Sequential and Parallel Algorithms for Generalized Coloring of Trees

by

Tarique Mesbaul Islam

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

April 2002

Submitted to
Bangladesh University of Engineering and Technology

in partial fulfillment of the requirements for
M.Sc. Engineering (Computer Science and Engineering)

SEQUENTIAL AND PARALLEL ALGORITHMS FOR GENERALIZED COLORING OF TREES

A Thesis submitted by

TARIQUE MESBAUL ISLAM
Student No. 040005021P
for the partial fulfillment of the degree of
M. Sc. Engineering (Computer Science and Engineering).
Examination held on April 6, 2002.

Approved as to style and contents by:

[illegible]

DR. MD. ABUL KASHEM MIA Associate Professor & Head Department of Computer Science and Engineering B.U.E.T., Dhaka – 1000, Bangladesh.	Chairman, Supervisor and Ex-officio

[illegible]

Dr. Chowdhury Mofizur Rahman Professor Department of Computer Science and Engineering B.U.E.T., Dhaka – 1000, Bangladesh.	Member

[illegible]

Dr. Muhammad Masroor Ali Associate Professor Department of Computer Science and Engineering B.U.E.T., Dhaka – 1000, Bangladesh.	Member

[illegible]

Dr. Abul L. Haque Associate Professor & Chairman Dept. of Computer Science North South University, Dhaka	Member (External)

Certificate

This is to certify that the work presented in this thesis paper is the outcome of the investigation carried out by the candidate under the supervision of Dr. Md. Abul Kashem Mia in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka. It is also declared that neither of this thesis nor any part thereof has been submitted or is being concurrently submitted anywhere else for the award of any degree or diploma.

[illegible]

Signature of the Supervisor

[illegible]

Signature of the Author

List of Figures

List of Tables

List of Algorithms

Abstract

In this thesis we present efficient sequential and parallel algorithms for solving generalized vertex-coloring (l-vertex-coloring) problem and generalized edge-coloring (l-edge-coloring) problem on trees. For any positive integer l, an l-vertex-coloring of G is an assignment of colors to the vertices of G in such a way that any two vertices u and v in G get different colors if $dist(u,v) \leq l$, where $dist(u,v)$ is the length of the shortest path between u and v in G. An l-vertex-coloring is optimal if it uses minimum number of colors. The l-vertex-coloring problem has applications in various scheduling problems. We present an $O\left(\log\alpha \times (l+2)^{2\alpha} \times n\right)$ time sequential algorithm to solve the l-vertex-coloring problem on trees T, where $\alpha = \frac{\Delta^{l+1}-1}{\Delta-1}$, n is the number of nodes in the tree T and Δ is the maximum vertex-degree in T. If both Δ and l are bounded integers, then our algorithm runs in linear time. This is the first sequential algorithm that guarantees an optimal solution for the problem. We then present a parallel algorithm that solves the l-vertex-coloring problem on trees in $O(\log_2 n)$ time using $O\left(\alpha \times (l+2)^{2\alpha} \times n\right)$ processors on the EREW PRAM model. The l-edge-coloring problem on trees can be defined analogous to l-vertex-coloring problem on trees. We then present a simple strategy to transform l-edge-coloring problem to l-vertex-coloring problem and develop a sequential as well as a parallel algorithm to solve the l-edge-coloring problem on trees. The sequential l-edge-coloring algorithm takes $O\left(\log\alpha \times (l+2)^{2\alpha} \times n\right)$ time while the parallel l-edge-coloring algorithm takes $O(\log_2 n)$ time using $O\left(\alpha \times (l+2)^{2\alpha} \times n\right)$ processors on the EREW PRAM model. All our algorithms, except the sequential algorithm for l-vertex-coloring problem, are first known algorithms to solve the corresponding problems.

Acknowledgement

First and foremost, I would like to acknowledge my gratitude to Dr. Md. Abul Kashem Mia, Associate Professor and Head, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology. His constant supervision, scholarly guidance, valuable advice and encouragement have been a great impetus to this thesis work. His long experience and in-depth knowledge in graph theory and parallel algorithms have helped significantly to achieve a smooth completion of this thesis work.

I would like to thank the members of the graduate committee, Dr. Chowdhury Mofizur Rahman, Professor Department of Computer Science and Engineering, BUET, Dr. Muhammad Masroor Ali, Associate Professor, Department of Computer Science and Engineering, BUET and Dr. Abul L. Haq, Associate Professor and Chairman, Department of Computer Science, North South University, for their valuable suggestions.

I would like to acknowledge the all-out co-operation and services rendered by the faculty members and staff of the CSE Department. Finally, I express my ever gratefulness to all who contributed to this thesis work.

Chapter 1

Introduction

In this chapter we provide the necessary background and motivation for the study on the coloring of graphs. In Section 1.1, we give a historical background of the development of coloring of graphs. In Section 1.2, we give a generalization of ordinary vertex-coloring problem and provide the necessary motivation for generalization. In Section 1.3 we define a generalization of ordinary edge coloring problem. Finally Section 1.4 summarizes our results together with the known ones.

1.1 Background

Recent research efforts in algorithm theory have concentrated on designing efficient algorithms for solving combinatorial problems, particularly graph problems. A graph $G = (V, E)$ with n vertices and m edges consists of a vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and an edge set $E = \{e_1, e_2, \ldots, e_n\}$, where an edge in E joins two vertices in V. Figure 1.1 depicts a graph of 8 vertices and 11 edges, where vertices are drawn by circles, edges by lines,

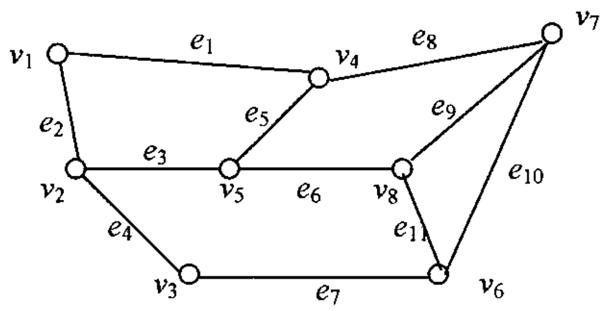

Figure 1.1: A graph with 8 vertices and 11 edges.

vertex names next to the vertices and edge names next to the edges. Efficient algorithms have been obtained for various graph problems, such as coloring problems, planarity testing problem and maximum flow problem.

1.1.1 Vertex-coloring Problem

The vertex-coloring problem is one of the most fundamental problems on graphs. A *vertex-coloring* of a graph G is an assignment of colors to the vertices of G in such a way that any two adjacent vertices get different colors [Wes99]. The *vertex-coloring problem* is to find a vertex-coloring that requires the minimum number of colors. The minimum number of colors needed to vertex-color a graph is called the *chromatic number* of the graph G and is denoted by $\chi(G)$. Figure 1.2 depicts an optimal vertex-coloring of a graph G using 3 colors, where the colors are written next to the vertices.

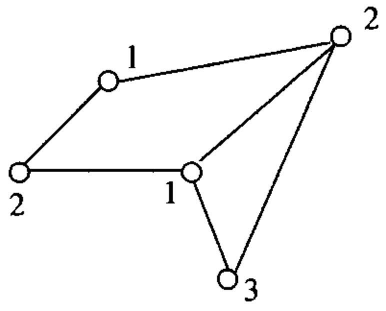

Figure 1.2: An optimal vertex-coloring of a graph G.

A graph G is k-colorable (or k-vertex-colorable, more precisely) if there is a valid vertex-coloring of G with k colors. If $\chi(G) = k$, then G is k-chromatic. All vertices having the same color are in the same color class and form an independent set. Therefore, G is k-colorable if and only if G is k-partite. Let $\omega(G)$ is the number of vertices in the largest complete subgraph of G. Since vertices of a complete subgraph require distinct colors, $\chi(G) \geq \omega(G)$. Suppose, $\alpha(G)$ is the maximum size of an independent set of vertices in G and G contains n vertices. Since each color class is an independent set, $\chi(G) \geq n/\alpha(G)$. Let Δ(G) be the maximum degree of any vertex in G. Then $\chi(G) \leq \Delta(G) + 1$. The details of these observations can be found in [Wes99].

Vertex coloring arises in a variety of scheduling and clustering applications. Compiler optimization is the canonical application for coloring, where we seek to schedule the use of a finite number of registers. In a program fragment to be optimized, each variable has a range of times during which its value must be kept intact, in particular, after it is initialized and before its final use. Any two variables whose life spans intersect cannot be placed in the same register. A graph can be constructed where there is a variable associated with each vertex and an edge between any two vertices indicates that the variable life spans intersect. A coloring of the vertices of this graph assigns the variables to classes such that two variables with the same color do not clash and so can be assigned to the same register. The most important application of vertex-coloring is in scheduling of any kind. If the vertices of a graph G represent a set of university courses, with edges between courses with common students, then the chromatic number is the minimum number of periods needed to schedule examinations without conflicts. One of the most famous problems in graph theory also involves coloring. It is known as 4-color problem. A map is a partition of the plane into connected regions. It requires determining whether the regions of every map can be colored using at most four colors so that no two neighbouring regions have the same color.

1.1.2 Edge-coloring Problem

In graph theory, many problems about vertices have natural analogues for edges. Edge-coloring problem is one such example. An *edge-coloring* of a graph G is a labeling of the edges in G in such a way that any two adjacent edges get different colors [Wes99]. A graph G is k-edge-colorable if G has a valid edge-coloring with k colors. The *edge-chromatic number* $\chi'(G)$ of a loopless graph G is the minimum k such that G is k-edge-colorable. $\chi'(G)$ is also called *chromatic index* in some literature. The *edge-coloring problem* is to find an edge-coloring that requires minimum number of colors. Figure 1.3 shows an edge-coloring of a graph G using 3 colors where the colors are written next to the edges. It may be noted that a graph consisting of an even-length cycle can be edge-colored with 2 colors, while odd-length cycles have an edge-chromatic number of 3.

As it was with vertex-coloring, various theorems exist that indicate bounds for $\chi'(G)$. Since edges sharing a common vertex need different colors, $\chi'(G) \geq \Delta(G)$. Vizing [Viz64] and Gupta [Gup66] independently proved that $\Delta(G) + 1$ colors suffice when G is simple.

They also proved that every simple graph with maximum degree Δ has a valid $(\Delta + 1)$-edge-coloring. For graphs with multiple edges, Vizing and Gupta proved that $\chi'(G) \le \Delta(G) + \mu(G)$, where $\mu(G)$ is the maximum edge multiplicity.

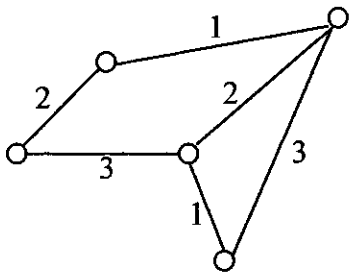

Figure 1.3 Edge-coloring of a graph G.

The edge coloring of graphs arises in a variety of scheduling applications, typically associated with minimizing the number of non-interfering rounds needed to complete a given set of tasks. For example, consider a situation where we need to schedule a given set of two-person interviews, where each interview takes one hour. All meetings could be scheduled to occur at distinct times to avoid conflicts, but it is less wasteful to schedule non-conflicting events simultaneously. We can construct a graph whose vertices are the people and whose edges represent the pairs of people who want to meet. An edge coloring of this graph defines the schedule. The color classes represent the different time periods in the schedule, with all meetings of the same color happening simultaneously. Scheduling games in a football league is another example where edge-coloring can be applied.

1.2 Generalized Vertex-Coloring

There are many generalizations of ordinary vertex-coloring. In this section we describe a generalized vertex-coloring called an l-vertex-coloring [ZKN00].

1.2.1 *l*-vertex-coloring

A natural generalization of the ordinary vertex coloring is the l-vertex-coloring. Let l be a positive integer and G be a graph with positive integer weights on all its edges. Then an l-vertex-coloring of G is an assignment of colors to the vertices of G in such a way that any two vertices u and v in G get different colors if $dist(u,v) \leq l$, where $dist(u,v)$ is the length of the shortest path between u and v in G. Clearly an ordinary vertex-coloring is a 1-vertex-coloring. Figure 1.4 shows a 3-vertex-coloring of a graph G using 2 colors where colors are drawn next to the vertices and edge-weights are given next to the edges.

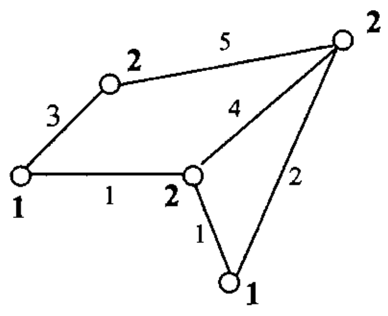

Figure 1.4: A 3-vertex-coloring of a graph using 2 colors.

The minimum number of colors needed for an l-vertex-coloring of a graph G is called the l-chromatic number of G and is denoted by $\chi_l(G)$. An l-vertex-coloring of G using $\chi_l(G)$ colors is called an optimal l-vertex-coloring of G. The l-vertex-coloring problem is to find an optimal l-vertex-coloring of a given graph G.

The applications of the l-vertex-coloring are all the applications of vertex-coloring. The examination scheduling problem in Section 1.1.1 can be generalized now so that there is a gap of l periods between any two examinations of courses having common students.

1.3 Generalized Edge-Coloring

Like the generalization of vertex-coloring, there are different generalizations for edge-coloring problem. In this section we describe a generalized edge-coloring called an l-edge-coloring.

1.3.1 l-edge-coloring

A *generalized edge-coloring* or l-*edge-coloring* of a graph G is an assignment of colors to the edges of G in such a way that any two edges e_1 and e_2 in G get different colors if $dist(e_1, e_2) \leq l$, where $dist(e_1, e_2)$ is the length of the shortest path between the nearest endpoints of e_1 and e_2 in G. The minimum number of colors with which we can obtain an l-edge-coloring of a graph G is called the *l-edge-chromatic number* of G. The l-edge-coloring problem is to find an optimal l-edge-coloring of a given graph G. Figure 1.5 shows a 2-edge-coloring of a graph with 4 colors.

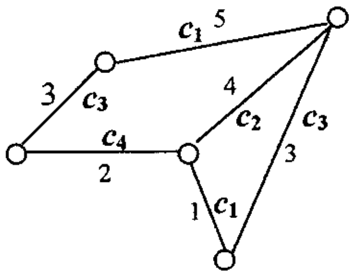

Figure 1.5: 2-edge-coloring with 4 colors.

l-edge-coloring can be applied to all applications where edge-coloring can be applied. However, l-edge-coloring can incorporate more constraints than the ordinary edge-coloring. For example, the problem scheduling of football matches can be extended so that rematches are scheduled with a reasonable time gap between them.

1.4 Summary

This thesis gives sequential and parallel algorithms for solving *l*-vertex-coloring problem and *l*-edge-coloring problem on trees. In this section, we summarize our main results. The known results are given in Table 1.1. Our new results are given in Table 1.2.

Classes of Graphs	**Time Complexity**	**Reference**
Partial *k*-tree	$O\left(n^3 + n \times (\alpha+1)^{2^{2(k+1)(l+2)+1}}\right)$	Zhou *et al.* (IEICE 2000)
Tree ($k = 1$)	$O\left(n^3 + n \times \alpha^{2^{4l+9}}\right)$	Zhou *et al.* (IEICE 2000)
Tree	$O(n \times \alpha)$	Kashem *et al.* (ICCIT 2000)

Table 1.1: Known results for *l*-vertex-coloring.

Classes of Graphs	**Problem**	**Algorithm**	**Complexity**
Trees	*l*-vertex-coloring	Sequential	$O\left(\log\alpha \times (l+2)^{2\alpha} \times n\right)$ time
Trees	*l*-edge-coloring	Sequential	$O\left(\log\alpha \times (l+2)^{2\alpha} \times n\right)$ time
Trees	*l*-vertex-coloring	Parallel	$O(\log_2 n)$ time using $O\left(\alpha \times (l+2)^{2\alpha} \times n\right)$ processors on the EREW PRAM model
Trees	*l*-edge-coloring	Parallel	$O(\log_2 n)$ time using $O\left(\alpha \times (l+2)^{2\alpha} \times n\right)$ processors on the EREW PRAM model

Table 1.2: Our new results for *l*-vertex-coloring and *l*-edge-coloring.

The symbol α in table 1.1 stands for the number of colors used in the coloring. First algorithm has been developed actually for solving *l*-vertex-coloring problem on partial *k*-trees and second algorithm is a restriction of first algorithm with $k = 1$. Third algorithm is the first known direct approach to solve the *l*-vertex-coloring problem on trees. This algorithm uses a greedy strategy to assign colors to the nodes in a post order fashion.

Though third algorithm runs in linear time it does not guarantee an optimal solution. No parallel algorithm exists for l-vertex-coloring problem. Moreover, no attempt has been made to date to solve the l-edge-coloring problem.

In this thesis, we present sequential and parallel algorithms to solve the l-vertex-coloring and l-edge-coloring problems on trees. For bounded l and α, our sequential algorithm for l-vertex-coloring algorithm can find a valid coloring of a given tree with α colors in linear time. Moreover, it can find the optimal number of colors as well as the optimal l-vertex-coloring in time $O(\log\alpha \times (l+2)^{2\alpha} \times n)$, where $\alpha = \frac{\Delta^{l+1}-1}{\Delta-1}$, is the upper bound of the number of colors. Our remaining algorithms are the first known algorithms of their kind.

The thesis is organized as follows. Chapter 2 gives preliminary definitions. We give an efficient sequential algorithm for l-vertex-coloring problem on trees in Chapter 3. Chapter 4 gives an efficient parallel algorithm for the same problem. In Chapter 5, we present a sequential and a parallel algorithm to solve the l-edge-coloring problem. Finally Chapter 6 concludes with the results and future works.

Chapter 2

Preliminaries

In this chapter we present some basic terms and easy observations. Definitions that are not included in this chapter will be introduced, as they are needed. In Section 2.1, we start with some definitions of the standard graph-theoretical terms used throughout the thesis. In Section 2.2 we discuss about properties of trees. In Section 2.3 we define parallel computer models most widely used in parallel algorithm development and analysis. Section 2.4 describes tree contraction algorithm that is used in Bottom-up Algebraic Tree Computation problems. And finally in Section 2.5 we show a way to convert a rooted tree into equivalent binary tree so that we can apply tree contraction algorithm on the equivalent binary tree.

2.1 Basic Terminology

2.1.1 Graphs and Multigraphs

A *graph* is a structure (V, E) which consists of a finite set of vertices V and a finite set of edges E; each edge is an unordered pair of distinct vertices. We call $V(G)$ the *vertex-set* of graph G and $E(G)$ the *edge-set* of G. Throughout this thesis, the number of vertices of G is denoted by n, that is $n = |V|$ and the number of edges of G is denoted by m, that is $m = |E|$. If $e = (v, w)$ is an edge, then e is said to join the vertices v and w and these vertices are said to be *adjacent*. In this case we also say that w is a neighbour of v and that e is incident to v and w. If the graph G has no "multiple edges" or "loops" then G is said to be a *simple graph*. *Multiple edges* join the same pair of vertices while a *loop* joins a vertex to itself. The graph in which loops and multiple edges are allowed is called a *multigraph*. Sometimes a simple graph is simply called *graph*, if doing so creates no confusion.

2.1.2 Degree of a vertex

The degree of a vertex v in a graph G is the number of edges incident to v and is denoted by $d_G(v)$ or simply by $d(v)$. The maximum degree of G is denoted by $\Delta(G)$ or simply by Δ. A vertex of degree 0 is called an isolated vertex.

2.1.3 Subgraphs

A *subgraph* of a graph $G = (V, E)$ is graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$; we write this as $G' \subseteq G$. If G' contains all the edges of G that join two vertices in V' then G′ is said to be the *subgraph induced by* V' and is denoted by $G[V']$. If V' consists of exactly the vertices on which edges in E' are incident then G' is said to be the *subgraph induced by* E' and is denoted by $G[E']$.

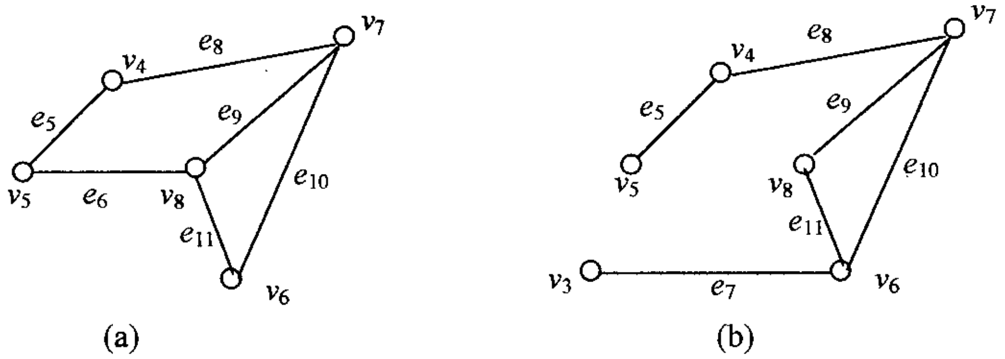

Figure 2.1: Subgraphs of G in Fig. 1.1: (a) vertex-induced subgraph, and (b) edge induced subgraph.

We often construct new graphs from old ones by deleting some vertices or edges. If v is a vertex of a given graph $G = (V, E)$ then $G - v$ is a subgraph of G obtained by deleting the vertex v and all the edges incident to v. More generally, if V' is a subset of V then $G - V'$ is subgraph of G obtained by deleting the vertices in V' and the edges incident to them. Then $G - V'$ is a subgraph of G induced by $V - V'$. Similarly if e is an edge of G then $G - e$ is th subgraph of G obtained by deleting the edge e. More generally, if $E \subseteq E'$ then $G - E'$ is a subgraph of G obtained by deleting the edges in E'.

2.1.4 Weighted Graphs

A graph where each of the edges has a positive weight associated with it is called a *weighted graph*. Now if I^+ is the set of all positive integers then we can define the weight function for the edges as $w: E \rightarrow I^+$.

2.1.5 Paths and Distances

A v_0–v_l walk in G is an alternating sequence of vertices and edges of G, $v_0, e_1, v_1, \ldots, v_{l-1}, e_l, v_l$, beginning and ending with a vertex, in which each edge is incident to two vertices immediately preceding and following it. If the vertices v_0, $v_1, \ldots, v_l$ are distinct (except possibly v_0, v_l), then the walk is called a *path* and is usually denoted by v_0 $v_1 \ldots v_l$. The *length* of a path in an unweighted graph is l, one less than the number of vertices on the path. A path or walk is *closed* if $v_0 = v_l$. A closed path of length at least one is called a *cycle* where $v_0 = v_l$ is the only vertex repetition.

In a weighted graph, the length of a path is determined by weights of the edges constituting the path. So the length $w(P)$ of a path P is defined as $w(P) = \sum_{e \in P} w(e)$. The distance between any two vertices in a graph is the length of the shortest path in the graph between the two vertices. We denote the distance from a vertex u to another vertex v by $dist(u,v)$. Now if the shortest path in G from u to v is P, then $dist(u,v) = w(P)$.

2.2 Trees

A (free) tree is a connected graph without any cycles. We often omit the word "free" when we say that a graph is a tree. Fig 2.2 is an example of a tree. A rooted tree is a free tree in which one of the nodes is distinguished from others. This distinguished node is called the *root* of the tree. The root of the tree is generally drawn at the top. In Fig. 2.2, the root is node 1.

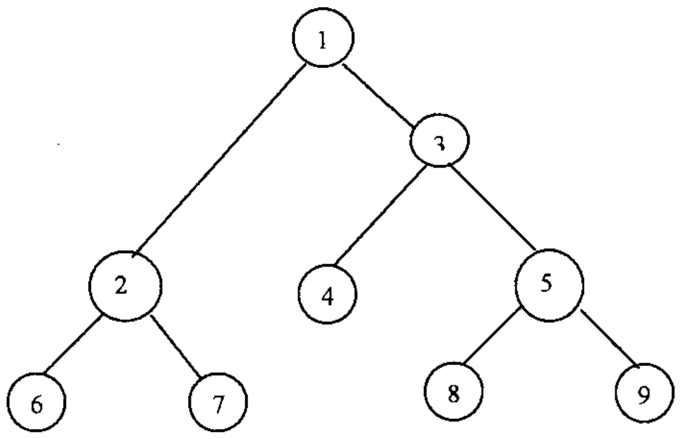

Figure 2.2: A tree with 9 vertices.

Every vertex *u* other than the root is connected by an edge to some other vertex *p* called the *parent* of *u*. We also call *u* a child of *p*. We draw the parent of a node above that node. For example, in Fig. 2.2, node 1 is the parent of node 2 and node 3. Alternately, nodes 6 and 7 are children of node 2. A leaf is a node of a tree that has no child. Thus every node of a tree is either a leaf or an internal node, but not both. In Fig. 2.2, the leaves are 4, 6, 7, 8 and 9 and nodes 1, 2, 3 and 5 are internal nodes.

The parent-child relationship can be extended naturally to ancestors and descendants. Suppose, u_1, u_2, ..., u_l is a sequence of nodes in a tree such that u_1 is the parent of u_2, which is the parent of u_3 and so on. The node u_1 is called an ancestor of u_l and node u_l is called a descendant of u_1. The root is the ancestor of every node in a tree and every node is a descendant of the root. In Fig. 2.2, all nodes other than node 1 are descendants of node 1 and node 1 is an ancestor of all other nodes.

In a tree *T*, a node *u* together with all of its descendants, if any, is called a *subtree* of *T*. Node *u* is the root of this subtree. Referring again to Fig. 2.2, node 6 by itself is a subtree, since node 6 has no descendant. Again, nodes 2, 6 and 7 form a subtree with root 2. Finally the entire tree of Fig. 2.2 is a subtree of itself with root 1. The height of a node *u* in a tree is the length of the longest path from *u* to a leaf. The height of a tree is the height of a root. The depth of a node *u* in the tree is the length of a path from the root to *u*. In Fig 2.2, for example, node 3 is of height 2 and depth 1. The tree has height 3.

2.3 PRAM Models

The RAM model has been used successfully to predict the performance of sequential algorithms. The PRAM model is natural extension of the basic sequential model. The PRAM model consists of a number of processors, each of which has its own local memory and can execute its own local program. The processors communicate by exchanging data through a shared memory unit. Each processor is uniquely identified by an index, called a *processor id.* All the processors operate synchronously under the control of a common clock. Fig. 2.3 shows a general view of a PRAM model with p processors. These processors are indexed $1, 2, \dots, p$. Shared memory is also referred to as global memory.

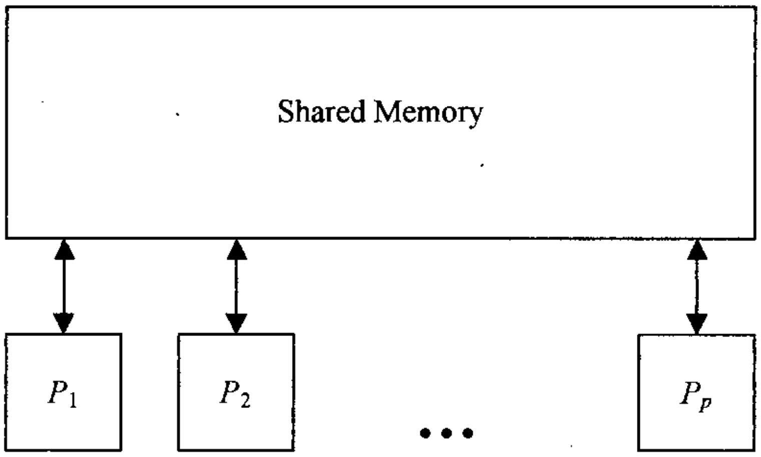

Figure 2.3: The PRAM model.

There are two basic modes of operation of a shared-memory model. In *asynchronous* mode, *each processor operates under a separate clock* and it is the programmer's responsibility to set appropriate synchronization points whenever necessary. In *synchronous* mode, *all the processors operate synchronously under the control of a common clock.* A standard name for the synchronous shared-memory model is the *parallel random-access machine* (PRAM) model. Since each processor can execute its own local program, the shared memory model is a *multiple instruction multiple data* (MIMD) type. That is, each processor may execute an instruction and operate on data different from those executed or operated on by any other processor during any given time unit.

There are several variations of the PRAM model based on the assumption regarding the handling of the simultaneous access of several processors to the same location of the shared-memory. The *exclusive read exclusive write* (*EREW*) PRAM does not allow any simultaneous access to a single memory location. The *concurrent read exclusive write* (*CREW*) PRAM allows simultaneous access for a read instruction only. The *concurrent read concurrent write* (*CRCW*) allows simultaneous access for a read or a write instruction. The three principal varieties of CRCW PRAMs are differentiated by how concurrent writes are handled. The *common* CRCW PRAM allows concurrent writes only when all processors are attempting to write the same value. The *arbitrary* CRCW PRAM allows an arbitrary processor to succeed. The *priority* CRCW PRAM assumes that the indices of the processors are linearly ordered and allows the one with minimum index (or the maximum index) to succeed.

2.4 Tree Contraction Algorithm

Bottom-up Algebraic Tree Computation (B-ATC) technique is widely used to develop parallel algorithms to solve various problems. B-ATC technique uses *tree contraction* algorithm. Tree contraction is a systematic way of shrinking a tree into a single node by successively applying the operation of merging a leaf with its parent or merging a degree-2 vertex with its parent. Miller and Reif [MR86] originally introduced the tree contraction algorithm with $O(\log_2 n)$ running time and $O(n)$ processors. Several authors have improved their algorithm to run in $O(\log_2 n)$ time with $O(n/\log_2 n)$ processors [GMT88, ADK89]. Now we briefly describe the version developed in [ADK89].

A structure is a triple (*S*, *NF*, *EF*) consisting of a set *S*, a *vertex function* set $NF \subseteq \{f \mid f : S \times S \to S\}$ and an edge function set $EF \subseteq \{f \mid f : S \to S\}$. A *bottom-up algebraic computation tree* on (*S*, *NF*, *EF*) is a binary tree *T* such that each leaf vertex *v* of T is labeled by a value $L(v) \in S$; each internal vertex u of T is labeled by a function $f(u) \in NF$; and each edge e of T is labeled by a function $f(e) \in EF$. The *bottom-up algebraic tree computation* (B-ATC) problem on T is as follows.

Let u be a vertex of T whose left child v_1 and right child v_2 are leaf vertices. Let $e_1 = (v_1, u)$ and $e_2 = (v_2, u)$. Compute a value $L(u) \in S$ defined by

$$L(u) = F(u)[F(e_1)L(v_1), F(e_2)L(v_2)],$$

and delete v_1, v_2, e_1 and e_2 from T. Thus u becomes a new leaf vertex. Repeat this operation until all vertices of T receive a value in S.

In order to solve B-ATC problem in parallel, Abrahamson *et. al.* introduced the *shunt operation* [ADK89]. Let u be a vertex of T with left child v_1, right child v_2 and parent w. Let $e_1 = (v_1, u)$, $e_2 = (v_2, u)$ and $e_0 = (u, w)$. Suppose, v_1 is a leaf vertex (Figure 2.4). A shunt operation on v_1 is described as follows: delete v_1 and from T and make v_2 the left child of w. Let $e' = (v_2, w)$ be the new edge. For the B-ATC problem, assign e' a function $F(e')$ defined by –

$$F(e')(x) = F(e_0)(F(u)[F(e_1)(L(v_1)), F(e_2)(x)])$$

If the right child v_2 of u is a leaf vertex, a shunt operation performed on v_2 is defined similarly. Clearly a shunt operation does not affect the subsequent evaluation on T. The following algorithm solves the BATC problems [JOS92].

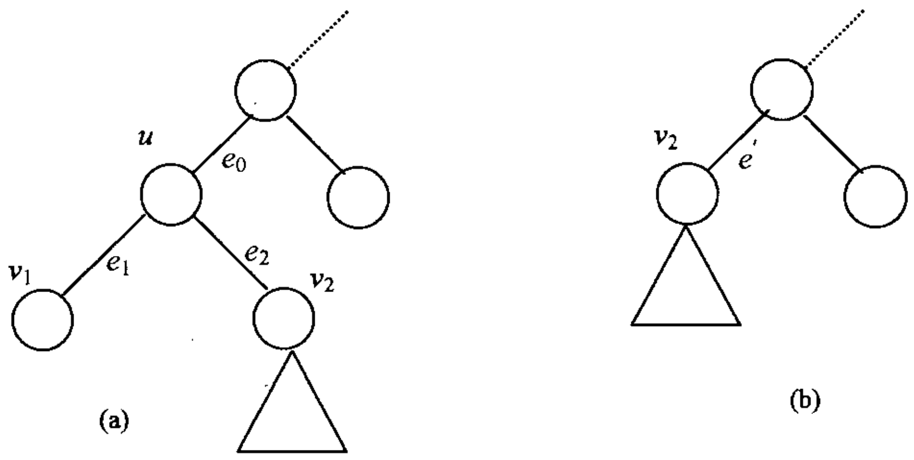

Figure 2.4: Shunt operation: (a) before (b) after.

Algorithm 2.1 *Tree Contraction*

Input: A rooted binary tree T such that each vertex has exactly two children

Output: T is connected to a three node binary tree

```
begin
  1. Number the leaf vertices of $T$ by 1,2,... from the left to the right excluding the
     leftmost and rightmost leaves and store the labeled leaves in an array $A$ of size $n$.
  2. for $\lceil \log_2(n+1) \rceil$ iterations do
  3.     Apply shunt operation concurrently to all elements of $A_{odd}$ that are left
         children
  4.     Apply shunt operation to remaining elements in $A_{odd}$
  5.     Set $A := A_{odd}$
end.
```

The following theorem was proved in [JOS92]. Details of implementation and analysis of algorithm 2.1 can be found in [JOS92].

Theorem 2.1 Algorithm 2.1 correctly contracts the input binary tree into a three node binary tree. This algorithm can be implemented on the EREW PRAM in $O(\log_2 n)$ time, using a linear number of operations, where n is number of vertices in the input tree. □

2.5 Equivalent Binary Tree of a Tree

B-ATC problem requires a regular binary tree to apply tree contraction algorithm. To solve the l-vertex-coloring and l-edge-coloring problem on trees using B-ATC, we have to convert an arbitrary rooted tree into a regular binary tree. We now show how an arbitrary rooted tree rooted at s can be reduced to a regular binary tree T_b, which is the canonical binary tree representation of T. Any arbitrary node in T can be chosen as root. Every internal node u having d children, say $y_1, y_2, \ldots, y_d$, is replaced with d+1 new nodes $u^1, u^2, \ldots, u^{d+1}$, such that u_i, $1 \leq i \leq d$ is the father of u_{i+1} and the i'th child y_i of u. In T_b, u^{i+1}, $1 \leq i \leq d$, is the right child of u^i and u^{d+1} is a leaf of the tree representing node u in T.

If vertex y is the i-th child of u in T, then y^1 is the left child of u^i in T_b. Fig. 2.5 illustrates an example.

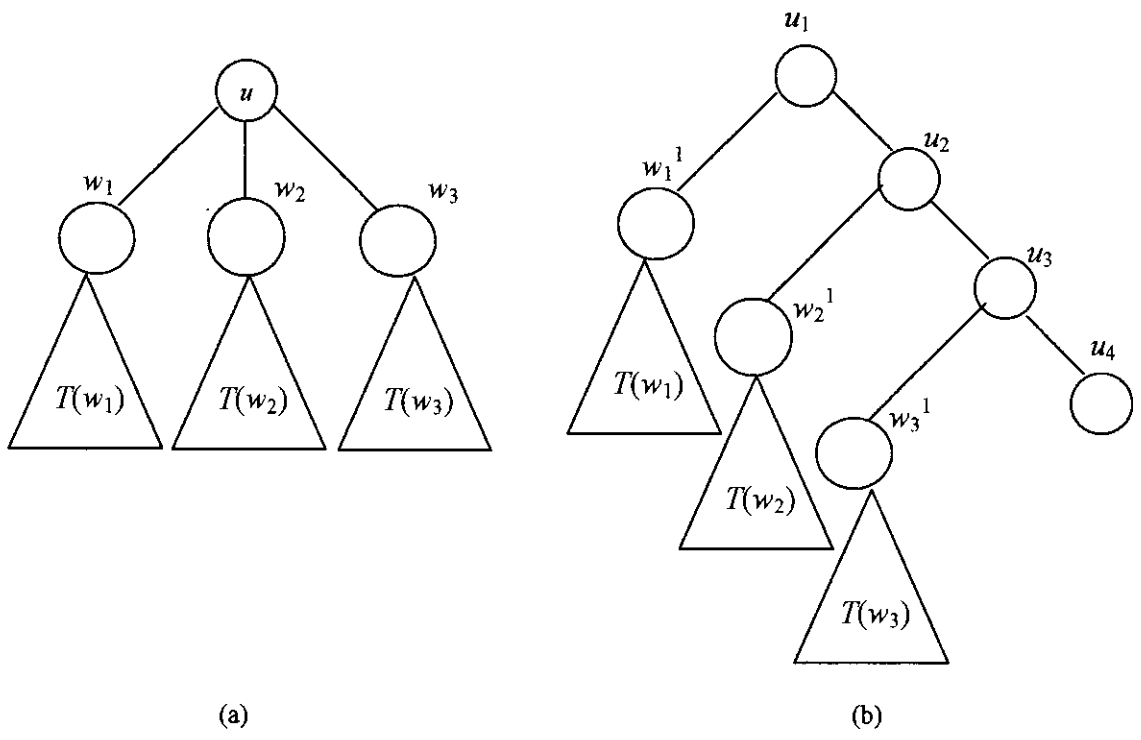

Figure 2.5: (a) A tree $T(u)$ rooted at u. (b) Equivalent binary decomposition tree of $T(u)$.

In binary tree T_b, the nodes with superscript $d + 1$ represent a node of tree T while the other nodes are dummy nodes and do not correspond to any node of tree T.

Chapter 3

Sequential Algorithm for *l*-Vertex-Coloring of Trees

This chapter deals with sequential generalized vertex coloring problem on trees. Since ordinary vertex coloring problem is NP-hard [GJ79], the *l*-vertex-coloring problem is also NP-hard in general. So it is very unlikely that there exists an efficient algorithm to solve the *l*-vertex-coloring problem for general graphs. But polynomial time algorithms have been developed for special subset of graphs – partial *k*-trees in particular. Zhou *et al.* [ZKN00] presented a polynomial time algorithm that determines whether any *l*-vertex-coloring exists for partial *k*-trees in time $O\left(n^3 + n \times (\alpha+1)^{2^{2(k+1)(l+2)+1}}\right)$, where α is the number of colors. It may be noted that partial *k*-trees are graphs of treewidth bounded by a fixed constant *k* [ZKN00]. Ordinary trees are partial 1-trees. So if we put $k = 1$ in their algorithm, the time complexity comes out to be $O\left(n^3 + n \times (\alpha)^{2^{4l+9}}\right)$. Recently Kashem *et al.* have presented an $O(\chi_l(T) \times n)$ time sequential algorithm for the *l*-vertex-coloring problem on ordinary trees [KY00], but their algorithm does not guarantee an optimal solution.

In this chapter, we present an $O\left((l+2)^{2\alpha} \times n\right)$ time sequential algorithm to determine whether any *l*-vertex-coloring with α colors exists for the given tree. If both *l* and the maximum vertex degree of the tree are bounded by constant integers, then $\chi_l(T)$ becomes a constant number [KY00], and our algorithm solves the problem in linear time. Given a particular number of colors and a tree, we have applied post order traversal technique to determine all valid *l*-vertex-coloring for the tree (if there exists any). We then used this algorithm together with binary search technique over a bounded range of positive integers to determine the optimal number of colors.

The l-vertex-coloring problem on a weighted graph $G = (V, E)$ can be easily reduced to the ordinary vertex coloring problem on a new non-weighted graph $G' = (V, E')$ such that $(u, v) \in E'$ for any two vertices u and v in V if and only if $dist(u, v) \leq l$ in G [Jag96, ZKN00]. Therefore, one may expect that the l-vertex-coloring problem for a tree T can be solved by applying a linear-time algorithm to solve an ordinary vertex-coloring problem for trees [BPT92]. However, it is not the case, because G' obtained for a tree is not always a tree.

The remainder of this chapter is organized as follows. Section 3.1 gives some preliminary definitions and easy observations. Section 3.2 gives a sequential algorithm for l-vertex-coloring of trees. Finally, Section 3.3 concludes with our sequential l-vertex-coloring algorithm.

3.1 Preliminaries

In this section we define some terms and easy observations. Let $T = (V, E)$ be a tree with vertex set V and edge set E. The number of vertices in T is denoted by n. T is a "free tree", but we regard T as a rooted tree. For the sake of convenience an arbitrary vertex of tree T is designated as the root of T. We will use notions as: root, internal vertex, child and leaf in their usual meaning. The maximum vertex degree of T is denoted by Δ. An edge joining vertices u and v is denoted by (u, v). Each of the edges has a positive weight associated with it. If, N is the set of all positive integers then we can define the weight function for the edges as w: $E \rightarrow N$. The distance from a vertex u to another vertex v, denoted by $dist(u, v)$, is the length of the path P from u to v in T. Therefore, $dist(u,v) = w(P) = \sum_{e \in P} w(e)$.

Definition 3.1 Let l be a positive integer and C be a set of colors. Then a function $\varphi : V \rightarrow C$ is an l-vertex-coloring of T if $\varphi(u) \neq \varphi(v)$ for any two vertices u and v such that $dist(u, v) \leq l$.

Definition 3.2 The minimum number of colors needed to perform an l-vertex-coloring of T is called l-chromatic number of T and is denoted by $\chi_l(T)$.

Let, the number of colors used by a vertex coloring φ of tree T is denote by $\#\varphi$. Clearly $\chi_l(T) \leq \#\varphi$ for an l-vertex-coloring of tree T. One may assume without loss of generality that φ uses the consecutive integers 1, 2, ..., $\#\varphi$ as the colors. Then C is the color set having colors 1, 2, ..., $\#\varphi$. If the maximum vertex degree of T is Δ then $\chi_l(T) \leq \frac{\Delta^{l+1}-1}{\Delta-1}$ [KY00]. If both l and Δ are bounded integers then $\chi_l(T) = O(1)$.

Definition 3.3 Let, φ be an l-vertex-coloring of T with m colors. The distance vector of a vertex $u \in V$, denoted as $D(\varphi, u)$ or simply $D(u)$, is defined as an m-tuple $\langle d_i \mid 1 \leq i \leq m, d_i \in \{0,\ldots,l,\infty\}\rangle$. Each $d_i = \min\{dist(u,v)\}$, $1 \leq i \leq m$, where v, $v \in V$ is a descendent of u in T and $\varphi(v) = i$. All $d_i > l$ are represented by ∞. Moreover, if no vertex in T has been assigned the color i, $d_i = \infty$.

Example: Let us consider 4-vertex-coloring of the following tree with 3 colors. Consider the partial coloring where some of the nodes have already been assigned colors while nodes a, b and c are yet to be colored. Edge weights are written next to the edges.

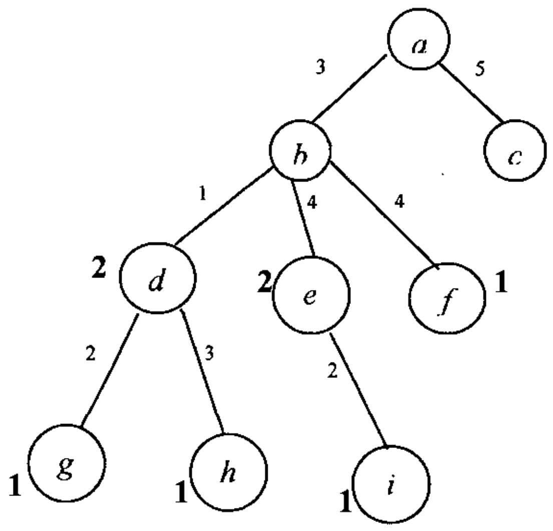

Fig 3.1: A partial 4-vertex-coloring of a tree with 3 colors

The distance vector for node *b* is *D*(*b*) = <3, 1, ∞>. Similarly, distance vector for node *d* is *D*(*d*) = <2, 0, ∞>.

Lemma 3.1 In an *l*-vertex-coloring φ of a tree *T* with *m* colors, there can be at most $(l+2)^m$ different distance vectors.

Proof: Immediate. □

These $(l+2)^m$ distance vectors can be enumerated in numerous ways. If we consider the distance vectors in the domain of $(l+2)$-base number system, we get one such enumeration. Here, $\{0,\ldots,l,\infty\}$ correspond to $(l+2)$ different digits (∞ is represented by $l+1$) and the distance vectors are numbered from 0 to $(l+2)^m-1$, based on their values in the number system.

Example: The distance vectors in 4-vertex-coloring shown in Fig 3.1 can be considered in the domain of 6 (= 4 + 2)-base number system. In that case, *D*(*b*) can be represented by decimal value 119 $(= 3\times 6^2 + 1\times 6^1 + 5\times 6^0)$. Similarly, *D*(*d*) can be represented by decimal value 77 $(= 2\times 6^2 + 0\times 6^1 + 5\times 6^0)$.

Definition 3.4 In an *l*-vertex-coloring φ with *m* colors, *combine* operation of two distance vectors $X = \langle x_i \mid 1 \le i \le m, x_i \in \{0,\ldots,l,\infty\}\rangle$ and $Y = \langle y_i \mid 1 \le i \le m, y_i \in \{0,\ldots,l,\infty\}\rangle$ where $(x_i + y_i > l,\ 1 \le i \le m)$, generates a third distance vector $R = \langle r_i \mid 1 \le i \le m, r_i \in \{0,\ldots,l,\infty\}, \text{and } r_i = \min\{x_i, y_i\}\rangle$. If the condition $x_i + y_i > l$, for any *i*, $1 \le i \le m$, is not satisfied, *X* and *Y* cannot be combined.

Example: In a 4-vertex-coloring with 3 colors, the distance vectors *X* = <2, ∞, 4> and *Y* = <3, 1, ∞> can be combined to generated a new distance vector *R* = <2, 1, 4>. But the distance vectors X = <2, ∞, 4> and Y = <1, 1, ∞> cannot be combined because $(x_1 + y_1) \leq 4$.

If we consider the distance vectors in the domain of $(l + 2)$-base number system, X, Y and R will be represented by decimal values $106(= 2 \times 6^2 + 5 \times 6^1 + 4 \times 6^0)$, $119(= 3 \times 6^2 + 1 \times 6^1 + 5 \times 6^0)$ and $82(= 2 \times 6^2 + 1 \times 6^1 + 4 \times 6^0)$ respectively. Therefore, combination of distance vectors X=106 and Y=119 produces the distance vector R=82. We can pre-compute these combination results and store them in a $(l+2)^m \times (l+2)^m$ two-dimensional matrix *CM*. Any invalid combination result can be represented by some special value not in the range $\left(0 \ldots \left((l+2)^m - 1\right)\right)$ (-1 for example). We then have the following lemma.

Lemma 3.2 Combination of two distance vectors can be done in $O(1)$time.

Proof: Immediate. □

Definition 3.5 In an l-vertex-coloring φ with m colors, *addition* of a weight w to a distance vector $X = \langle x_i \mid 1 \leq i \leq m, x_i \in \{0, \ldots, l, \infty\} \rangle$ produces a new distance vector $R = \langle r_i \mid 1 \leq i \leq m, r_i \in \{0, \ldots, l, \infty\} \rangle$ as follows

$$r_i = \begin{cases} x_i + w, \text{if } (x_i + w) \leq l; \text{and} \\ \infty, \quad \text{otherwise.} \end{cases}$$

Example: In a 4-vertex-coloring with 3 colors, if we add weight 2 to the distance vector X = <2, ∞, 4>, we get a new distance vector R = <4, ∞, ∞>.

As before, if we consider the distance vectors in the domain of $(l + 2)$-base number system, X and R will be represented by decimal values $106(= 2 \times 6^2 + 5 \times 6^1 + 4 \times 6^0)$ and $179(= 4 \times 6^2 + 5 \times 6^1 + 5 \times 6^0)$ respectively. Therefore, addition of weight 2 to distance vector X=106 produces the distance vector R=179. We can pre-compute these addition results and store them in a $(l+2)^m \times (l+1)$two-dimensional matrix AM. For all $w > l$ the resultant distance vector $\langle d_i \mid 1 \le i \le m, d_i = \infty \rangle$ is directly computed without consulting the matrix AM. We then have the following lemma.

Lemma 3.3 Addition of a weight to a distance vector can be done in $O(1)$time.

Proof: Immediate. □

Our algorithm tries to assign each possible color to a particular node. So, each node will be associated with multiple distance vectors. In the worst case, at most $(l+2)^m$distance vectors will be generated for any particular node. Using the enumeration explained above, we can represent the distance vectors using a one-dimensional bitmap of length $(l+2)^m$. Each bit in the bitmap corresponds to a particular distance vector. If the bit is 1 the distance vector has been generated for the node. Otherwise, the distance vector is not included in the list.

Using the bitmap representation and Lemma 3.2 we get the following lemma.

Lemma 3.4 Two lists of distance vectors obtained from the left child and right child of a node $u \in T$ can be combined at u in $O\left((l+2)^{2m}\right)$ time.

Proof: Each of the two lists can contain at most $(l+2)^m$distance vectors. Each distance vector of one list has to be checked against each distance vector of another list. Therefore, at most $(l+2)^m \times (l+2)^m$ pair of distance vectors have to be combined. □

By Lemma 3.3 we get the following lemma.

Lemma 3.5 The list of distance vectors of any particular node can be updated with an edge weight in $O\left((l+2)^m\right)$ time.

Proof: Immediate. □

3.2 An Efficient Algorithm

The main result of this section is the following theorem.

Theorem 3.6 For any positive integer l, an optimal l-vertex-coloring of a tree having n vertices can be found in time $O(\log\alpha \times (l+2)^{2\alpha} \times n)$, where $\alpha = \dfrac{\Delta^{l+1}-1}{\Delta-1}$.

If both l and maximum vertex degree Δ of the tree T are bounded integers, then by theorem 3.6 we have the following corollary.

Corollary 3.7 An optimal l-vertex-coloring of a tree T with bounded degrees can be found in linear time for any bounded integer l.

Remainder of this section gives an algorithm to solve the *l-vertex-coloring problem* on trees in time $O(\log\alpha \times (l+2)^{2\alpha} \times n)$, where $\alpha = \dfrac{\Delta^{l+1}-1}{\Delta-1}$. Given m number of colors, the algorithm assigns colors to nodes in post order fashion to find an l-vertex-coloring of T with m colors. At first, the input tree T is transformed into an equivalent binary tree T_b as explained in Section 2.5. Since we are dealing with weighted trees, we need to consider assignment of weights to the edges of the binary tree T_b in such a way that $dist(u,v)$ for all $u, v \in T$ is equal to $dist(x,y)$ for all $x, y \in T_b$, where x and y in T_b represents u and v in T respectively. This can be achieved by the following weight function w_b.

Let x and y be two nodes in T_b obtained from nodes u and v in T respectively. If $(x,y) \in E_{T_b}$ then

$$w_b(x,y) = \begin{cases} w(u,v) & \text{if } u \neq v \\ 0 & \text{if } u = v \end{cases}$$

Example:

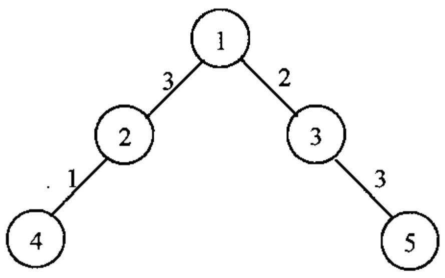

Figure 3.2: Original Tree T.

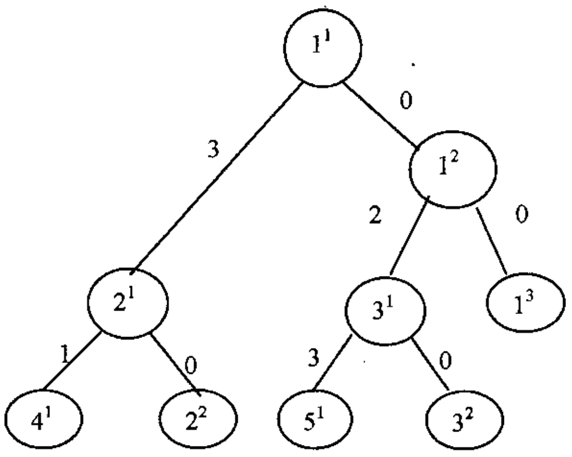

Figure 3.3: Binary decomposition tree T_b for the tree T in Fig-3.2.

Therefore, we have the following lemma [BH95].

Lemma 3.8 The binary equivalent tree T_b of any arbitrary weighted tree T can be obtained in linear time. □

First, we will present an algorithm *l_vertex_color* that finds an l-vertex-coloring of T_b with a given set of colors $[1, \ldots, m]$. Later we will give another algorithm *optimal_l_vertex_color* that uses *l_vertex_color* and binary search technique to determine the optimal number of colors for T_b.

The algorithm *l_vertex_coloring* applies algorithm *node_function* to each node in T_b in post order fashion. At each leaf node, *l_vertex_coloring* initializes the list of distance vectors of that leaf while at each internal node it combines the lists obtained from its left subtree and right subtree usign *node_function*. The algorithm also applies *edge_function* to each edge during backtracking along the edge. The algorithm terminates when all the nodes have been completely visited. If the list of distance vectors at root node contains no distance vector then it is not possible to obtain an l-vertex-coloring of T_b with m colors. Otherwise, each existing distance vector at root represents a different l-vertex-coloring of the given tree using m colors. We now present the algorithms *l_vertex_coloring*, *node_function* and *edge_function*.

Algorithm 3.1 *l-vertex-coloring*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, …, m]

Output: All possible l-vertex-coloring of subtree rooted at v with m colors (if there exists any)

```
begin
    1. if $v$ is not a leaf node then
    2.     l_vertex_coloring(left($v$),$m$);
    3.     l_vertex_coloring(right($v$),$m$);
    4.     node_function($v$,$m$);
    5. else
    6.     Create a new list of distance vectors $D_v$ containing no vector, for this leaf node
    7.     for all integers $i$, $0 \le i \le m-1$ do
    8.         $p = (l+2)^m - 1$;          // $\langle d_i \mid 1 \le i \le m, d_i = \infty \rangle$
    9.         set_digit($p$, 0, $i$);      // algorithm 3.6
    10.        $D_v[p] = 1$;
    11. end if
    12. edge_function($v$, $m$);
end.
```

Algorithm 3.2 *node_function*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, …, m]

Output: Updated bitmap of list of distance vectors of node v

```
begin
    1. $D_{left}$ = bitmap(left($v$));
    2. $D_{right}$ = bitmap(right($v$));
    3. Create two new lists $D_v$ and $D_f$ containing no distance vector for this node;
    4. for all integers $i$, $0 \le i \le (l+2)^m - 1$ do
    5.     for all integers $j$, $0 \le j \le (l+2)^m - 1$ do
    6.         if $D_{left}[i]$ and $D_{right}[j]$ are 1 and $CM[i,j]$ is not invalid then
    7.             $D_v[CM[i,j]] = 1$;
```

```
8.          end if
end.
```

Algorithm 3.3 *edge_function*(*v*, *m*)

Input: A node v in the binary decomposition tree T_b, a color set $[1, \ldots, m]$

Output: Updated bitmap of list of distance vectors of node v

```
begin
    1. $d = weight(<v, \text{parent}(v)>)$;
    2. Create a new list of distance vectors $D_{fe}$ containing no vectors;
    3. for all integers $j$, $0 \le j \le (l+2)^m - 1$ do
    4.          if $D_f[j]$ is 1 then
    5.              $D_{fe}[AM[j, d]] = 1$;
    6.          end if
end.
```

We now present the *optimal_l_vertex_coloring* algorithm (Algorithm 3.4) that uses *l_vertex_coloring* to solve the optimal l-vertex-coloring on trees. Algorithm 3.4 first transforms tree T into equivalent binary tree T_b. It then determines the upper bound of the number of colors (α) and computes matrices CM and AM. The algorithm then performs a binary search in the bounded domain of colors to determine the optimal number of colors.

Algorithm 3.4 *optimal_l_vertex_coloring*(*T*)

Input: An arbitrary tree T

Output: An optimal l-vertex-coloring of T

```
begin
1. Construct Binary Decomposition Tree $T_b$ from $T$
2. Determine the upper bound for number of colors needed ($\alpha$)
3. Compute the matrices $CM$, $AM$
4. $l = 1$          // lower bound
5. $u = \alpha$          // upper bound
6. loop until the optimum solution is found
```

```
7.        $i = (l + u)/2$
8.        $l\_vertex\_coloring(\text{root}(T_b), i)$
9.        adjust the color range depending on the outcome of previous step
10.       perform next iteration in the updated range
11.  if $j$, $1 \le j \le \alpha - 1$ is an integer such that a coloring with $j + 1$ colors exists but there is
        no coloring with $j$ colors then
12.       $j$ is the optimal number of colors
13.  else
14.       $\alpha$ is the optimal number of colors
end.
```

By Lemma 3.5 *edge_function* takes $O\left((l+2)^m\right)$time, where m is the number of colors. Line 4 to 8 of *node_function* combines the two distance vectors obtained from left and right child and this takes $O\left((l+2)^{2m}\right)$ time (Lemma 3.4). So, overall complexity of *node_function* is $O\left((l+2)^{2m}\right)$ where m is the number of colors. Algorithm *l_vertex_coloring* simply performs a post order traversal of the input tree and calls *node_function* and *edge_function* for each internal node. However, if the node being processed is a leaf node, *l_vertex_coloring* constructs the initial list of distance vectors instead of calling *node_function*. Line 4 (for internal node) of algorithm 3.1 can be done in time $O\left((l+2)^{2m}\right)$ (Lemma 3.4) while Lines 6 to 10 takes time $O(m)$. Therefore, each single node can be processed in time $O\left((l+2)^{2m}\right)$. The edge function can be computed in time $O\left((l+2)^m\right)$ (Lemma 3.5). So, the algorithm *l_vertex_coloring* runs in $O\left(\left((l+2)^{2m} + (l+2)^m\right) \times n\right) = O\left((l+2)^{2m} \times n\right)$ time.

The only part yet to be specified is how to compute the matrices *CM, AM.* Following algorithms describe the mechanism of computing these matrices. Algorithms 3.5 and 3.6 are supporting functions for subsequent matrix computing algorithms. These supporting functions consider the input number as a value in (l +2)-base number system.

Algorithm 3.5 *get_digit(number, pos)*

Input: A number and the position of the digit to be extracted

Output: The digit in (l + 2) base number system, at specified position

```
begin
1. $d = \frac{number}{(l+2)^{pos-1}}$;        // integer division
2. $r = MOD(d, l+2)$;   // modulus operation
3. return $r$;
end
```

Algorithm 3.6 *set_digit(number, digit, pos)*

Input: A number, a digit and the position of the digit

Output: Updated number with input digit placed at specified position

```
begin
1. $d = get\_digit(number, pos)$;
2. $number = number - d \times (l+2)^{pos-1}$;  // clear old digit at specified position
3. $val = digit \times (l+2)^{pos-1}$;
4. $number = number + val$;              // set new digit
end
```

Algorithm 3.7 *compute_CM*()

Input: None

Output: Initialized matrix *CM*

```
begin
1. for all integers $r$, $0 \le r \le (l+2)^{\alpha} - 1$ do        // loop 1
2.     for all integers $c$, $0 \le c \le (l+2)^{\alpha} - 1$ do    // loop 2
3.         $CM(r, c) = -1$;              // mark as invalid combination
4.         $target\_vector = 0$;
5.         for all integers $p$, $1 \le p \le \alpha$ do          // loop 3
6.             $d1 = get\_digit(r, p)$;
```

```
7.            $d2 = get\_digit(c, p)$;
8.            **if** $d1 + d2 \le l$ **then**
9.               $target\_vector = -1$
10.              **exit** loop 3 and **start next iteration** of loop 2;
11.           **else**
12.              $set\_digit(target\_vector, \min\{d1, d2\}, p)$;
13.        **end** loop 3
14.        $CM(r, c) = target\_vector$;
15.    **end** loop 2
16. **end** loop 1
**end**
```

Algorithm 3.8 *compute_AM*()

Input: None

Output: Initialized matrix *AM*

```
**begin**
1.  **for** all integers $r$, $0 \le r \le (l+2)^{\alpha} - 1$ **do**        // **loop 1**
2.     **for** all integers $c$, $0 \le c \le l$ **do**     // **loop 2**
3.        $AM(r, c) = -1$;              // mark as invalid combination
4.        $target\_vector = 0$;
5.        **for** all integers $p$, $1 \le p \le \alpha$ **do**          // **loop 3**
6.           $d = get\_digit(r, p)$;
7.           **if** $d + c \le l$ **then**
8.              $set\_digit(target\_vector, d + c, p)$;
9.           **else**
10.             $set\_digit(target\_vector, l + 1, p)$;
11.       **end** loop 3
12.       $AM(r, c) = target\_vector$;
13.    **end** loop 2
14. **end** loop 1
**end**
```

Having specified all relevant steps, we can now determine the complexity of algorithm *optimal_l_vertex_coloring* (algorithm 3.4). Line 1 of algorithm 3.4 can be done in linear time (Lemma 3.8). Line 3 can be done in $O\left((l+2)^{2\alpha} \times \alpha\right)$. Line 2, 4 and 5 take $O(1)$ time. The loop from line 6 to line 10 iterates at most $\log_2 \alpha$ times. At each iteration, line 8 takes $O\left((l+2)^{2i} \times n\right) \approx O\left((l+2)^{2\alpha} \times n\right)$ time. So, the overall time complexity of *optimal_l_vertex_coloring* is $O\left(\log\alpha \times (l+2)^{2\alpha} \times n\right)$, where $\alpha = \frac{\Delta^{l+1} - 1}{\Delta - 1}$. This proves Theorem 3.6. □

3.3 Conclusion

In this chapter, we have given a sequential algorithm to solve the *l-vertex-coloring problem* on trees. Our algorithm runs in time $O\left(\log\alpha \times (l+2)^{2\alpha} \times n\right)$, where $\alpha = \frac{\Delta^{l+1} - 1}{\Delta - 1}$. If both Δ and l are bounded integers, then our algorithm runs in linear time. Note that ordinary vertex coloring problem is a special case of l-vertex-coloring problem with $l = 1$. Our algorithm can solve the ordinary vertex-coloring problem in linear time. This is the first known direct solution to *l-vertex-coloring problem* on ordinary trees that guarantees an optimal solution.

Chapter 4

Parallel Algorithm for *l*-Vertex-Coloring of Trees

This chapter deals with parallel generalized vertex coloring problem on trees. Basically this is the parallel version of our sequential algorithm presented in Chapter 3. In this chapter, we present an $O(\log_2 n)$ time parallel algorithm for solving the *l*-vertex-coloring problem on trees. Given a particular number of colors and a tree, we will apply Bottom-up Algebraic Tree Computation technique (B-ATC) to determine a valid *l*-vertex-coloring for the tree (if there exists any). As specified in Section 3.1, we can determine the upper bound of the number of colors and we have to search for the optimal number of color in this bounded range. If the upper bound is k for some integer $k > 0$, we will use k disjoint sets of processors and assign color i, $1 \le i \le k$ to set i, $1 \le i \le k$. Each set of processors then tries to find all *l*-vertex-colorings with the number of colors assigned to that set. The optimal number of colors will be c, $1 \le c \le k-1$ where set c has found a valid *l*-vertex-coloring but set $c-1$ has not. If there is no such c then k is the optimal number of colors.

The remainder of this chapter is organized as follows. Section 4.1 gives some preliminary definitions and easy observations. Section 4.2 gives a parallel algorithm for *l*-vertex-coloring of trees. Finally, Section 4.3 concludes with our parallel *l*-vertex-coloring algorithm.

4.1 Preliminaries

We assume that the tree being colored is a binary decomposition tree. Any arbitrary tree with n vertices can be transformed into equivalent binary decomposition tree in $O(\log_2 n)$ time using $O(n)$ operations on the EREW PRAM model [BH95]. The same weight assignment technique explained in Section 3.2 will be applied here. However the operations will be done in parallel. We will use the same representation of distance vectors as explained in Section 3.1. Therefore the algorithms for computing node functions and edge functions basically remain the same. However, these algorithms will be executed in parallel. So the algorithms presented in Section 3.2 need to be modified. Besides, we also need to carefully assign the operations to different processors so that they can be done in parallel. The modified algorithms are presented below.

Algorithm 4.1 *node_function_parallel*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, ..., m]

Output: Updated bitmap of list of distance vectors of node v

```
begin
    1. $D_{left}$ = bitmap(left($v$));
    2. $D_{right}$ = bitmap(right($v$));
    3. Create two new lists $D_v$ and $D_f$ containing no vector for this node;
    4. for all integers $i$, $0 \le i \le (l+2)^m - 1$ in parallel do
    5.     for all integers $j$, $0 \le j \le (l+2)^m - 1$ in parallel do
    6.         if $D_{left}[i]$ and $D_{right}[j]$ are 1 and $CM[i,j]$ is not invalid
    7.             $D_v[CM[i,j]] = 1$;
    8.         end if
end.
```

Line 4-8 of algorithm 4.1 combines two lists of distance vectors obtained from its children in parallel. Each combination is assigned to a different processor, so the number of required processors and number of operations will be $O\left((l+2)^{2m}\right)$. The combination operation at each processor can be done in $O(1)$ time (Lemma 3.2). Each processor will read a unique element of *CM* matrix but multiple processors may write to the same location in resultant distance vector list.

Algorithm 4.2 *edge_function_parallel*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, …, m]

Output: Updated bitmap of list of distance vectors of node v

```
begin
    1. $d = weight(<v, \text{parent}(v)>)$;
    2. Create a new list of distance vector $D_{fe}$ that contains no vector;
    3. for all integers $j$, $0 \le j \le (l+2)^m - 1$ in parallel do
    4.         if $D_v[j]$ is 1 then
    5.             $D_{fe}[AM[j, d]] = 1$;
    6.         end if
end.
```

Therefore, we should use common ERCW PRAM model. However, since the number of processors that can write to a particular location is bounded (for bounded l and m), we can use EREW PRAM model. The edge function given in algorithm 4.2 deals with $(l+2)^m$ distance vectors and each addition operation can be done in $O(1)$ time (Lemma 3.3). So this function can be implemented using $O((l+2)^m)$ processors in EREW PRAM model. The number of operations in this stage is $O((l+2)^m)$.

Therefore, we have the following lemmas.

Lemma 4.1 The node function *node_function_parallel* can be computed in $O(1)$ time and $O((l+2)^{2m})$ operations using $O((l+2)^{2m})$ processors on the EREW PRAM model, where m is the number of colors. □

Lemma 4.2 The edge function *edge_function_parallel* can be computed in $O(1)$ time and $O((l+2)^m)$ operations using $O((l+2)^m)$ processors on the EREW PRAM model, where m is the number of colors. □

4.2 An Efficient Parallel Algorithm

We now give an efficient parallel algorithm to solve the *l-vertex-coloring problem.* Algorithms 4.1 and 4.2 are applied for each internal node and each edge respectively. However, unlike our sequential algorithm presented in Chapter 3, multiple nodes and edges will be processed in parallel. As mentioned before, we will apply B-ATC to solve the problem. Given a particular number of colors, algorithm 4.3 determines whether there exists a valid l-vertex-coloring of the given tree with that many colors. We can use algorithm 4.3 to determine the optimal l-vertex-coloring in two different ways. Let, α is the upper bound for the number of colors. We can use binary search technique in the range $[1, \ldots, \alpha]$ and apply algorithm 4.3 at each stage. This strategy requires $O(\log_2 \alpha \times \log_2 n)$ time and $O((l+2)^{2\alpha} \times n)$ processors. Alternately, we can use α disjoint sets of processors. We use one different set of processors for applying algorithm 4.3 with each color in the range $[1, \ldots, \alpha]$. In this case, the *l-vertex-coloring problem* can be solved in $O(\log_2 n)$ time using $O(\alpha \times (l+2)^{2\alpha} \times n)$ processors. Algorithms 4.3 and 4.4 implement the second strategy.

The main result of this section is the following theorem.

Theorem 4.3 An optimal l-vertex-coloring of a tree T can be found in $O(\log_2 n)$ time using $O(\alpha \times (l+2)^{2\alpha} \times n)$ processors on the EREW PRAM model, where n is the number of vertices in T and α is the upper bound of the number of colors.

```
Algorithm 4.3 l_vertex_coloring_parallel(T, m)
Input: A binary decomposition tree T and a color set [1, ..., m]
Output: All l-vertex-coloring of T with m colors (if there exists any)
begin
    1. For all integers $i$, $1 \le i \le m$, initialize the list of distance vectors of each leaf so
       that the list contains all distance vectors of the form –
       $\langle d_1, d_2, \ldots d_{i-1}, 0, d_{i+1}, \ldots, d_m \mid d_j = \infty \text{ where } 1 \le j \le m \text{ and } j \ne i \rangle$;
    2. Initialize the list of distance vectors of each internal vertex to be an empty list;
    3. Number the leaf vertices of T by 1, 2, ... from the left to the right excluding
       the leftmost and rightmost leaves;
    4. for $\lceil \log_2(n+1) \rceil$ iterations do
    5.     Apply shunt operation concurrently to all odd numbered leaves that are
           left children;
    6.     Apply shunt operation to remaining odd numbered leaves;
    7.     Remove odd numbered nodes from consideration and sequentially
           renumber all even numbered leaves;
end
```

```
Algorithm 4.4 optimal_l_vertex_coloring_parallel(T)
Input: A tree T to be colored
Output: Optimal l-vertex-coloring of T
begin
  1. Construct the binary decomposition tree $T_b$ of T;
  2. $k \leftarrow$ upperbound($T_b$);
  3. Compute the matrices CM, AM;
  4. for all integers $j$, $1 \le j \le k$ in parallel do
  5.     l_vertex_color_parallel($T_b$, $j$);
  6. for all integers $j$, $1 \le j \le k$ in parallel do
  7.     if a coloring with $j+1$ colors exists but there is no coloring with $j$ colors then
  8.         $j$ is the optimal number of colors;
end
```

As mentioned in Section 2.4, B-ATC is defined on a triple (*S*, *NF*, *EF*) consisting of a set *S*, a *vertex function set* $NF \subseteq \{f \mid f : S \times S \to S\}$ and an *edge function set* $EF \subseteq \{f \mid f : S \to S\}$. For our *l*-vertex-coloring problem, the set *S* consists of lists of distance vectors. Initially the list in each leaf node contains all distance vectors representing a single color assignment. Each internal vertex contains an empty list of distance vectors. As the B-ATC algorithm proceeds, these lists are gradually filled up. For this problem, both *NF* and *EF* have a single element – the node function and the edge function described in algorithms 4.1 and 4.2 respectively. For convenience of description, we will use the notations $F_{node}(v)$ and $F_{edge}(e)$ for the node function and edge function respectively. The shunt operation has been introduced in Section 2.4. Let *u* be a vertex of *T* with left child v_1, right child v_2 and parent *w*. Let $e_1 = (v_1, u)$, $e_2 = (v_2, u)$ and $e_0 = (u, w)$ and v_1 is a leaf vertex. The shunt operation defined in Section 2.4 modifies the edge function corresponding to the edge e_0. We will use a modified version of the shunt operation where we modify the parameters of the node function associated with node v_2 while keeping the edge function associated with e_0 as it is. The new node function for v_2 will be–

$$F_{node}(D_{left}, x') = F_{node}(F_{edge}(e_1)(D(v_1)), F_{edge}(e_2)(x))$$

Here, $D(v_1)$ is the list of distance vector of node v_1. The first parameter D_{left} can be computed in $O(1)$ time (Lemma 4.2). If v_2 is a leaf node, then *x* will be the list of distance vectors of node v_2 and the second parameter x' can also be compute in $O(1)$ time. Therefore, $F_{node}(D_{left}, x')$ can be readily evaluated to produce the list of distance vector of node *u* and this step also takes $O(1)$ time (Lemma 4.1). However, if v_2 is not a leaf node, evaluation of the second parameter x' and hence the node function itself has to be postponed until the value of x' is available. In this case, the shunt operation simply updates the parameters previously associated with v_2 and this operation takes $O(1)$ time. So, the shunt operation can be carried out in $O(1)$ time.

Now we are ready to prove theorem 4.3. Since the tree has *n* vertices, lines 1-3 in algorithm 4.3 can be done in $O(1)$ time using $O(n)$ operations on the EREW PRAM model. Lines 4-7 iterate $\lceil \log_2(n+1) \rceil$ times. At each iteration, at most $O(n/2)$ shunt operations are performed in parallel in lines 5 and 6. These steps take $O(1)$ time and $O(((l+2)^{2m} + (l+2)^m \times m) \times n_i)$ operations on EREW PRAM model (Lemma 4.1 and 4.2) where n_i is the number of shunt operations performed in i'th iteration. The number of processors needed at this stage is $O((l+2)^{2m} \times n_i)$. Line 7 can be carried out in $O(1)$ time and $O(n)$ operations using $O(n)$ processors on EREW PRAM model. So lines 4-7 takes $O(\log_2 n)$ time and $O((l+2)^{2m} \times \sum_{1 \le i \le \log_2 n} n_i) = O((l+2)^{2m} \times n)$ operations on EREW PRAM model. The last step follows from the observation that each node function is visited at most twice – first to change the parameter and then to evaluate it finally. The number of processors needed is $O((l+2)^{2m} \times n)$. So, we have the following lemma.

Lemma 4.4 Given m number of colors, algorithm 4.3 can find all l-vertex-coloring of a tree T in $O(\log_2 n)$ time and $O((l+2)^{2m} \times n)$ operations using $O((l+2)^{2m} \times n)$ processors in EREW PRAM model. □

Algorithm 4.4 starts with constructing the binary decomposition tree T_b of the given input tree T. Since the tree T has n vertices, line 1 in algorithm 4.4 can be done in $O(\log_2 n)$ time using $O(n)$ operations on the EREW PRAM model [BH95]. Lines 2 and 3 can be done in $O(1)$ time using $O((l+2)^{2\alpha})$ processors, where α is the upper bound of the number of colors. Lines 4 and 5 apply algorithm 4.3 for each color in the range $[1, \ldots, \alpha]$ in parallel. According to Lemma 4.4, this step can be carried out in $O(\log_2 n)$ time and $O(\alpha \times (l+2)^{2\alpha} \times n)$ operations using $O(\alpha \times (l+2)^{2\alpha} \times n)$ processors. Each set of processor can be implemented on the EREW PRAM model. There will be α such sets working in parallel and processors from different sets may read the same entry from *CM* or *AM* matrix. Therefore, algorithm 4.4 should be implemented on the CREW PRAM model. However, for if both l and maximum degree of tree T are bounded, α becomes a constant number. So, algorithm 4.4 can be implemented on the EREW PRAM model. This proves theorem 4.3. □

4.3 Conclusion

In this chapter we have presented a parallel algorithm to solve the *l-vertex-coloring problem* on trees. We have used B-ATC that uses tree contraction algorithm to solve the problem. Our algorithm solves the *l-vertex-coloring problem* on trees in $O(\log_2 n)$ time using $O\left(\alpha \times (l+2)^{2\alpha} \times n\right)$ operations on the EREW PRAM model, where n is the number of vertices in T and α is the upper bound of the number of colors. Ours is the first known parallel algorithm that solves the *l-vertex-coloring problem* on trees.

Chapter 5

Optimal *l*-Edge-Coloring of Trees

Like the generalized vertex-coloring problem, generalized edge-coloring problem on general graphs is NP-hard. However, it is possible to develop polynomial time algorithms for specialized graphs such as trees. In this chapter, we present efficient algorithms, both sequential and parallel, to solve the *l*-edge-coloring problem on trees. The sequential algorithm solves the problem in time $O(\log\alpha \times (l+2)^{2\alpha} \times n)$, where n is the number of nodes in the tree T and α is the upper bound of the number of colors. Our parallel algorithm solves the problem in $O(\log_2 n)$ time using $O(\alpha \times (l+2)^{2\alpha} \times n)$ operations on the EREW PRAM model, where n is the number of vertices in T and α is the upper bound of the number of colors. We extensively make use of the concepts of Chapter 3 and 4 in this chapter.

The rest of the chapter is organized as follows. Section 5.1 discusses techniques to transform the *l*-edge-coloring problem into *l-vertex-coloring problem* on trees. Section 5.2 gives a sequential algorithm to solve the *l*-edge-coloring problem. In Section 5.3 we give a parallel algorithm for the same problem. Section 5.4 describes a technique to obtain the actual colors assigned to the edges. Finally Section 5.5 draws the conclusion.

5.1 Transforming *l*-edge-coloring problem into *l*-vertex-coloring problem

Each node in a tree has a single parent and therefore is connected by a unique edge to its parent. This key observation leads to a simple mechanism to transform the *l*-edge-coloring problem into *l*-vertex-coloring problem on a tree. Each edge $e = \langle u, v \rangle$, where u is the parent of v, can be uniquely associated with the vertex v. In this case, assigning a color to edge e is equivalent to assigning a color to vertex v. Therefore, we can apply the optimal l-vertex-coloring algorithms presented in Chapters 3 and 4 to solve the l-edge-coloring problems. However, the basic operations like combining two distance vectors and adding the edge weight to distance vectors need to be modified.

Let us consider the tree in Fig. 5.1. Suppose, nodes v_1 and v_2 have been assigned colors 1 and 2 respectively. In case of l-vertex-coloring, the distance of colors 1 and 2 from node u were w_1 and w_2 respectively. However, in l-edge-coloring nodes v_1, v_2 and u represent edges $\langle u, v_1 \rangle$, $\langle u, v_2 \rangle$ and $\langle r, u \rangle$ respectively and the distance among these three edges is 0. Therefore, colors 1 or 2 cannot be assigned to node u (that is to edge $\langle r, u \rangle$).

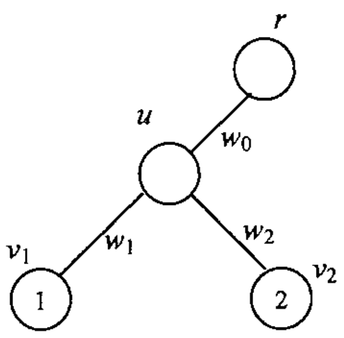

Figure 5.1: A partially colored tree.

If r represents the edge between its parent and itself, then r can be assigned color 1 (or 2) provided $w_1 \geq l$ (or $w_2 \geq l$). This suggests that our previous edge function needs to be modified so that edge weight is not added to the i'th element of a distance vector where i is the color assigned to the vertex corresponding to the edge being considered. Moreover, in l-vertex-coloring algorithm we could assign same color to vertices v_1 and v_2 if $w_1 + w_2 \geq l$. In l-edge-coloring algorithm, vertices v_1 and v_2 represent edges $\langle u, v_1 \rangle$, $\langle u, v_2 \rangle$ respectively and these two edges are adjacent to each other. So we cannot assign the same color to vertices v_1 and v_2 in l-edge-coloring algorithm even if $w_1 + w_2 \geq l$. Therefore, if two distance vectors from vertices v_1 and v_2 are generated due to same color assignment to both the nodes, they cannot be combined at vertex u. Now we give the modified matrix construction algorithms for generalized edge-coloring problem.

Algorithm 5.1 *compute_CM_e*()

Input: None

Output: Initialized matrix *CM_e*

```
begin
1. for all integers $r$, $0 \le r \le (l+2)^{\alpha} - 1$ do        // loop 1
2.    for all integers $c$, $0 \le c \le (l+2)^{\alpha} - 1$ do    // loop 2
3.       $CM\_e(r, c) = -1$;              // mark as invalid combination
4.       $target\_vector = 0$;
5.       for all integers $p$, $1 \le p \le \alpha$ do           // loop 3
6.          $d1 = get\_digit(r, p)$;
7.          $d2 = get\_digit(c, p)$;
8.          if $d_1 + d_2 \le l$ then
9.             $target\_vector = -1$
10.            exit loop 3 and start next iteration of loop 2;
11.         else
12.            $set\_digit(target\_vector, \min(d1, d2), p)$;
13.      end loop 3
14.      $CM\_e(r, c) = target\_vector$;
15.   end loop 2
16. end loop 1
end
```

Algorithm 5.2 *compute_AM_e*()

Input: None

Output: Initialized matrix *AM_e*

```
begin
1. for all integers $r$, $0 \le r \le (l+2)^{\alpha} - 1$ do        // loop 1
2.    for all integers $c$, $0 \le c \le l$ do    // loop 2
3.       $AM\_e(r, c) = -1$              // mark as invalid combination
4.       $target\_vector = 0$
```

```
5.          for all integers $p$, $1 \le p \le \alpha$ do          // loop 3
6.              $d = get\_digit(r, p)$
7.              if $d$ is 0 then                            // color $p$ has been assigned to this edge
8.                  $set\_digit(target\_vector, d, p)$      // don't add the edge weight to this color
                                                            // distance
9.              else if $d + c \le l$ then
10.                 $set\_digit(target\_vector, d + c, p)$
11.             else
12.                 $set\_digit(target\_vector, l + 1, p)$
13.         end loop 3
14.         $AM\_e(r, c) = target\_vector$
15.     end loop 2
16. end loop 1
end
```

Having modified these matrix computation algorithms we are now ready to present our *l*-edge-coloring algorithms.

5.2 A sequential algorithm for optimal *l*-edge-coloring of trees

In this section we specify an efficient sequential algorithm to solve the *l*-edge-coloring problem on trees. The node function and the edge function basically remain same as specified in Chapter 3 (algorithms 3.2 and 3.3 respectively). The only difference is that these algorithms will now use the pre-computed matrices *CM_e* and *AM_e* (Section 5.1) instead of *CM* and *AM* respectively. The *l_edge_coloring* and the *optimal_l_edge_coloring* algorithms are basically similar to *l_vertex_coloring* (algorithm 3.1) and *optimal_l_vertex_coloring* (algorithm 3.4) algorithms respectively. We can re-specify the algorithms as follows.

Algorithm 5.3 *l_edge_coloring*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, ..., m]

Output: All possible l-edge-coloring of subtree rooted at v with m colors (if there exists any)

```
begin
  1. if v is an internal node then
  2.       l_edge_coloring(left(v),m)
  3.       l_edge_coloring(right(v),m)
  4.       node_function_edge(v,m)
  5. else if v represents root(T) then
  6.     create a list of distance vector $D_v$ containing a single distance vector
         $\langle d_i, 1 \le i \le m$ and $d_i = \infty \rangle$
  7. else
  8.     Create a new list of distance vectors $D_v$ containing no vector, for this leaf node
  9.     for all integers i, $0 \le i \le m$-1 do
 10.         $p = (l+2)^m - 1$;          // $\langle d_i \mid 1 \le i \le m, d_i = \infty \rangle$
 11.         set_digit(p, 0, i);        // algorithm 3.6
 12.         $D_v[p] = 1$;
 13. end if
 14. edge_function_edge(v,m)
end.
```

Algorithm 5.4 *node_function_edge_coloring*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, ..., m]

Output: Updated bitmap of list of distance vectors of node v

```
begin
  1. $D_{left}$ = bitmap(left(v));
  2. $D_{right}$ = bitmap(right(v));
  3. Create two new lists $D_v$ and $D_f$ containing no vector for this node;
  4. for all integers i, $1 \le i \le (l+2)^m - 1$ do
```

```
5.      for all integers $j$, $0 \le j \le (l+2)^m - 1$ do
6.          if $D_{left}[i]$ and $D_{right}[j]$ are 1 and $CM\_e[i,j]$ is not invalid
7.              $D_v[CM\_e[i,j]] = 1$;
8.          end if
end.
```

Algorithm 5.5 *edge_function_edge_coloring* (v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, …, m]

Output: Updated bitmap of list of distance vectors of node v

```
begin
    1. $d = weight(<v, parent(v)>)$;
    2. Create a new list $D_e$ containing no vector for this edge;
    3. for all integers $j$, $0 \le j \le (l+2)^m - 1$ do
    4.         if $D_v[j]$ is 1 then
    5.             $D_e[AM\_e[j, d]] = 1$;
    6.         end if
end.
```

Algorithm 5.6 *optimal_l_edge_coloring*(T)

Input: A binary decomposition tree T

Output: An optimal l-edge-coloring of T

```
begin
    1. Construct Binary Decomposition Tree $T_b$ from $T$;
    2. Determine the upper bound for number of colors needed ($\alpha$);
    3. Compute the matrices $CM\_e$ and $AM\_e$;
    4. $l = 1$;          // lower bound
    5. $u = \alpha$;          // upper bound
    6. loop until the optimum solution is found
    7.         $i = (l + u)/2$;
    8.         $l\_edge\_coloring(root(T_b), i)$;
```

```
9.          adjust the color range depending on the outcome of previous step;
10.         perform next iteration in the updated range;
11. if $j$, $1 \le j \le \alpha - 1$ is an integer such that a coloring with $j + 1$ colors exists but
        there is no coloring with $j$ colors then
12.         $j$ is the optimal number of colors
13. else
14.         $\alpha$ is the optimal number of colors
end.
```

The main result of this section is the following theorem whose proof follows directly from the proof of theorem 3.6.

Theorem 5.1 For any positive integer l, an optimal l-edge-coloring of a tree having n vertices can be found in time $O(\log\alpha \times (l+2)^{2\alpha} \times n)$, where $\alpha = \frac{\Delta^{l+1} - 1}{\Delta - 1}$. □

5.3 A parallel algorithm for optimal *l*-edge-coloring of trees

In this section we will give an efficient parallel algorithm to solve the l-edge-coloring problem. As in Section 5.2, the algorithms will use the pre-computed matrices *CM_e* and *AM_e* (Section 5.1) instead of *CM* and *AM* respectively. The *l_edge_coloring_parallel* is exactly equivalent to *l_vertex_coloring_parallel* algorithm (Algorithm 4.3) and the *optimal_l_edge_coloring_parallel* algorithm is basically similar to *optimal_l_vertex_coloring_parallel* (Algorithm 4.4) algorithm. We now specify the parallel algorithms for the l-edge-coloring problem on trees.

Algorithm 5.7 *node_function_edge_color_parallel*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, …, m]

Output: Updated bitmap of list of distance vectors of node v

```
begin
1. $D_{left}$ = bitmap(left($v$));
2. $D_{right}$ = bitmap(right($v$));
3. Create two new lists $D_v$ and $D_f$ containing no vector for this node;
4. for all integers $i$, $0 \le i \le (l+2)^m - 1$ in parallel do
5.     for all integers $j$, $0 \le j \le (l+2)^m - 1$ in parallel do
6.         if $D_{left}[i]$ and $D_{right}[j]$ are 1 and $CM\_e[i,j]$ is not invalid
7.             $D_v[CM\_e[i,j]] = 1$;
8.         end if
end.
```

Algorithm 5.8 *edge_function_edge_color_parallel*(v, m)

Input: A node v in the binary decomposition tree T_b, a color set [1, …, m]

Output: Updated bitmap of list of distance vectors of node v

```
begin
1. $d$ = weight(<$v$, parent($v$)>);
2. Create a new list $D_e$ containing no vector for this edge;
3. for all integers $j$, $0 \le j \le (l+2)^m - 1$ in parallel do
4.         if $D_v[j]$ is 1 then
5.             $D_e[AM\_e[j, d]] = 1$
6.         end if
end.
```

```
Algorithm 5.9 l_edge_coloring_parallel(T, m)
Input: A binary decomposition tree T and a color set [1, ..., m]
Output: All l-vertex-coloring of T with m colors (if there exists any)
begin
  1. For all integers i, $1 \le i \le m$, initialize the list of distance vectors of each leaf
     (excluding root(T)) so that the list contains all distance vectors of the form –
     $\langle d_1, d_2, \ldots d_{i-1}, 0, d_{i+1}, \ldots, d_m \mid d_j = \infty$ where $1 \le j \le m$ and $j \ne i \rangle$;
     root(T) contains a single vector $\langle d_i, 1 \le i \le m, d_i = \infty \rangle$
  2. Initialize the list of distance vectors of each internal vertex to be an empty list;
  3. Number the leaf vertices of T by 1, 2, ... from the left to the right excluding the
     leftmost and rightmost leaves;
  4. for $\lceil \log_2(n+1) \rceil$ iterations do
  5.     Apply shunt operation concurrently to all odd numbered leaves that are left
         children;
  6.     Apply shunt operation to remaining odd numbered leaves;
  7.     Remove odd numbered nodes from consideration and sequentially renumber
         all even numbered leaves;
end
```

```
Algorithm 5.10 optimal_l_edge_coloring_parallel(T)
Input: A tree T to be colored
Output: Optimal l-edge-coloring of T
begin
  1. Construct the binary decomposition tree $T_b$ of T
  2. $\alpha \leftarrow$ upperbound($T_b$)
  3. Compute the matrices CM_e and AM_e
  4. for all integers j, $1 \le j \le \alpha$ in parallel do
  5.     l_edge_color_parallel($T_b$, j)
  6. for all integers j, $1 \le j \le \alpha - 1$ in parallel do
  7.     if a coloring with j+1 colors exists but there is no coloring with j colors then
  8.         j is the optimal number of colors
```

end

The main result of this section is the following theorem whose proof follows directly from the proof of theorem 4.3.

Theorem 5.2 An optimal l-edge-coloring of a tree T can be found in $O(\log_2 n)$ time using $O\left((l+2)^{2\alpha} \times n \times \alpha\right)$ processors on the EREW PRAM model, where n is the number of vertices in T and α is the upper bound of the number of colors. □

5.4 Finding the Color Assignment

Till now we have not mentioned how we can obtain the actual colors assigned to the nodes. In this section we turn our focus to that topic. We will store two numbers with each distance vector d_v. These two numbers imply that the distance vector d_v is the result of a combine operation between d_l and d_r. From this information we can go from root down to the leaf level and retrieve one color assignment or color vector for each existing distance vector at root. If we are interested in all color vectors, we have to maintain a list of 2-tuples $< d_l, d_r >$ that include all combinations that resulted the distance vector. To retrieve color information, we need to recursively explore with each tuple, one at a time. With this representation, the node function still runs in $O(1)$ time. The color retrieval operation can be done in $O(n)$ time if we are interested in any valid color assignment. So the overall complexities of our algorithms remain unaffected.

5.5 Conclusion

In this chapter we have presented a sequential and a parallel algorithm to solve the l-edge-coloring problem on trees. Our sequential algorithm finds the optimal l-edge-coloring of a tree in time$O\left(\log\alpha \times (l+2)^{2\alpha} \times n\right)$ while the parallel algorithm finds the optimal l-edge-coloring of a tree in $O(\log_2 n)$ time using $O\left(\alpha \times (l+2)^{2\alpha} \times n\right)$ processors on the EREW PRAM model, where n is the number of vertices in T and α is the upper bound of the number of colors. Both these algorithms are first known algorithms that solve the l-edge-coloring problem on trees.

Chapter 6

Conclusion

This thesis deals with a generalized vertex-coloring problem and a generalized edge-coloring problem on trees. Our parallel algorithms for solving the l-vertex-coloring and l-edge-coloring problem as well as the sequential algorithm for solving l-edge-coloring problem are first such algorithms. Besides, our sequential algorithm for solving l-vertex-coloring problem is the first direct solution to the problem that guarantees an optimal solution.

In Chapter 2 we have defined some basic terms needed for solving generalized coloring problems. We also described different PRAM models. Finally we described techniques of tree contraction and decomposition of an arbitrary tree into equivalent binary tree. These techniques played key roles in our algorithms.

In Chapter 3 we gave a sequential algorithm to solve the l-vertex-coloring problem on trees. For any positive integer l, our algorithm finds an optimal l-vertex-coloring of a tree having n vertices in time $O(\log\alpha \times (l+2)^{2\alpha} \times n)$, where $\alpha = \frac{\Delta^{l+1}-1}{\Delta-1}$. For bounded l and a tree with bounded degrees, the algorithm finds an optimal l-vertex-coloring of T in linear time. This is the first sequential algorithm for the l-vertex-coloring problem that guarantees an optimal solution.

Chapter 4 gives a parallel algorithm for the same problem in Chapter 3. Our algorithm uses B-ATC with tree contraction technique. Our parallel algorithm finds an optimal l-vertex-coloring of a tree T in $O(\log_2 n)$ time using $O(\alpha \times (l+2)^{2\alpha} \times n)$ processors on the EREW PRAM model, where n is the number of vertices in T and α is the upper bound of the number of colors. This is the first parallel algorithm to solve l-vertex-coloring problem.

Chapter 5 is devoted to *l*-edge-coloring problem. First we have described a technique to transform the *l*-edge-coloring problem into *l*-vertex-coloring problem on trees. Then we identified the modifications in our previous algorithm that are needed to solve the *l*-edge-coloring problem. Finally we described a sequential and a parallel algorithm to solve the *l*-edge-coloring problem on trees. The complexities of these algorithms are the same as their corresponding algorithms for *l*-vertex-coloring.

In this thesis, we have given sequential and parallel algorithms for a generalized coloring of trees. However, following problems are still open:

1. Can we find the optimal coloring of a tree without an exhaustive search in the solution space?

2. Can we determine tight bounds on the number of colors?

References:

[ADK89] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick and T. Przytycka, A simple parallel tree contraction algorithm, *J. Algorithms* **10**, pp. 287-302, 1989.

[BH95] H. L. Bodlaender, T. Hagerup, Parallel algorithms with optimal speedup for bounded treewidth, *Proc 22nd Int. Colloq. on Automata, Languages and Programming, Lecture Notes in Computer Science* **10**, Springer, Berlin, .pp. 268-279, 1995.

[BPT92] R. B. Borie, R. G. Parker and C. A. Tovey, Automatic generation of linear time algorithms from predicate calculus descriptions of problems on recursively constructed graph families, *Algorithmica* **7**, pp. 555-581, 1992.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the theory of NP-completeness*, W. H. Freeman & Co., New York, 1979.

[GMT88] H. Gazit, G.L. Miller and S.H. Teng, Optimal tree contraction in an EREW model, *Concurrent Computations: Algorithms, Architecture and Technology*, (S.K. Tewkesbury, B.W. Dickinson and S. C. Schwartz, Eds.) Plenlum, New York, 1988.

[Gup66] R. P. Gupta, The Chromatic index and the degree of a graph, *Not. Amer. Math.. Soc.* **13**, pp. 719, 1966.

[Jag96] Chris Jagger, Coloring powers of graphs, Manuscript, 1996.

[Jos92] J. Joseph, *An Introduction to Parallel Algorithms*, Addison-Wesley Inc., 1992.

[KY00] Md. Abul Kashem and Ayesha Yasmeen, Optimal *l*-vertex-coloring of trees, *International Conference on Computer and Information Technology* (ICCIT), pp. 151-153, 2000.

[MR86] G.L. Miller and J.H. Reif, *Parallel tree contraction and its applications*, Proc. of the 26th IEEE Symp. on Foundation of Comp. Sci. **47**, pp. 277-298, 1986.

[Viz64] V.G. Vizing, On an estimate of the chromatic class of a p-graph, *Diskret. Analiz* **3**, pp 25-30, 1999.

[Wes99] D. B. West, *Introduction to Graph Theory*, Prantice-Hall Inc.,1999.

[ZKN00] X. Zhou, Y. Kanari and T. Nishizeki, Generalized vertex-colorings of partial k-trees. *IEICE Trans. on Fundamentals*, pp. 555-581, 2000.

Index