism however does not capture all possible applications of the technique. The idea of tree contraction occurs in implicit or explicit ways in parallel solutions to many problems which do not seem to have a natural definition as Algebraic Tree Computations. The cotree construction algorithm presented in Chapter 3 is an example of such a problem. The process of cotree construction can be viewed as the reverse of a cotree contraction

process. As well, the general valley filling technique (Chapter 5) has a hidden application of the tree contraction technique. Namely, if we consider the level tree corresponding to a given input sequence then we can observe that consecutive iterations of the valley filling step can be visualized as iterative contraction steps performed on the level tree. This observation provides more intuition as to why the explicit application of the tree contraction technique makes it possible to obtain the accelerated version of the general valley filling technique.

Both tree contraction and list ranking techniques were originally developed to support parallel dynamic evaluation of an expression provided in the form of a tree or a list. In the first case, we usually deal with a parse tree of an expression. In the second case, we have one associative operation which is applied to all elements of the list. We assume that all arguments are available at the beginning of the computation. Both the tree contraction and the list ranking, can be viewed as techniques for efficient scheduling operation. It is natural to ask which scheduling technique should be applied if the arguments in the leaves of a computation tree or a list arrive with certain (known) delays. A partial answer to this question is summarized in Figure 6.1. In this table, basic two questions are left open providing an interesting opportunities for future research.

We have presented a number of parallel tree construction algorithms. Some of these algorithms are optimal and some are open for improvement. A challenging open question which is whether there exists an efficient parallel algorithm to construct an optimal binary tree such that the processor time product is $O(n \log n)$ (or even $O(n^{2-\varepsilon})$ as stated in [AtaKosLarMilTen89]).

Description of an expression	no delays	delays
One associative and commutative operation on a sequence of data presented in the form of a list	list ranking	non-alphabetic minimax tree
One associative operation on a sequence of data presented in the form of a list	list ranking	alphabetic minimax tree
One associative and decomposable operation on a sequecne of data presented in a list	list ranking	?
A decomposable algebraic tree expression	tree contraction	?

Figure 6.1. Generalizations of the parallel dynamic expression evaluation problem

As mentioned before, the parallel valley filling technique discussed in Chapter 5 is related to the sequential algorithm of Hu and Tucker (the T-C algorithm) for the construction of an alphabetic tree with almost optimal weighted path length. It would be interesting to see whether this technique (in its basic or accelerated version) can be used to obtain a parallel algorithm for this problem (in its exact or approximate version) and other families of trees which can be built sequentially using the T-C algorithm [HuKleTan79]. As we have mentioned before, the best currently known algorithm for constructing alphabetic trees with optimal weighted path length uses the parallel dynamic programming technique and requires $O(\log^2 n)$ time with roughly n^6 processors. The best approximation algorithm for the problem is due to Atallah et. al [AtaKosLarMilTen89] and requires $O(\log^2 n)$ time using $n^2/\log^2 n$processors.

References

[AbrDadKirPrz87] K.ABRAHAMSON, N.DADOUN, D.G.KIRKPATRICK, T.PRZYTYCKA, A simple parallel tree contraction algorithm, Proc. 25th Allerton Conference on Communication, Control and Computing (1987),

[AbrDadKirPrz87b] K.ABRAHAMSON, N.DADOUN, D.G.KIRKPATRICK, T.PRZYTYCKA, A simple optimal randomized parallel list ranking algorithm, Computer Science Department Technical Report 87-14, University of British Columbia Vancouver, to appear in *Information Processing Letters*.

[AbrDadKirPrz89] K.ABRAHAMSON, N.DADOUN, D.G.KIRKPATRICK, T.PRZYTYCKA, A simple parallel tree contraction algorithm, *Journal of Algorithms 10*, 1989, 287-302.

[AdkPen89] G.ADHAR, S. PENG, Parallel algorithms for cograph recognition and applications, Proc. of 1989 Workshop on Algorithms and Data Structures, August 1989, Ottawa, 335-351.

[AhoHopUll86] A.V.AHO, J.E.HOPCROFT, J.D.ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[All82] B.ALLEN, On the cost of optimal and near-optimal Binary Search Trees, *Acta Inform.* 18, 1982, 255-263.

[AltHagMehPre97] H.ALT, T.HAGERUP, K.MEHLHORN, F.P.PREPARATA, Simulation of idealized parallel computers on more realistic ones, *SIAM Journal of Computing*, **16** (**5**), 1987, 808-835.

[AndMil88] R.J.ANDERSON, G.L.MILLER, Deterministic parallel list ranking, VLSI Algorithms and Architecture, 3rd Egean Workshop on Computing (1988) 81-90.

[AtaColGoo89] M.J.ATALLAH, R.COLE, M.T.GOODRICH, Cascading divide-and-conquer: A technique for designing parallel algorithms", *SIAM, J. Comput.* Vol **18**, No **3**, 499-533, 1989.

[AtaKosLarMilTen89] M.J.ATALLAH, S.R.KOSARAJU, L.L.LARMORE, G.L.MILLER, S-H.TENG, Constructing trees in parallel, Proc 1st ACM Symposium on Parallel Algorithms and Architectures, 1989, 421-431.

[BarVis85] I. BAR-ON and U. VISHKIN, "Optimal parallel generation of a computation tree form". *ACM Transactions on Programming Languages and Systems* 7,2, 1985, pp. 348-357.

[Bay75] P.J.BAYER, Improved bounds on the cost of optimal and balanced Binary Search Trees, Project MAC Technical Memorandum 69, MIT, Cambridge, 1975.

[BeBrGalSchVis89] O.BERKMAN, D.BRSLAUER,Z.GALIL, B.SCHIEBER, U.VISHKIN, Highly parallelizable problems, Proc. of 21th Annual ACM Symposium on Theory of Computing, 1989, 309-320.

[BerLawWon85] M.W.BERN, E.L.LAWER, A.L.WONG, Why certain subgraph computation require only linear time, 26th Annual Symposium of Foundations of Computer Science, (1985) 117-125.

[Bop89] R.BOPPAMA, Optimal separations between Concurrent-Write Parallel Machines, Proc. of 21th Annual ACM Symposium on Theory of Computing, 1989, 320-327.

[Bor77] A.BORODIN, On relating time and space to size and depth, *SIAM J. Comput.*, vol 6, 1977, 733-744.

[BorHop85] A.BORODIN, J.E.HOPPCROFT, Routing, merging and sorting on parallel models of computation, *J. Comp. Sys. Sci.*, Vol. 30, 1985, 130-145.

[BurUhr84] M.BURLET, J.P.UHRY, Parity graphs, *Annals of Discrete Mathematics* **21** (1984) 253–277.

[BrSchVis88] D.BRSLAUER, B.SCHIBER, U.VISHKIN, Some double logarithmic parallel algorithms based on finding all nearest smaller values, UNIMACS-TR-88-79, University of Maryland Inst. for Advanced Comp. Studies (1988)

[Bre74] R.P.BRENT, The parallel evaluation of general arithmetic expressions, *JACM* **21** (1974) 201-208.

[Cal1857] A.CAYLEY, On the theory of the analytical forms called trees, *Philosophical Magazine*, vol. XIII, 1857, 172-176.

[Cal1859] A.CAYLEY, On the analytical forms called trees. Second part., *Philosophical Magazine*, vol. XVIII, 1859, 374-378.

[ChaKozSto81] A.K.CHANDRA, D.C.KOZEN, L.J.STOCKMEYER, Alternation, *JACM*, vol 28, 1981, 114-133.

[Col86] R.COLE, Parallel merge sort, Proc 27th Annual IEEE Symp. on Foundation of Computer Science, 1986, 511-516.

[ColVis86] R. COLE and U. VISHKIN, Approximate and exact parallel scheduling with applications to list, tree and graph problems, Proc. 27th Annual IEEE Symposium on Foundations of Computer Science (1986) 478-491.

[ColVis86a] R. COLE and U. VISHKIN. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms, Proc 18th Annual Symposium on Theory of Computing, 1986, pp. 206-219.

[ColVis88] R. COLE, U. VISHKIN, The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, *Algorithmica* **3, 3** (1988) 329-346.

[CooDwoRes86] S.A.COOK, C.DWORK, R.A.REISCHUK, Upper and lower bounds for parallel random access machines without simultaneous writes, *SIAM J. Computation*, vol 15, 1986, 87-97.

[CopWin87] D.COPPERSMITH, S.WINOGRAD, Matrix multiplication via arithmetic progressions, Proc. 28th Annual IEEE Symp. on Foundations of Comp. Sci., 1987, 260-270.

[CopKlaPip86] D.COPPERSMITH, M.M. KLAWE, N.J.PIPPENGER, Alphabetic minimax trees of degree at most t, *SIAM J. Comput.* Vol **15**, No **1**, 1986 189-192.

[CorLerS81] D.G.CORNEIL, H.LERCHS and L.STEWART, Complement reducible graphs, *Discrete Applied Mathematics* **3** (1981) 163-175.

[CorPerSte85] D.G.CORNEIL,Y.PERL,L.K.STEWART, A linear recognition algorithm for cographs, *SIAM J.Comput.* **14, 4** (1985) 926-934.

[DadKir87] N. DADOUN, D. KIRKPATRICK, "Parallel processing for efficient subdivision search". In 3rd ACM Symposium on Computational Geometry, Waterloo, Ontario, 1987, pp. 205-214.

[DahKa86] E.DAHLHAUS, M.KARPINSKI, The matching problem for strongly connected graphs is in NC, Research Report No 855-CS, University of Bonn, 1986.

[DahKa87] E.DAHLHAUS, M.KARPINSKI, Fast parallel computation of Perfect and Strongly Perfect elimination schemes, Research Report No 8513-CS, University of Bonn, 1987.

[DahKa89] E.DAHLHAUS, M.KARPINSKI, An efficient parallel algorithm for the Minimal Elimination Ordering (MEO) of an arbitrary graph, Proc 30th Annual IEEE Symp. on Foundation of Computer Science, (1989), 454-459.

[Dij59] E.W.DIJKSTRA, A note on two problems in connection with graphs, *Numer Math.*, **1**, 1959, 269-271.

[Ede87] A.EDENBRANDT, Chordal graphs recognition is in NC, *Information Processing Letters* **24**, 1987, 239-241.

[FicRagWid88] F.E.FICH, P.RAGDE, A.WIDGERSON, Relations Between Concurrent-Write Models of Parallel Computation, *SIAM, J.Comp.* Vol **17**, No **3** (1988), 606-677.

[ForWyl87] S.FORTUNE, J.WYLLIE, Parallelism in Random Access Machines, Proc. 10th Annual ACM Symp. on Theoretical Computer Science,1987, 114-118.

[Fre75] M.L.FREDMAN, Two applications of a probabilistic search technique: sorting X+Y and building balanced search trees, Proc 7th Annual ACM Symposium on Theory of Computation, 1975, 240-244.

[GarJon79] M.R.GAREY, D.S.JOHNSON,*Computers and Intractibility: A Guide to the Theory of NP-completness,* W.H.Freeman, San Francisco.

[GibRyt86] A. GIBBONS, W. RYTTER, An optimal parallel algorithm for dynamic tree expression evaluation and its applications, Proc.Symp. on Foundations of Software Technology and Theoretical Comp. Sci. (1986) 453-469.

[GibRyt88] A. GIBBONS, W. RYTTER, *Efficient Parallel Algorithms*, Cambridge University Press (1988).

[Go78] L.M.GOLDSCHLAGER, A unified approach to models of synchronous parallel machines, Proc. 10th Annual ACM Symp. on Theoretical Computer Science,1978, 89-94.

[Gol76] M.C.GOLUMBIC, Combinatorial merging, *IEEE Trans. Comp.* **25, 11** (Nov. 1976), 1164-1167.

[Gol80] M.C.GOLUMBIC, *Algorithic Graph Theory and Perfect Graphs*, Academic Press (1980).

[Goo87a] M.T.GOODRICH, Efficient Parallel Techniques for Computational Geometry, Ph.D. thesis, Purdue University, August 1987.

[Goo87b] M.T.GOODRICH, Finding the convex hull of a sorted point set in parallel, *Information Processing Letters* **26**, 1887, 173-179.

[Goo89] M.T.GOODRICH, Triangulation of a polygon in parallel, *Journal of Algorithms* **10** (1989) .

[He86a] X. HE, Efficient parallel algorithms for solving some tree problems, Proc. 24th Allerton Conference on Communication, Control and Computing (1986) 777-786.

[He86b] X.HE, Parallel recognition and decomposition of two terminal series parallel graphs, Computer & Information Science Research Center Technical Report, The Ohio State University Columbus (1986).

[HerBil88] K.T.HERLEY, G.BILARDI, Deterministic Simulations of PRAM's on bounded degree networks, Prec. of the 26th Annual Allerton Conference on Communication, Control, and Computation, Illinois, 1988.

[Her89] K.T.HERLEY, Efficient Simulations of Small Shared Memories on Bounded Degree Networks, Proc. 30th IEEE Symp. on Foundations of Computer Science, 1989, 390-395.

[HirChaSar79] D.S.HIRSCHBERG, A.K.CHANDRA, D.V.SARWATE, Computing connected components on parallel computers, *Com. of ACM*, **22, 8** (1979) 461-464.

[HooKlaPip84] H.J.HOOVER, M.M.KLAWE, N.J.PIPPENGER, Bounding Fan-out Logical Networks, *Journal of the ACM*, Vol **31**, No **1** (1984) 13-18.

[HorPre89] S.W.HORNICK, F.P.PREPARATA, Deterministic PRAM simulation with constant redundancy, Proc of the 2nd ACM Symposium on Parallel Algorithms, Santa Fe, New Mexico, 1989,

[Hu73] T.C.HU, "A new proof of the T-C algorithm", *SIAM J.Appl. Math.*, Vol. **25**, No **1**, July 1973, 83-94.

[HuKleTan79] T.C.HU, D.J. KLEITMAN, J.K.TAMAKI, Binary trees optimum under various criteria, *SIAM J. Appl. Math.*, **37**, 1979, 246-256.

[Huf52] D.A.HUFFMAN, A method for the construction of minimum redundancy codes, Proc. IRE, 40, 1952, 1098-1101.

[Jar30] V.JARNIK, O jistem problemu minimalnim, Praca, *Moravske Prirodovedecke Splecnosti*, 1930, 57-63 (in Czech).

[Jun78] H.A.JUNG, On a class of posets and corresponding comparability graphs, *J.Combinatorical Theory* (B) **24** (1978) 125-133.

[KaUpf88] A.KARLIN, E.UPFAL, Parallel hashing-an efficient implementaiton of shared memory, *SIAM Journal of Computing*, 15 (4), 1988, 876-892.

[KarRam88] R.M.KARP, V.RAMACHANDRAN, A survey of parallel algorithms for shared-memory machines, Report No.UCB/CSD 88/408, Computer Science Division University of California, Berkeley (1988).

[Ki1847] G.KIRCHHOFF, Uber die Aufloosung der Gleichungen, auf welche man beider Undersuchund der liniearen Vertheilung gallvanischer Stromegefurd wird, *Ann Phis, Chem.* **72** (1847) 497-508.

[KirKla85] D.G.KIRKPATRICK, M.M.KLAWE, Alphabetic Minimax Trees, *SIAM J. Comput*, Vol **14**, No. **3** (1985) 514-526.

[KirPrz88] D.G.KIRKPATRICK, T.PRZYTYCKA, Parallel recognition of cographs and cotree construction, Computer Science Department Technical Report 88-1,University of British Columbia, Vancouver, January 1988; to appear in *Journal of Discrete Math.*

[KirPrz89] D.G.KIRKPATRICK, T.PRZYTYCKA, Parallel constructions of binary trees with almost optimal weighted path length, TR 89-25,to appear in Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures.

[KirRe84] D.G.KIRKPATRICK, S.REISCH, Upper bound for sorting integers on random access machines, *Theoretical Computer Science* **28** (1984) 236-276.

[Kle88] P.KLEIN, Efficient parallel algorithms for chordal graphs, Proc. 29th Symp. of Foundation of Comp. Sci., (1988), 150-161.

[Knu68] D. KNUTH, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Addison-Wesley, 1968.

[Kos89] S.RAO KOSARAJU, Pipelining Computation in a tree of processors, Proc 30th Annual IEEE Symp. on Foundation of Computer Science, (1989), 184-189.

[KosDel88] S.RAO KOSARAJU, A.L.DELCHER, Optimal parallel evaluation of tree structured computation by raking, Proc. 3rd Aegean Workshop on Computing (1988).

[KruRudSni85] C.P.KRUSKAL, L.RUDOLF, and M.SNIR, Efficient parallel algorithms for graph problems, International conference on parallel processing, 1985, 180-185.

[LadFis80] R.E.LADNER, M.J.FISCHER, Parallel Prefix Sum Computation, *Journal of the ACM*, 1980, 830-838.

[Lar87] L.L.LARMORE, A subquadratic algorithm for constructing Approximately Optimal Binary Search Trees, *Journal of Algorithms* **8**, 1987, 579-591.

[Ler71] H. LERCHS, On cliques and kernels, Dept. of Comp. Science Technical Report, University of Toronto (1971).

[Ler72] H.LERCHS, On the clique-kernel structure of graphs, Dept. of Comp. Science Technical Report, University of Toronto (1972).

[McC60] J.MCCARTHY, Recursive functions of symbolic expressions and their computation by machine, Part I, *Com. ACM*, **3**, 1960, 184-195.

[Meg83] N. MEGIDDO, Applying parallel computation algorithms in the design of serial algorithms, *Journal of the ACM,* 30, 4, Oct. 1983, pp. 852-865.

[Meh75] K. MEHLHORN, Nearly optimal binary search trees, *Acta Informatica* **5** (1975) 287-295.

[Meh84] K. MEHLHORN, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag (1985).

[Mer86] E.MERKS, An optimal parallel algorithm for triangulating a set of points in the plane, *International Journal of Parallel Programming*, Vol **15**, No **5** (1986) 339-411.

[Mey84] H.MEYNIEL, The graphs whose odd cycles have at least two crossing chords, *Annals of Discrete Mathematics* **21** (1984) 115-119.

[MilRamKal86] G.L.MILLER, V.RAMACHANDRAN, E.KALTOFEN, Efficient parallel evaluation of straight line code and arithmetic circuits, Proc. 2nd Aegean Workshop on Computing (1986)

[MilRei85] G. L. MILLER and J. REIF, Parallel tree contraction and its application, Proc. 26th IEEE Symp. on Foundations of Computer Science (1985) 478-489.

[MilTen87] G.L.MILLER, S-H.TENG, Systematic method for tree based parallel algorithm development, Second International Conference on Supercomputing, 1987.

[MilTen87] G.L.MILLER, S-H.TENG, Dynamic computing of computational circuits, Technical Report, CRI 87-17 Dept. of Comp. Sciences, University of Southern California, Los Angeles.

[NaoNaoSc87] J.NAOR, M.NAOR, A.A.SCHAFFER, Fast parallel algorithms for chordal graphs, Proc. 19th Annual ACM Symp. on Theory of Computing (1987) 355-364.

[Nov89] M.B.NOVICK, Fast parallel algorithms for the modular decomposition, Notes, Cornell University, Ithaca.

[Nov90a] M.B.NOVICK, Parallel algorithms for the split decomposition, Notes, Cornell University, Ithaca.

[Nov90b] M.B.NOVICK, Private communication.

[Ofm63] YU. OFMAN, On the algorithmic complexity of discrete functions, *Soviet Phisics - Doklady*, Vol **7** no **7**, 1933, 589-591.

[PatHew70] M.S.PATERSON, C.E.HEWITT, Comparative Schematology, *Project MAC Conference on Concurrent Systems and Parallel Computation*, Woods Hole, MA, 1970, 119-127.

[Pip79] N.PIPPENGER, On simulating resource bounds, Proc. 20th Annual IEEE Symposium on Foundation of Computer Science, 1979, 307-311.

[PraSto89] V.R.PRATT, L.J.STOCKMEYER, A characterization of the power if vector machines, *J.Comput. System Sci*, vol 12, 1979, 198-221.

[Pri55] R.C.PRIM, Shortest connection networks and some generalizations, Bell System Tech., J., 36, 1955, 1389-1401.

[PrzCor88] T.PRZYTYCKA, D.G.CORNEIL, Parallel algorithms for parity graphs, to appear in *Journal of Algorithms.*

[Ran87] A.G.RANADE, How to emulate shared memory, Proc. 28th IEEE Symp. on Foundations of Computer Science, 1987, 185-192.

[Rei85] J.H.REIF, Depth-First Search is inherently sequential, *Information Processing Letters* 20,1985, 119-234.

[Ruz80] W.L.RUZZO, Tree-size bounded alternation, *J.Comp. Syst. Sci.*, vol **21**, 1980, 218-235.

[Ruz81] W.L.RUZZO, On uniform circuit complexity, *J.Comp. Syst. Sci.*, vol **22**, 1981, 365-383.

[Ryt85] W.RYTTER, The complexity of two way pushdown automata and recursive programs, Combinatorical Algorithms on Words, eds

Apostolica and Z.Galil, NATO ASI series, F:12, Springert Verlag (1985).

[RytSzy89] W.RYTTER, T.SZYMACHA, Parallel algorithms for a class of graphs generated recursively, *Information Processing Letters* **30**, 1989, 225-231.

[Sac70] H.SACHS, On the Berge conjecture concerning perfect graphs, in *Combinatorial Structures and their Applications*, Gordon and Breach, pp 377-384, New York, 1970.

[ShiVisi81] Y.SHILOACH, U.VISHKIN, Finding the maximum, merging, and sorting in a parallel computation model, *Journal of Algorithms* **2**, (1981), 88-102.

[Shy88] C.H.SHYU, A fast algorithm for cographs, presented at the French Israeli Conference on Combinatorics and Algorithm, November 1988.

[Sli82] A.SLISENKO, Context-free grammars as a tool for describing polynomial time subclasses of hard problems, *Inform. Process. Lett.* **14** (2) (1982) 52-57.

[Ste78] L. STEWART, Cographs, A Class of Tree Representable Graphs, M. Sc. Thesis, Dept. of Computer Science, Univ. of Toronto (1978).

[Sum74] D.P.SUMNER, Dacey graphs, *J.Australian Math. Soc.* **18, 4** (1974) 492-502.

[Sys84] M.M.SYSLO, NP-complete problems on some tree structured graphs: a review, Proc. of the Workshop on Graph Theoretical Concepts in Computer Science, Universitat Osnabruck, June 1983, 342-353.

[Tar83] R. E. TARJAN, *Data Structures and Network Algorithms*, SIAM 1983.

[TarVis84] R. E. TARJAN, U. VISHKIN, Finding biconnected components and computing tree functions in logarithmic parallel time, In *25th Annual Symp. on Foundations of Comp. Science,* 1984, pp. 12-22.

[Ten87] S-H.TENG, The construction of Huffman equivalent prefix code is in NC, *ACM SIGACT*, **18(4)** 1987, 54-61.

[UpfWid87] E.UPFAL. A. WIDGERSON, How to share memory in a distrubuted system, *Journal of the ACM*, **34 (1)** , 1987, 185-192.

[Val75] L.VALIANT, Parallelism in comparison problems, *SIAM J.Comput.*, vol. **4**, 1975, 348-355.

[Vis83a] U. VISHKIN, Synchronous parallel computation — a survey, TR-71, Dept. of Computer Science, Courant Institute, NYU, 1983.

[Vis83b] U. VISHKIN, Implementation of simultaneous memory access in model that forbid it, *Journal of Algorithms*, vol **4**, 1983, 45-56.

[Vis84] U. VISHKIN, Randomized speed-ups in parallel computation, Proc. 16th Annual ACM Symp. on Theory of Computing (1984) 230-239.

[Wyl81] J.WYLLIE, The complexity of parallel computation, Ph.D. dissertation, Computer Science Department Cornell Univ., Ithaca, NY, 1981

Index