SEQUENTIAL AND PARALLEL OPTIMIZATION ALGORITHMS FOR RECURSIVE GRAPHS

by

Sanjeev Mahajan

B.Tech., Indian Institute Of Technology, Delhi; 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

Sanjeev Mahajan 1986

SIMON FRASER UNIVERSITY

December 1986

APPROVAL

Name : Sanjeev Mahajan

Degree : Master of Science

Title of Thesis : Sequential And Parallel Optimization Algorithms For Recursive Graphs.

Examining Committee :

Chairperson : Binay Bhattacharya

Senior Supervisor: Joseph Peters

Pavol Hell

External Examiner: Brian Alspach
Department of Mathematics
Simon Fraser University

Date Approved: Dec. 15, 1986

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

SEQUENTIAL AND PARALLEL OPTIMIZATION ALGORITHMS FOR RECURSIVE GRAPHS

Author:

(signature)

SANJEEV MAHAJAN

(name)

14th April, 1987

(date)

ABSTRACT

In a recent paper, Bern, Lawler and Wong described a uniform theory for developing linear time algorithms which find optimal subgraphs obeying certain properties called *regular* in certain families of graphs called *k-terminal recursive*. In this thesis, the theory is generalized to non-regular properties for k-terminal recursive graphs. Algorithms for constructing the tables on which the linear time algorithms operate are developed for regular properties which are *local*. The linear time sequential algorithms are adapted to yield $O(\log n)$ time parallel algorithms which use $O(n)$ processors on a CRCW (concurrent-read-concurrent-write) PRAM.

To my elder brother.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Joseph Peters, for his patience, guidance and encouragement during the past year for preparation of this thesis, and for all the time he spent into reading the many drafts of the thesis, giving valuable advice and corrections for both the research results and stylistic errors.

I would also like to thank the other member of my committee, Pavol Hell for giving me valuable suggestions and for reading and correcting my thesis. My sincere thanks are also in order for Brian Alspach, my external examiner for correcting mathematical aspects of my thesis. I would like to thank the faculty and the staff of the school, who have contributed greatly in creating an environment akin to carrying out my research.

There are a number of friends without which this list of acknowledgements cannot be completed. I am deeply indebted to Ada Fu who had the patience to read some parts of my thesis and to give me valuable suggestions. My special thanks go to Claudia Morawetz who gave me moral support when I most needed it. Finally, I am grateful to Simon Fraser University and the School of Computing Science for financial support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

A wide variety of situations in fields such as Operations Research, Engineering and Economics can be modeled by graphs. Some of the problems arising from these situations can be transformed into graph optimization problems such as maximum independent set, minimum dominating set and minimum coloring (or chromatic number). For example, course and exam scheduling problems are often formulated as graph coloring problems. The Traveling Salesperson Problem (TSP) is another example that can be modeled by graphs.

Unfortunately, most of the graph-optimization problems that arise in practice are NP-hard for general graphs [9]. However, many of these problems can be solved in polynomial time (and sometimes even linear time) when restricted to certain families of graphs. Linear time algorithms have been found for such problems for families of graphs such as trees and series parallel graphs. Until recently, most of the fast algorithms for such problems had been solved by ad hoc techniques until Bern, Lawler and Wong developed a uniform theory for linear time graph optimization problems in [1]. The theory applies to problems which ask for optimal subgraphs (maximum or minimum cardinality or weight) obeying properties called *regular*, in families of graphs called *k-terminal recursive*. We will define the terms *k−terminal recursive* families and *regular* properties in latter sections. The theory explains why certain subgraph problems which are NP-hard for general graphs, have linear time algorithms when restricted to some families of graphs.

The BLW theory also predicts linear time algorithms for several thousands of other problems and provides a methodology of constructing such algorithms. The algorithms proceed in a bottom up fashion through *parse-trees* of the graphs and can be succinctly described by using either a *state table* or a finite set of recurrences relating an

internal node of a parse tree to the children of the node.

With the advent of a large number of parallel computers, the importance of finding parallel algorithms is increasing. Fast parallel algorithms have been found for a variety of problems such as connectivity [3] and biconnectivity [5] in graphs, and minimum spanning trees in graphs [6]. Miller and Reif describe a technique called "parallel tree contraction" in [4]. A wide variety of problems in parallel computation are amenable to tree contraction. Tree contraction can be briefly described by two operations : RAKE and COMPRESS. RAKE is the operation of removing the leaves of the given tree in parallel. COMPRESS identifies pairs of adjacent nodes of all maximal chains in parallel (a chain is a path in a tree in which each node has exactly one child). CONTRACT is the simultaneous application of RAKE and COMPRESS. In [4], it is shown that $O(log\ n)$ CONTRACT operations are sufficient to reduce a tree to a single node. The potential of tree contraction lies in the fact that RAKE and COMPRESS can be thought of as abstract operations and can take different forms depending on the problem under consideration.

This research forks off in two directions : On one hand, this research extends and generalizes the BLW theory. The existence of non-regular properties is shown and a generalized theory is developed to handle non-regular properties. The generic algorithm for the BLW theory uses a look-up table called *multiplication table* which depends on the property and the family corresponding to a particular graph optimization problem. The construction of the table is left to human ingenuity. It is shown in this research how to mechanize the construction of such tables for certain properties which are called *local*. The BLW theory deals only with subgraph problems. The theory is extended so as to encompass vertex partitioning problems such as *chromatic number* and *domatic number* problems. These extensions and generalizations will be discussed in detail in a latter section.

It is also shown in this research how to adapt the generic linear time algorithm of the BLW theory to yield a tree contraction based $O(log\ n)$ time algorithm using $O(n)$ processors. As the BLW theory operates on the parse trees of the given graphs, we need to build fast parallel algorithms for finding parse trees for the family of graphs we are dealing with. Fast algorithms for building parse trees for rooted trees and 2-terminal series parallel graphs are also presented in this research.

1.2. Background

The BLW theory applies to k-terminal recursive families of graphs. Intuitively, a family of graphs is said to be *k-terminal recursive* if every graph in the family can be generated by a finite number of applications of

predefined *composition operations* starting with a finite number of *basis graphs*. Each graph in the family has a distinguished set of vertices of fixed size k, called the *terminals* and the composition operations can only be applied to the set of terminals.

A graph G is a triple (V, E, T) where V is the set of vertices and E is the set of edges and T is a subset of V of size k, for some fixed k.

Definition [2]: A family Γ of graphs is called *k-terminal recursive* if it can be defined according to the following schema.

(1) Let $B = \{ B_1, B_2, \ldots, B_l \}$ be a finite set of basis graphs, where each B_i is a finite graph having an ordered set of k *terminals* (i.e. vertices), for some fixed integer k.

(2) Let $O = \{ o_1, o_2, \ldots, o_m \}$ be a finite set of rules of composition, whereby any two graphs $G_i=(V_i, E_i, T_i)$ and $G_j=(V_j, E_j, T_j)$ may be combined to produce a new graph $H=G_i\ o_r\ G_j$ $(1 \le r \le m)$. Each rule of composition must be defined in terms of the following allowable operations:

i) identify a terminal in T_i with a terminal in T_j.

ii) add an edge between a terminal in T_i and a terminal in T_j.

iii) select k terminals for the graph $H=G_i\ o_r\ G_j$ from $T_i \cup T_j$.

(3) Define the family Γ of graphs recursively as follows:

i) Any $B_i \in B$ is in Γ.

ii) If H' and H'' are in Γ and o_j is any operation in O, then the graph $H=H'\ o_j\ H''$ is also in Γ.

Examples: An example of a k-terminal recursive family of graphs, where $k = 1$, is rooted trees [1]:

(1) The triple $(\{x\}, \{\}, \{x\})$ is a rooted tree with root x.

(2) If $T_1 = (V_1, E_1, r_1)$ and $T_2 = (V_2, E_2, r_2)$ are rooted trees with roots r_1 and r_2 respectively, then the triple $T = (V, E, r)$ where

$$V = V_1 \cup V_2$$

$$E = E_1 \cup E_2 \cup \{r_1 r_2\}$$

$r = r_1$

is a rooted tree.

Figure 1.1 illustrates the composition of two rooted trees to yield a third rooted tree.

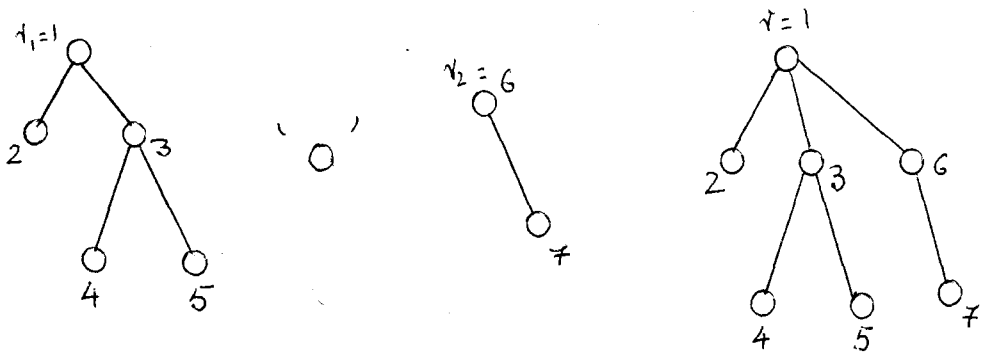

Figure 1.1 The Rooted Tree Composition Operation

We now give an example of a 2-terminal recursive family:

Definition [1]: The series-parallel graphs are defined in the following way:

1) The graph $G = (\{l, r\}, \{(l, r)\})$ is series-parallel, with left and right terminals $[l, r]$.

2) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are series-parallel graphs, with left and right terminals $[l_1, r_1]$ and $[l_2, r_2]$ respectively, then :

a) The graph obtained by identifying r_1 and l_2 is a series-parallel graph, with l_1 and r_2 as its left and right terminals respectively. This graph is the *series* composition of G_1 and G_2.

b) The graph obtained by identifying l_1 and l_2 and also r_1 and r_2 is a series-parallel graph called the *parallel* composition of G_1 and G_2. This graph has l_1 $(= l_2)$ and r_1 $(= r_2)$ as its left and right terminals respectively.

An example of a 3-terminal recursive family is *protoHalin graphs.*

Definition [1]: *ProtoHalin graphs* are defined as follows:

1) The graph $G = (\{v'\}, \phi)$ is a protoHalin graph; v' is its root and its left and right terminals.

2) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are protoHalin graphs, with terminals (t_1, l_1, r_1) and (t_2, l_2, r_2) respectively, then:

i) If $G_1 = (\{v'\}, \phi)$, then $G = (\{v'\} \cup V_2, E_2 \cup \{(v', t_2)\})$ is a protoHalin graph, with terminals (v', l_2, r_2).

ii) If $G_1 \neq (\{v'\}, \phi)$, then $G = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(t_1, t_2), (r_1, l_2)\})$ is a protoHalin graph with terminals (t_1, l_1, r_2).

Definition: A *parse tree* for a graph G belonging to a k-terminal recursive family Γ of graphs is a tree whose internal nodes represent composition operations of Γ and the leaves represent the basis graphs to which these operations are applied to result in G. Each internal node in the parse tree can be considered as representing the graph corresponding to the parse tree rooted at the node.

A rooted tree and its corresponding parse tree are shown in Figure 1.2.

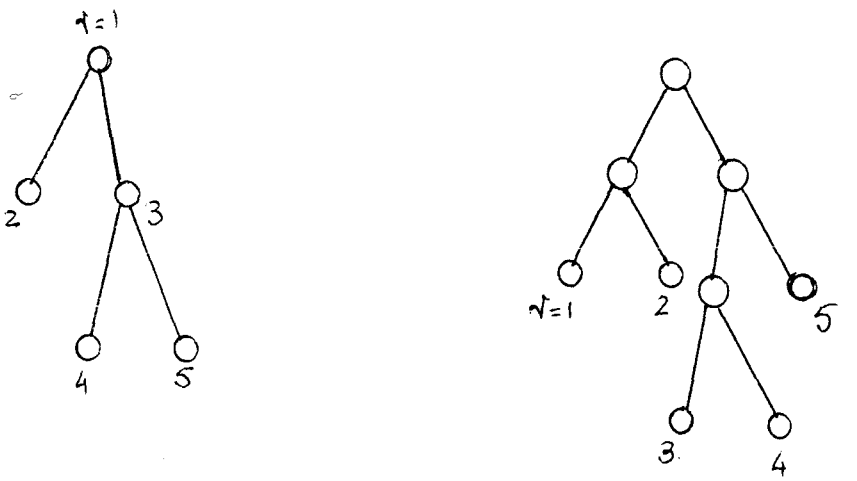

Figure 1.2 A Rooted Tree And Its Corresponding Parse Tree

Now we are in a position to define *regular* properties. Let Γ be a k-terminal recursive family and P be a computable predicate defined on graph × subgraph pairs. Assume for now, that Γ is defined by a single rule of composition o and that we are only interested in subgraphs as vertex subsets. In this restricted case, we can unambiguously extend the definition of the operator o to graph × subgraph pairs by defining $(G_1, S_1)\ o\ (G_2, S_2) = (G_1\ o\ G_2, S_1 \cup S_2)$. Let the class of all graph × subgraph pairs, where the graph belongs to Γ, be denoted by Γ_S. Intuitively, we say that a property P is *regular* if Γ_S can be partitioned into a finite number of

classes such that, for each class, either all or none of the members satisfy P, and the class C to which $(G, S)=(G_1, S_1)\ o\ (G_2, S_2)$ belongs, is a binary function of the classes of (G_1, S_1) and (G_2, S_2) respectively. Since the number of equivalence classes is finite, this binary function can be represented as a multiplication table on the set of equivalence classes.

More formally, we define regularity as follows.

Definition: Let Γ be a k-terminal recursive family with a single composition operation o and Γ_S be the class of all graph × subgraph pairs on Γ. Let T be a set that has a binary operation • defined on it. We say that a mapping h from Γ_S onto T defines a *homomorphism* with respect to the class Γ and the property P if for every (G_1, S_1) and (G_2, S_2) in Γ_S

(1) $h(G_1, S_1) = h(G_2, S_2)$ implies $P(G_1, S_1) = P(G_2, S_2)$.

(2) $h((G_1, S_1)\ o\ (G_2, S_2)) = h(G_1, S_1) \bullet h(G_2, S_2)$.

We say that P is *regular* with respect to Γ if it admits of a homomorphism with a finite range. This finite range may be identified with the set of equivalence classes and • is the binary multiplication operation defined on it. Bern, Lawler and Wong [1] showed that vertex independence in rooted trees is a regular property by giving the following multiplication table (Table 1.1).

•	C_1	C_2	C_3
C_1	C_1	C_1	C_3
C_2	C_2	C_3	C_3
C_3	C_3	C_3	C_3

Table 1.1 Multiplication Table For Independence In Rooted Trees

Intuitively, C_1 represents the class of all graph × subgraph pairs (G, S) such that S is independent in G and the root of G is not in S, C_2 is the class of all pairs (G, S) such that S is independent in G and the root of G is in S and C_3 is the class of all pairs (G, S) such that S is *not* independent in G. C_1 and C_2 are *initial classes*, that is, the classes characterizing graph × subgraph pairs on the basis graph. The corresponding homomorphism maps a given graph × subgraph pair (G, S) to the class to which (G, S) belongs.

The multiplication table of a regular property is the key to the linear time optimal subgraph algorithm of the BLW theory. We briefly outline the generic algorithm as given in [1] and illustrate it by finding a maximum independent set in a rooted tree. The algorithm works on the parse tree of the given graph. Starting with the leaves,

it finds an *optimal representative* for each equivalence class for the graph × subgraph pair represented by each node of the parse tree. At the leaves of the parse tree, an optimal representative will be an optimal *primitive* graph × subgraph pair in the class (consisting of a basis graph in that family and a subgraph of that basis graph). At each interior node v of the parse tree, an optimal representative of each equivalence class for the graph × subgraph pair represented by v is a function of the optimal representatives of the children of v. For each equivalence class C_k, we consider all pairs C_i , C_j such that $C_k = C_i \bullet C_j$, and find the "best" such pair. This can be done by relating the optimal weight in the class C_k for node v to the optimal weights in classes C_i and C_j in the left and the right children of v and then optimizing over all such i and j. This can be compactly represented by storing respectively a vector of length l, at each node v, where l is the number of equivalence classes. The ith entry in the vector is the optimal weight of a subgraph satisfying P, assuming that the graph × subgraph pair corresponding to v belongs to class C_i. So, the vector at v is related to the corresponding vectors at its children by certain recurrence relations.

If we are interested only in the cardinality (or weight) of an optimal subgraph, we need only remember the optimal cardinality (or weight) achieved by each equivalence class. If we are interested in recovering the optimal subgraph itself, we also remember, for each node and each class, which pair C_i, C_j achieved the optimum. The optimal solution is the best of the optimal representatives of the accepting classes at the root of the parse tree.

We will explain this algorithm with the help of an example. Let us find a maximum independent set in the rooted tree given in Figure 1.3a. The parse tree of the tree is shown in Figure 1.3b. At each internal node, we store a vector of length 2 corresponding to classes C_1 and C_2. We do not need to store information about C_3 because it is a rejecting class and it is a *sink* class. (A sink class, when composed with any other class yields the same sink class.) We denote the maximum cardinality of an independent set belonging to class C_i at node v by $a_i(v)$ and we denote the left and right children of node v by v_l and v_r respectively. The multiplication table for the maximum independent set problem for rooted trees yields the following recurrences:

$$a_1(v) = \max \{ a_1(v_l) + a_1(v_r), a_1(v_l) + a_2(v_r) \}$$

$$a_2(v) = a_2(v_l) + a_1(v_r).$$

These recurrences are propagated bottom up through the parse tree of Figure 1.3b to get the maximum independent sets corresponding to classes C_1 and C_2 in the given rooted tree. The maximum of these two gives the size of the maximum independent set.

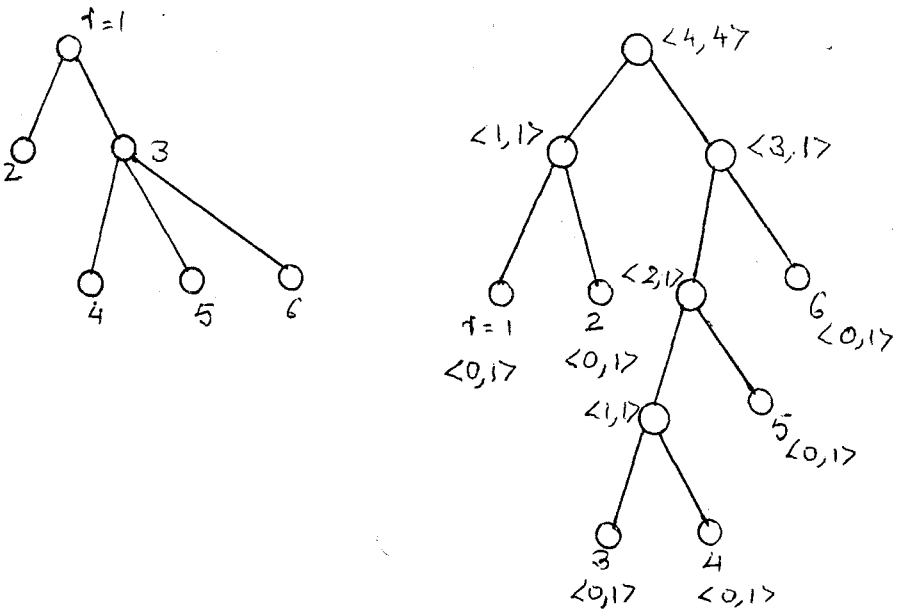

Figure 1.3 Finding Maximum Independent Sets in Rooted Trees

It is easily seen that this algorithm is linear in the *size* of the parse tree (The *size* of a parse tree is the number of nodes in the parse tree). If the size of the parse tree is linear in the size of input and if the parse tree for the given family of graphs can be obtained in linear time, then we have a linear time algorithm for finding optimal subgraphs satisfying a given regular property P with respect to a given k-terminal recursive family of graphs.

Hedetneimi, Laskar and Wimer [2] have developed a theory similar to the BLW theory using a slightly different approach. However, our results are based on the BLW theory because of its simplicity and elegance. One of the contributions of this thesis is an efficient parallel algorithm for the generic algorithm of the BLW theory. The parallel algorithm is based on a technique called *parallel tree contraction* introduced by Miller and Reif [4]. Miller and Reif show how to use their technique to obtain fast parallel algorithms for the evaluation of arithmetic expressions, for isomorphism in trees, and for finding canonical labelings of trees and planar graphs.

Parallel tree contraction consists of two parallel operations which are applied simultaneously. Let $T = (V, E)$ be a rooted tree with n nodes and root r. RAKE is the operation of removing all leaves of T in parallel. If the tree is highly unbalanced (that is, the difference between the number of nodes in the subtrees of the root of the tree is very large); RAKE may have to be applied $O(n)$ time. This problem is overcome by introducing another operation. A sequence of nodes $v_1, v_2, \ldots, v_k$ is said to be a *chain* if v_{i+1} is the only child of v_i $(1 \leq i \leq k)$ and v_k has exactly one child. COMPRESS identifies v_i and v_{i+1} for all odd i in parallel. CONTRACT is the simultaneous application

of RAKE and COMPRESS to the entire tree. Figure 1.4 shows an example of the CONTRACT operation.

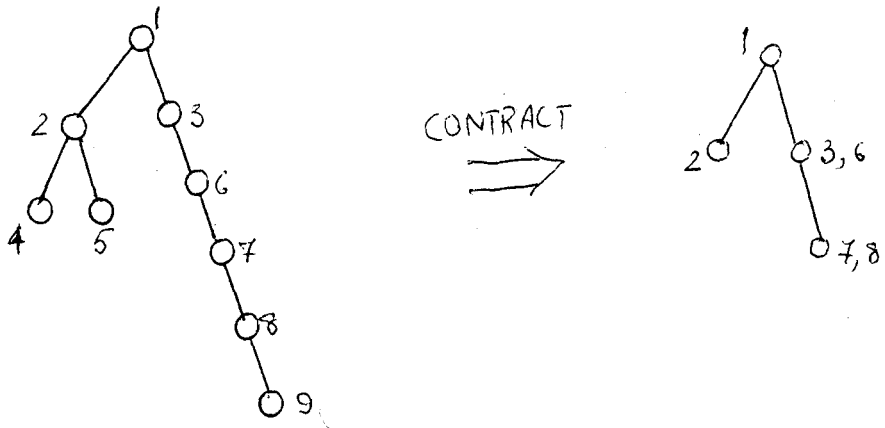

Figure 1.4 An Example Of The Contract Operation

Theorem [4]: After $\lceil \log_{5/4} n \rceil$ executions of CONTRACT, a tree with n vertices is reduced to its root.

A naive implementation of COMPRESS requires $O(\log n)$ parallel time. We have to determine the degree of each node and then combine consecutive pairs of nodes. Miller and Reif [4] present constant time deterministic and randomized algorithms for COMPRESS. These algorithms for COMPRESS lead to an $O(\log n)$ deterministic tree contraction algorithm which uses $O(n)$ processors and an $O(\log n)$ randomized version which uses $O(n/\log n)$ processors.

1.3. Overview

This thesis is arranged as follows:

1) Chapter 2 contains extensions and generalizations of the BLW theory. In particular, the existence of non-regular properties is shown and a generalized framework is provided for finding efficient algorithms for optimal subgraphs satisfying non-regular properties in k-terminal recursive families of graphs. The BLW theory does not provide a method for constructing multiplication tables or finite range homomorphisms for regular properties. In a sense, this process is ad hoc and depends heavily on human ingenuity. A natural question to ask is whether the process can be mechanized. We prove that the process cannot be mechanized by showing that it is unsolvable to determine a finite range homomorphism (if it exists) for any arbitrary computable property.

If a property is infinite, then it has to be described in terms of an algorithm, and one of the main reasons which gives rise to unsolvability of determining a finite range homomorphism for a given computable property is the arbitrariness of the algorithm describing the property We could insist that the algorithm for describing the property be given in terms of a finite range homomorphism, but a finite range homomorphism for the property is dependent both on the property and the k-terminal recursive family being considered. Hence, for example, we would have different finite range homomorphisms for independence in rooted trees and series parallel graphs. Therefore we cannot uniformly describe a property in terms of a finite range homomorphism.

This disadvantage motivates us to look for a characterization of regular properties, which is both graph-theoretic and is uniformly defined for all graphs. Although we have not been fully successful in our venture, we have shown in this research that properties which we call *local* (we will define *locality* in Chapter 2) possess both the stated characteristics and are regular. We also build effectively, finite range homomorphisms for local properties for any k-terminal recursive family of graphs. In particular, we show that local properties such as independence, domination, irredundance and *vertex j–independence* (two vertices in a graph are j-independent if the number of edges in the shortest path between them is at least j) are regular for any k-terminal recursive family of graphs. We then extend these ideas to edge-induced subgraph properties and show that the *cycle property* (a graph $\times$ subgraph pair (G, H) has the *cycle property* if H is a cycle in G) and maximal matching are both regular with respect to any k-terminal recursive family of graphs.

We also extend BLW theory to encompass vertex partitioning problems. We formulate such a theory and then furnish some examples where the theory might be used.

2) In Chapter 3, we use parallel tree contraction to adapt the generic linear time algorithm of the BLW theory to yield an $O(\log n)$ time algorithm using $O(n)$ processors where n is the number of nodes in the parse tree. As the BLW generic algorithm works on the parse tree of the given graph in the given family, we need to have efficient parallel algorithms to generate parse trees for a given family of graphs. In this research, we exhibit algorithms to build parse trees for rooted trees in $O(\log n)$ time using $O(n)$ processors and for series-parallel graphs in $O(\log^2 n + \log m)$ time using $O(m + n)$ processors where m is the number of edges and n the number of vertices in the graph. He [11, 12] has independently obtained similar results on such parallel algorithms using similar techniques. We will describe his results and then compare his results with our results in Chapter 3.

Chapter 4 contains conclusions and a discussion of open problems emerging as a result of this research.

CHAPTER 2

EXTENSIONS AND GENERALIZATIONS OF THE BLW THEORY

2.1. Introduction

The BLW theory provides a strong foundation for designing linear time (sequential) algorithms for certain graph optimization problems restricted to k-terminal recursive families of graphs. In this chapter, we first show the existence of non-regular properties and then generalize the BLW theory to handle such properties. Next we show that it is unsolvable to find a finite range homomorphism, if one exists, for an arbitrary computable property (given in terms of an arbitrary algorithm) in any k-terminal recursive family of graphs. In contrast to this unsolvability result for arbitrary properties, we identify several regular properties for which we can effectively construct finite range homomorphisms for any k-terminal recursive family of graphs. In other words, the regularity of these properties does not depend on the particular k-terminal recursive family being dealt with. These regular properties are *vertex independence*, *vertex 2–independence*, *vertex domination*, and *irredundance*. These four properties are *local* in a certain sense and we prove that all local properties are regular. We also extend our results to edge-induced subgraph problems and show that properties such as *maximal matching* and *cycle* property are regular. An easy observation shows that the BLW theory can be modified so as to solve vertex partition problems. We describe such a theory and then show that *coloring* and domatic labelings are both regular with respect to all k-terminal recursive families of graphs.

Henceforth, we will drop the phrase 'regular with respect to a k-terminal recursive family' and instead simply use 'regular' when the statement is true for any k-terminal recursive family. However, note that a property which is regular with respect to one k-terminal recursive family may not be regular with respect to another. Let P be the property defined as follows:

A pair (G, S) satisfies P if either G is a tree and S is independent in G or G is not a tree and S is f*–independent* in G, where f is an integer function satisfying :

i) $f(n)-f(n-1) \leq 1$ for all n.

ii) for all non-negative integers j, there exists a non-negative integer n, such that $f(n) > j$ and $f(n)-f(n-1) < 1$.

f-independence will be defined more carefully in a latter section.

Then we can show that P is regular for trees but is not regular for series-parallel graphs. When we use the term graph × subgraph pair, it is assumed that the graph belongs to the k-terminal recursive family being considered. Unless explicitly stated, we also assume that the subgraph is a vertex subset.

2.2. Non-Regular Properties

In the spirit of the pumping lemma for regular languages [7], we develop a pumping lemma for regular properties, which helps us prove that certain graph properties are not regular.

Before we proceed any further, let us choose a convenient representation for graph × subgraph pairs. We represent a graph × subgraph pair (G, S) by a parse tree of G belonging to a k-terminal recursive family in which the leaves of the parse tree which belong to S are marked. We call this representation the *marked parse tree* representation. For example, if Γ is the family of rooted trees, a rooted tree × subgraph pair and its corresponding marked parse tree representation are given in Figure 2.1.

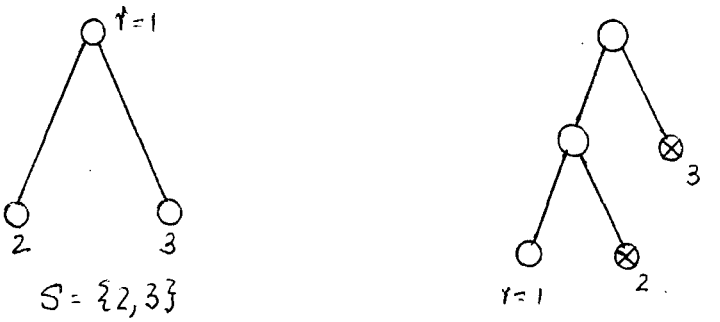

Figure 2.1 Marked Parse Tree Representation Of A Graph × Subgraph Pair.

We will use the marked parse tree representation of graph × subgraph pairs to state and prove the lemma. A path in a marked parse tree from node v to node u (where u is a descendent of v in the parse tree) will be denoted by $v \longrightarrow u$. The length of a path $v \longrightarrow u$ is denoted $|v \longrightarrow u|$. A *fragment* corresponding to a path $v \longrightarrow u$ and

denoted by $frag(v \longrightarrow u)$ is defined to be the marked parse tree obtained by deleting all descendents of u, all ancestors of v, all the edges incident on the deleted nodes, and by making v the root of the parse tree. An example of a fragment is shown in Figure 2.2. The *concatenation* of two fragments $frag(v \longrightarrow u)$ and $frag(x \longrightarrow w)$, denoted $frag(v \longrightarrow u)frag(x \longrightarrow w)$ is obtained by identifying u and x. $frag^n(v \longrightarrow u)$ will denote the concatenation of n copies of $frag(v \longrightarrow u)$.

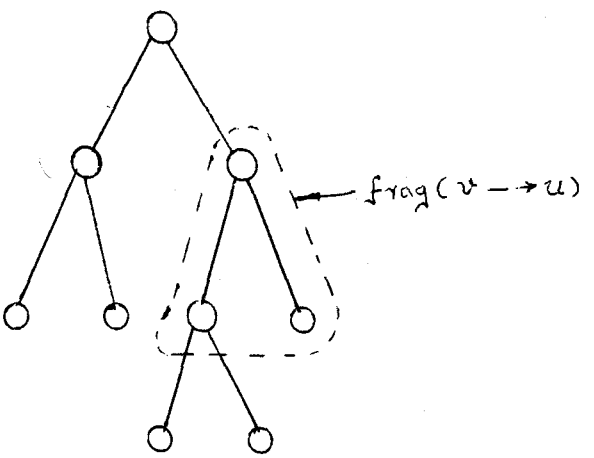

Figure 2.2 An Example Of a Fragment

Pumping Lemma : Let P be a regular property with respect to a given k-terminal recursive family of graphs Γ. Then there exists a constant j, such that, for every marked parse tree satisfying P which has a path $v \longrightarrow u$ with $|v \longrightarrow u| > j$, there exist three paths $w \longrightarrow u$, $x \longrightarrow w$ and $v \longrightarrow x$ where $|x \longrightarrow w| \geq 1$ and $|x \longrightarrow u| \leq j$ such that, if we replace $frag(v \longrightarrow u)$ by $frag(v \longrightarrow x)frag^n(x \longrightarrow w)frag(w \longrightarrow u)$ for any $n \geq 0$ in the marked parse tree, the resulting marked parse tree also satisfies P.

Proof : As the property P is regular, there exists a finite range homomorphism h. Let h have j equivalence classes, $C_1, \ldots, C_j$. Consider a marked parse tree which satisfies P and which has a path $v \longrightarrow u$ with $|v \longrightarrow u| > j$. Since this marked parse tree satisfies P, it must belong to some accepting class C_i. Now since $|v \longrightarrow u| > j$, and the number of equivalence classes is only j, there must exist two distinct nodes w and x in $v \longrightarrow u$ (where w is a descendent of x) such that $|x \longrightarrow u| \leq j$ and the class of node w is the same as that of x. Now the class of any node of the marked parse tree (that is the class of the graph represented by v) depends only on the classes of its children (by the homomorphism property). Hence the class of any node that is the ancestor of x remains unchanged, if

we either identify nodes w and x and eliminate $frag(x \longrightarrow w)$ or if we replace $frag(x \longrightarrow w)$ with $frag^n(x \longrightarrow w)$ for any $n \geq 0$ (when n=0, we just concatenate $frag(v \longrightarrow x)$ and $frag(w \longrightarrow u)$). As the root of the marked parse tree is an ancestor of any node, the class of the root does not change with this operation. Hence the root of the resulting marked parse tree also belongs to C_i which is an accepting class, and the resulting marked parse tree satisfies P. □

Definition : Let $f : N \rightarrow N$ be a function computable in time linear in n. Two vertices in a graph G are said to be *f-independent* if the shortest path between them has at least $f(n)$ vertices, where n is the number of vertices in G. A subgraph S in a graph G is said to be *f-independent* if every two vertices in S are f-independent.

Theorem : Let f be an integer function of n, satisfying the following conditions :

i) for all n, $f(n) \leq n$.

ii) for all n, $f(n) - f(n-1) \leq 1$

iii) for all non-negative integers j, there exists an n such that $f(n) > j$ and $f(n) - f(n-1) < 1$

Then vertex f-independence is a non-regular property for rooted trees.

Before we prove this result, we need the following definition :

Definition: A *pendant vertex* in a graph G is a vertex of degree 1 in G.

Proof: Assume to the contrary, that vertex f-independence is regular. Let us abbreviate this property as P. Then let j be the constant of the Pumping Lemma. Let n be such that $f(n) > j$ and $f(n) - f(n-1) < 1$. Such an n exists by hypothesis. As $f(i) - f(i-1) \leq 1$ for all i, we have $f(n) - f(n-k) < k$ for all $1 \leq k \leq n$. This implies $f(n-k) > f(n) - k$ for all $1 \leq k \leq n$. Consider the marked parse tree, rooted tree × subgraph pair of Figure 2.3.

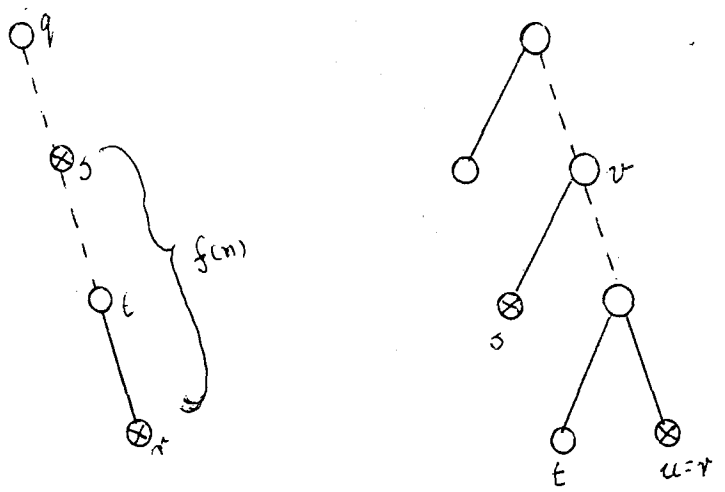

Figure 2.3

Here the rooted tree is a simple path of n vertices with one of the pendant vertices q being the root and the other pendant vertex being the leaf r. The subgraph contains vertices r and s and the distance between r and s is $f(n)$. In the corresponding marked parse tree representation of the graph × subgraph pair, let u be r and let v be the parent of s. Then clearly, the path $v \longrightarrow u$ has $f(n)$ edges in the marked parse tree. Also as $f(n) > j$ and $f(i) \leq i$ for all i, hence $n > j$. Then the given marked parse tree satisfies the property P. Also the path $v \longrightarrow u$ has more than j edges. Now as the property P is regular and the given marked parse tree satisfies the hypothesis of the Pumping Lemma, there exist nodes x and y in the path $v \longrightarrow u$, such that the $frag(y \longrightarrow x)$ can either be pumped up or pumped down and $|y \longrightarrow u| \leq j$. Hence y can never be v, as $|v \longrightarrow u| = f(n) > j$. Let $|y \longrightarrow x| = k$ $(k \geq 1)$. Then if we pump down $frag(y \longrightarrow x)$, the graph × subgraph pair corresponding to the resulting marked parse tree has $n - k$ vertices. The distance between the two marked nodes in the subgraph in the resulting marked parse tree is $f(n) - k$. Hence in the corresponding graph × subgraph pair, there are two vertices that belong to the subgraph and the distance between them is $f(n) - k$. For the resulting graph × subgraph pair to satisfy f-independence, $f(n) - k \geq f(n - k)$. This is a contradiction and hence f-independence is not a regular property. □

Example : The function f such that $f(n) = \lceil \log n \rceil$ for all n is a function that satisfies :

i) $\lceil \log n \rceil \leq n$ for all n.

ii) for all n, $\lceil \log n \rceil - \lceil \log (n-1) \rceil \leq 1$

iii) for all j, there exists an n such that $\lceil \log n \rceil > j$ and $\lceil \log n \rceil = \lceil \log(n-1) \rceil$ (or $\lceil \log n \rceil - \lceil \log(n-1) \rceil < 1$). One such n is 2^{j+2}.

$\lceil \log \rceil$-independence is a non-regular property by the theorem.

2.3. The Generalized Theory For Non-Regular Properties

We exhibited the existence of non-regular properties in the previous section. In this section, we extend the BLW theory to develop efficient algorithms for non-regular properties. The generic algorithm for the generalized theory is not linear for non-regular properties, but is still fast. We first present our theory and then provide an application of the theory.

For a property P to be regular with respect to a given k-terminal recursive family of graphs Γ with a single composition operation o, recall from section 1.2 that there must exist a homomorphism h which maps the class of all graph $\times$ subgraph pairs (where the graph belongs to the family) to a finite set T with an operation $\bullet$ defined on it. The homomorphism satisfies the following two conditions:

1) If $h(G_1, S_1) = h(G_2, S_2)$ then $P(G_1, S_1) = P(G_2, S_2)$

2) $h((G_1, S_1)\ o\ (G_2, S_2)) = h(G_1, S_1) \bullet h(G_2, S_2))$

The *finiteness* of T is a necessary condition for regularity of P. If T is not constrained to be finite, the identity mapping from Γ to T (where T is Γ) is a homomorphism for any arbitrary property. We will extend the BLW theory to non-regular properties by allowing the set T to be infinite. To get a useful extension, however, we will insist that the set T is "sparse".

Definition : Let P be a computable property. Let $\Gamma_S(n)$ be the class of all graph $\times$ subgraph pairs where the number of vertices in each graph is at most n. Let $g: N \rightarrow N$ be a function computable in time linear in n. Let $T(n)$ be a set with cardinality $g(n)$. A *generalized homomorphism* for P is a collection of functions h_n for all $n \geq 1$ where each h_n is a mapping from $\Gamma_S(n)$ to a set $T(n)$ such that the following conditions hold for each $n \geq 0$:

i) If $h_n(G_1, S_1) = h_n(G_2, S_2)$ then $P(G_1, S_1) = P(G_2, S_2)$.

ii) $h_n((G_1, S_1)\ o\ (G_2, S_2) = h_n(G_1, S_1) \bullet h_n(G_2, S_2)$

We assume that, given any n, h_n can be constructed in finite time.

Let P be a given non-regular property. For the generalized homomorphism to be useful, the set $T(n)$ should be *sparse*, so that it is easier to work with the set $T(n)$ than with the class of all graph $\times$ subgraph pairs. Suppose

that h_n is in the generalized homomorphism corresponding to P, and that the size of the corresponding set $T(n)$ is $g(n)$. Also assume that h_n, $T(n)$ and the set of accepting classes in $T(n)$ can be found in time $h(n)$, for any n (where h is a function computable in time linear in n). We claim that optimal subgraphs obeying P in the graphs belonging to the corresponding k-terminal recursive family can be found in time $O(n\ g^2(n) + h(n))$ and in space $g^2(n)$. Since the number of graph $\times$ subgraph pairs with each graph having at most n vertices is an exponential function of n, this is more efficient than the brute force approach, when $g(n)$ and $h(n)$ are both polynomial functions of n.

The generic algorithm of the BLW theory uses a fixed multiplication table. We generalize this generic algorithm by creating a multiplication table each time that depends on the graph that is input (in the form of a parse tree). We count the number of vertices of the given graph, say n, and we use the $O(h(n))$ time generalized homomorphism construction algorithm to yield a multiplication table with $g(n)$ classes. We then form the recurrences depending on the given table and whether we are asked to find the minimum or maximum subgraph obeying the property. These recurrences are obtained in a way similar to the BLW theory. Clearly, this process could require in the worst case $O(g^2(n))$ as the size of the table is $g^2(n)$. We now propagate these recurrences up the parse tree to yield the optimal solution. Since the size of the parse tree is linear in n, there are $O(n)$ propagation steps and the entire algorithm requires $O(h(n) + g^2(n))$ time. The only storage used in the algorithm is the multiplication table, so the space requirements are $g^2(n)$.

We now use this generic algorithm to find maximum f-independent sets in rooted trees where f satisfies :

i) for all n, $f(n) \leq n$;

ii) for all n, $f(n) - f(n-1) \leq 1$;

iii) for all non-negative integers j, there exists an n such that $f(n) > j$ and $f(n) - f(n-1) < 1$.

We showed in section 2.2 that vertex f-independence for such an f is not regular, so the BLW theory cannot be used. We first show that vertex j-independence is regular for rooted trees and build a finite range homomorphism for the property. This will then help us to develop a generalized homomorphism for vertex f-independence. The multiplication table for vertex j-independence for $j = 3$ is shown in Table 2.1. The table has $j + 1$ classes, with j accepting classes and one rejecting class. The representative of the first class, C_1, is the graph $\times$ subgraph pair where the graph is a single vertex, (the root) and the root is not included in the subgraph. The representative of the

second class, C_2, is the graph × subgraph pair where the graph is a single vertex (the root) and the root is included in the subgraph. For the representative of each class C_i, $3 \leq i \leq j$, we have the graph × subgraph pair in which the graph is a simple path of $i - 1$ vertices and one endpoint is the root. The subgraph in this case is the vertex that is the other endpoint of the path. The class C_{j+1} (or REJ) is the only rejecting class. To find $C_l \bullet C_m$ for $1 \leq l, m \leq j$, we just compose the corresponding representatives of the two classes using the rooted tree composition operation and determine whether the resulting graph × subgraph pair has two vertices in the subgraph which are separated by a distance of less than j in the graph. If so, the resulting class is C_{j+1} (or REJ), the only rejecting class. If not, we find the vertex in the resulting graph × subgraph pair, which has the minimum distance to the root in the tree. If this distance is 0, that is, if the root itself is included in the subgraph, then the resulting class is C_2. If the distance is k, where $k \leq j - 1$, the resulting class is C_{k+1}. If the distance is j or greater, then the class is C_1. If one of the classes being composed is the rejecting class C_{j+1}, then the class resulting from the composition is also C_{j+1} (or REJ).

•	C_1	C_2	C_3	REJ
C_1	C_1	C_3	C_1	REJ
C_2	C_2	REJ	REJ	REJ
C_3	C_3	REJ	C_3	REJ
REJ	REJ	REJ	REJ	REJ

Table 2.1 Multiplication Table For 3-independence In Rooted Trees

The procedure above tells us how to build the multiplication table for vertex j-independence for any j. It is easy to see that this table does in fact represent the multiplication table for the property. It is also clear that the algorithm has time complexity $O(j^2)$.

We can use the fact that vertex j-independence is regular for rooted trees for any j and the procedure for building the tables for any j to get an $O(n\, f^2(n))$ algorithm for finding maximum vertex f-independent sets in rooted trees. A generalized homomorphism $h_n : \Gamma_S(n) \rightarrow T(n)$ is obtained by using the homomorphism for $f(n)$-independence. This yields $f(n) + 1$ equivalence classes. So, in this special case of the generic algorithm of the generalized theory, we have $g(n) = f(n) + 1$ and $h(n) = (f(n) + 1)^2$. Therefore, the complexity of finding maximum f-independent sets in n-vertex rooted trees is $O(n\ (f(n) + 1)^2 + (f(n) + 1)^2) = O(n\, f^2(n))$.

It should be noted that the generalized theory does not always give an optimal algorithm. For example, the Pumping Lemma can be used to show that the property of being the center (or bicenter) of a rooted tree is not regular. The generalized theory must be used if we intend to solve the problem by parse tree techniques. The resulting

algorithm in this case cannot be linear, but there is a simple non parse tree algorithm for finding the center of a tree in linear time.

2.4. Unsolvability Of Regularity

Consider the following problem :

[P1] Given a fixed k-terminal recursive family of graphs, and an arbitrary computable property P on graph $\times$ subgraph pairs, find a finite range homomorphism for P with respect to the given family.

Theorem : P1) is unsolvable.

Proof : Without loss of generality, assume that for all n, there exists a graph on n vertices which belongs to the family. Let P2 be the following problem:

[P2] Given a fixed k-terminal recursive family Γ, an arbitrary computable property P, and a finite range mapping h on Γ_S, determine if h is a homomorphism for P.

If P1 were solvable, we could determine a finite range homomorphism h' for P, minimize the number of classes in both h' and h (Bern, Lawler and Wong [1] show how to minimize the number of classes in a given homomorphism), and test if the resulting homomorphisms are isomorphic. (If both h and h' have minimum number of classes and they represent the same property, then h and h' are unique up to isomorphism.) Therefore if P1 is solvable, then P2 is solvable, and if P2 is unsolvable, then P1 is unsolvable. We will now show that P2 is unsolvable by reducing the following unsolvable problem to P2 :

[P3] Given an arbitrary recursive language L over an alphabet $\Sigma = \{a\}$ (whose description is given in terms of an algorithm that always halts), determine if $L = \Sigma^*$.

To show that P3 is unsolvable, we reduce the following known unsolvable problem [7] to P3 :

[P4] Given an arbitrary recursive language L over an arbitrary finite alphabet Γ (whose description is given in terms of an algorithm that always halts), determine if $L = \Gamma^*$.

Given an instance L of P4, given in terms of an algorithm A which accepts strings in Γ^*, we transform A into A' which accepts strings in a^*, by recursively transforming each a^n into the a string in Γ^* using a function similar to the one used by Cantor to prove that rationals are countable. Let A' represent language L'. Then it is clear that

$L' = \Sigma^*$ iff $L = \Gamma^*$. As the transformation from A to A' is recursive and as P4 is unsolvable, P3 is unsolvable too.

We now show that P2 is unsolvable by making the following transformation from P3 to P2 :

Given in an instance L of P3 in terms of an algorithm A which accepts as input, strings of the form a^n, we transform A to A' (where A' accepts as input, graph × subgraph pairs on the given k-terminal recursive family) recursively as follows:

If $a^n \in L$, then P is true for all graph × subgraph pairs on n vertices (where the corresponding graphs belong to the k-terminal recursive family) and P is false for all other pairs (that is pairs which have n vertices such that a^n is not in L).

Also, P is true for all graph × subgraph pairs (where the corresponding graphs belong to the given family) iff $L = \Sigma^*$. There is a finite range homomorphism corresponding to the property that is true for all graph × subgraph pairs on the given family. The homomorphism has one equivalence class and the class is an accepting class. Since P3 is unsolvable, the problem of deciding if a given computable property over a given family admits of the homomorphism with one equivalence class is unsolvable. Therefore the more general problem of deciding whether P admits of a given arbitrary finite range homomorphism is also unsolvable. □

2.5. Some Properties That Are Regular for Any k-terminal recursive Family

In the previous section, we showed that it is impossible to effectively build a finite range homomorphism for an arbitrary computable property. In this section, we show how to build finite range homomorphisms for certain regular properties which are described in terms of certain type of algorithms. In particular, we show that properties such as *independence*, *domination*, *irredundance*, and *vertex j-independence* are regular for any k-terminal recursive family and we will show how to effectively build the corresponding homomorphisms for any k-terminal recursive family.

Definition: A vertex in a graph × subgraph pair (G, S) is *included* if it belongs to S. A vertex in a graph × subgraph pair is *excluded* if it is not included. An *internal vertex* in a graph is a vertex that is not a terminal. A graph × subgraph pair (G, S) *satisfies* P if S satisfies P in G.

2.5.1. Regularity Of Vertex Independence For Any k-terminal Recursive Family

Definition: An included vertex in a graph × subgraph pair is said to be *independent* if none of its neighbors is included. A vertex that is not independent is called *dependent*. A graph × subgraph pair (G, S) is said to be *independent* if all its included vertices are independent.

Theorem : Vertex-independence is a regular property for any k-terminal recursive family of graphs Γ.

Proof : We assume for now, that all the composition operations of Γ are *uniform*, in that they are defined in the same way for every pair of graphs. The proof for non-uniform composition operations is similar. The only difference is that several extra classes are needed. This is illustrated later by an example. If (G, S) has a vertex which is dependent, then S is not independent in G and the pair (G, S) can never yield an independent graph × subgraph pair when involved in a composition operation. Therefore, we group all such graph × subgraph pairs into REJ which is a rejecting class. Henceforth, we assume that all internal vertices in (G, S) are independent. Recall that, when composing two graphs from a k-terminal recursive family, the only vertices involved in the composition are the k terminals of each graph. Hence an included, independent, internal vertex in a graph × subgraph pair remains independent in any composition involving the pair. This means that the only information we need to know to determine the independence properties of a graph × subgraph pair is which of its k terminals are included and whether the graph × subgraph pair is independent. This information is enough to determine the independence properties of the pair when it is involved in a composition operation. There are at most 2^k possible ways of including or not including the k terminals of a graph × subgraph pair, so there are at most 2^k accepting classes. So, we need at most $2^k + 1$ classes in total and we have a finite range homomorphism for independence for any k-terminal recursive family. Therefore independence is a regular property for any k-terminal recursive family. □

To effectively build a finite range homomorphism for independence, we first define the set T which has 2^k accepting classes and 1 rejecting class. An accepting class is represented by a length k vector of 0's and 1's. A 1 in the i th position indicates that the i th terminal is included in any subgraph in the class and a 0 indicates that it is not included. We now determine $C_l \bullet_i C_m$ (corresponding to the composition operation o_i in the family). It is clear that $C_l \bullet_i C_m$ = REJ, if o_i adds an edge between any 2 of the $2k$ terminals of C_l and C_m. If such is not the case, we know which k of the $2k$ terminals from C_l are C_m are the terminals of the new class and it is easy to find $C_l \bullet_i C_m$. The initial classes are the classes of all the graph × subgraph pairs on basis graphs. The procedure is best explained

with the help of an example.

Example

Recall from Chapter 1, the definition of **ProtoHalin** graphs:

1) The graph $G = (\{v'\}, \phi)$ is a protoHalin graph; v' is its root and its left and right terminals.

2) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are protoHalin graphs, with terminals (t_1, l_1, r_1) and (t_2, l_2, r_2) respectively, then:

i) If $G_1 = (\{v'\}, \phi)$, then $G = (\{v'\} \cup V_2, E_2 \cup \{(v', t_2)\})$ is a protoHalin graph, with terminals (v', l_2, r_2).

ii) If $G_1 \neq (\{v'\}, \phi)$, then $G = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(t_1, t_2), (r_1, l_2)\})$ is a protoHalin graph with terminals (t_1, l_1, r_2).

ProtoHalin graphs are 3-terminal recursive. There is a single operation o defined on the graphs. o is non-uniform because it behaves differently when $G_1 = (\{v'\}, \phi)$. We handle this non-uniformity by reserving two special accepting classes, namely the class in which the graph consists of a single vertex v and v is included in the subgraph and the class in which the graph is a single vertex v and the subgraph is empty. These are also the initial classes as they are the classes which we get from the basis graph $(\{v'\}, \phi)$. We call these classes C_1 and C_2. In addition to these classes, we also have $2^3 = 8$ accepting classes which we represent by length 3 vectors of 0's and 1's. There is one rejecting class which we call REJ. The homomorphism in the form of a multiplication table (for independence) is shown in the Table 2.2.

•	C_1	C_2	000	001	010	011	100	101	110	111	REJ
C_1	000	011	000	001	010	011	000	001	010	011	REJ
C_2	100	111	100	101	110	111	REJ	REJ	REJ	REJ	REJ
000	000	001	000	001	000	001	000	001	000	001	REJ
001	000	REJ	000	001	REJ	REJ	000	001	REJ	REJ	REJ
010	010	011	010	011	010	011	010	011	010	011	REJ
011	010	REJ	010	011	REJ	REJ	010	011	REJ	REJ	REJ
100	100	REJ	100	101	100	101	REJ	REJ	REJ	REJ	REJ
101	100	REJ	100	101	REJ	REJ	REJ	REJ	REJ	REJ	REJ
110	110	REJ	110	111	110	111	REJ	REJ	REJ	REJ	REJ
111	110	REJ	110	111	REJ	REJ	REJ	REJ	REJ	REJ	REJ
REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ

Table 2.2 Multiplication Table For Independence In protoHalin Graphs

The initial classes corresponding to the basis graphs are C_1 and C_2.

2.5.2. Regularity Of Domination For Any k-terminal Recursive Family

Definition : An excluded vertex in a graph × subgraph pair is said to be *dominated* if at least one of its neighbors is included.

Theorem : Domination is a regular property for any k-terminal recursive family Γ.

Proof : We assume, as we did for vertex independence, that all the composition operations of Γ are uniform. As for vertex-independence, the regularity of domination holds for families which have non-uniform operations. It is easy to see that an internal, excluded, undominated vertex in a graph × subgraph pair will remain undominated in any composition operation involving the pair. Also observe that any vertex that is dominated in a graph × subgraph pair remains dominated after an operation involving the pair because edges are never deleted in an operation. Hence, to determine the domination properties of a graph × subgraph pair (i.e. how the domination properties of a graph × subgraph pair change when it is involved in a composition operation), we only need to know if all internal vertices are dominated and, if so, which terminals are included vertices (and therefore dominated vertices), which are excluded dominated vertices, and which are excluded undominated vertices. There are 3^k classes corresponding to the 3^k ways of classifying the k terminals when all internal nodes are dominated. Of these, the accepting classes are the ones in which all the terminals are dominated. Hence there are 2^k accepting classes and $3^k - 2^k$ rejecting classes when all the internal vertices are dominated. The initial classes are the classes of all graph × subgraph pairs on the basis graphs. We have already seen that if an internal node is undominated, it will remain undominated after an operation involving the pair. Therefore, we only need one rejecting class for such pairs, which we call REJ. This class will always yield itself when involved in a composition operation. Hence we have $3^k + 1$ equivalence classes for domination in any k-terminal recursive family. □

To effectively build the homomorphism for domination, we first choose a representation for the 3^k classes in which all internal nodes are dominated. We represent any of these classes by a length k vector of 0's, 1's and 2's. 0's indicate excluded undominated vertices, 1's indicate excluded dominated vertices and 2's indicate included vertices. The construction of the multiplication table is similar to the one for vertex independence. To illustrate the procedure, we build homomorphism tables for domination in *series–parallel* graphs as defined in Chapter 1.

Definition: The series-parallel graphs are defined in the following way:

1) The graph $G = (\{l, r\}, \{(l, r)\})$ is series-parallel, with left and right terminals $[l, r]$.

2) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are series-parallel graphs, with left and right terminals $[l_1, r_1]$ and $[l_2, r_2]$ respectively, then :

a) The graph obtained by identifying r_1 and l_2 is a series-parallel graph, with l_1 and r_2 as its left and right terminals respectively. This graph is the *series* composition of G_1 and G_2.

b) The graph obtained by identifying l_1 and l_2 and also r_1 and r_2 is a series-parallel graph called the *parallel* composition of G_1 and G_2. This graph has l_1 $(= l_2)$ and r_1 $(= r_2)$ as its left and right terminals respectively.

Series-parallel graphs are therefore, 2-terminal recursive and are defined by two operations s (series) and p (parallel) The multiplication tables for s and p are shown in Tables 2.3.1 and 2.3.2 respectively. The entry '-' in the tables indicates that the graph × subgraph pairs from the corresponding classes cannot be composed because of the incompatibility of the terminals being identified.

s	00	01	02	10	11	12	20	21	22	REJ
00	REJ	REJ	REJ	00	01	02	-	-	-	REJ
01	00	00	02	00	01	02	-	-	-	REJ
02	-	-	-	-	-	-	00	01	02	REJ
10	REJ	REJ	REJ	10	11	12	-	-	-	REJ
11	10	11	12	10	11	12	-	-	-	REJ
12	-	-	-	-	-	-	10	11	12	REJ
20	REJ	REJ	REJ	20	21	22	-	-	-	REJ
21	20	21	22	20	21	22	-	-	-	REJ
22	-	-	-	-	-	-	20	21	22	REJ
REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ

Table 2.3.1 Multiplication Table For Series Operation For Domination In Series Parallel Graphs

p	00	01	02	10	11	12	20	21	22	REJ
00	00	01	-	10	11	-	-	-	-	REJ
01	01	01	-	11	11	-	-	-	-	REJ
02	-	-	02	-	-	12	-	-	-	REJ
10	10	11	-	10	11	-	-	-	-	REJ
11	11	11	-	11	11	-	-	-	-	REJ
12	-	-	12	-	-	12	-	-	-	REJ
20	-	-	-	-	-	-	20	21	-	REJ
21	-	-	-	-	-	-	21	21	-	REJ
22	-	-	-	-	-	-	-	-	22	REJ

Table 2.3.2 Multiplication Table For Parallel Operation For Domination In Series Parallel Graphs

2.5.3. Regularity Of Irredundance For Any k-terminal Recursive Family

Definition : A subset of vertices S in a graph is said to be *irredundant* if no proper subset of S dominates the same set of vertices as S does. An included vertex v in a graph $\times$ subgraph pair is said to be *redundant* if the set of vertices dominated by v can be dominated by other included vertices. A vertex which is not redundant is called *irredundant*. An *immune* vertex in a graph is an internal vertex all of whose neighbors are internal vertices. An internal vertex that is not immune will be termed *vulnerable*.

Theorem : Irredundance is a regular property for any k-terminal recursive family of graphs Γ. As before, we assume that all operations in Γ are uniform.

Proof : It is clear that an internal included redundant vertex cannot become irredundant when involved in a composition operation because it is not connected to any new set of vertices and hence cannot dominate any new set of vertices. The class of all graph $\times$ subgraph pairs with at least one internal included redundant vertex is the rejecting class REJ. Henceforth, we will concentrate on pairs in which all included internal vertices are irredundant. An immune included irredundant vertex in a graph $\times$ subgraph pair can never become redundant when involved in a composition operation. To prove this fact, first notice that, since the given vertex v is irredundant originally, then either v or some vertex in the neighborhood of v must not be dominated by any other included vertex. As v is immune, both v and the vertices in the neighborhood of v are internal and therefore they do not become adjacent to any new vertices when the pair is involved in the composition operation. In either case v still dominates some vertex that cannot be dominated by any other vertex. Hence, in a graph $\times$ subgraph pair all of whose included internal vertices are irredundant, the only vertices that are potentially redundant are the terminals and the vulnerable

vertices. Therefore, we only have to concern ourselves with terminals and vulnerable vertices. An included irredundant vulnerable vertex (adjacent to a terminal v) that is potentially redundant can become redundant if an included vertex becomes adjacent to v. If an included vulnerable vertex is adjacent to an excluded terminal v, and it is the only vertex dominating v, then it could become redundant if an included vertex becomes adjacent to the v.

Let v be an excluded terminal and let u be an included vertex adjacent to v, Let u be irredundant by the virtue of the fact that it is the only included vertex dominating v. If an included vertex w becomes adjacent to v, then w can cause u to become redundant. For each excluded terminal v, we label it as follows :

i) If all vulnerable vertices adjacent to v are excluded, we label v by 0; In this case, no vertex in the neighborhood of v can either become redundant or can force a vertex which becomes adjacent to v, to be redundant.

ii) If there are one or more included vertices adjacent to v, but none of these can be forced to be redundant if a new included vertex becomes adjacent to v, then we label v by 1. In this case, a vertex u that becomes adjacent to v can become redundant.

iii) If there is an included vulnerable vertex u adjacent to v, and if u is irredundant only because it is the only vertex dominating v, then we label v by 2. In this case, u becomes redundant if an included vertex w becomes adjacent to v.

Now, let us assume that v is an included terminal. An included vulnerable irredundant vertex u adjacent to v cannot become redundant when another vertex w becomes adjacent to v. This is because v dominates itself, so u is not irredundant by the virtue of the fact that it is the only vertex dominating v. Hence we do not need to record information about vulnerable vertices adjacent to v.

v is irredundant if it is the only included vertex dominating itself or if it is the only included vertex dominating one of the excluded vertices adjacent to it. If v is the only included vertex dominating one of the vulnerable vertices adjacent to it, then it is not potentially redundant because no new vertex can become adjacent to the vulnerable vertex. Hence an included irredundant terminal is potentially redundant if it is the only included vertex dominating itself or some of its excluded terminal neighbors and if the rest of the vertices in the neighborhood of v can be dominated by other included vertices.

We label each included terminal v as follows :

i) If v is irredundant and is not potentially redundant, we label v by 3.

ii) If v is irredundant but potentially redundant then we label v according to what excluded terminals are dominated by v. (In this case, we know that v is irredundant only because it is dominating some excluded terminals). We label v by a 4 appended by a length k string of 0's and 1's where a 1 in the ith position signifies that v is the only vertex dominating the ith terminal. As v is irredundant, the string cannot have all 0's in it and therefore there are $2^k - 1$ labels of this form.

iii) If v is redundant, we label v by 5.

For each terminal v, we now have $2^k + 4$ *states* which will completely determine the present and future irredundance behavior of v and its neighboring vertices. As already shown, only the irredundance properties of terminals and their neighbors are affected during a composition operation. Since there are k terminals, the total number of equivalence classes resulting from these choices is at most $(2^k + 4)^k = O(2^{k^2})$. (The accepting classes are the ones which do not have 5 in any position in their corresponding vector representation). These equivalence classes characterize only those graph × subgraph pairs in which all internal included vertices are irredundant. We have already grouped the graph × subgraph pairs which have at least one included redundant vertex into the class REJ. Hence, the total number of equivalence classes is at most $(2^k + 4)^k + 1$, and irredundance is a regular property for any k-terminal recursive family. □

We now show how to effectively build a homomorphism (multiplication table) with operation $\bullet_i$ corresponding to the operation o_i in the given family. We represent each of the equivalence classes (except REJ) by a length k vector where the ith position in the vector indicates the label of the ith terminal. It is easy to see that, given a graph × subgraph pair, we can determine its class. We first determine if all the internal included vertices in the pair are irredundant. If not, the class of the pair is REJ. If all the internal vertices are irredundant, then we determine the class of the pair by determining the state of each terminal. The simplest way to construct the multiplication table is the following :

i) For each basis graph, determine the class of each graph × subgraph pair. As the number of basis graphs is finite and each basis graph has a bounded number of vertices, this process requires only finite time. Let T be the set of all equivalence classes. Initially $T = I$. These classes form the set of initial classes I. Select a pair that is a representative of each such class.

ii) For all operations o_i in the family and for all classes $C_l, C_m \in T$, determine the class C of $P = P_l\ o_i\ P_m$, where P_l and P_m are graph × subgraph pairs representing C_l and C_m respectively. Then clearly, $C = C_l \bullet_i C_m$. Determine if $C \in T$. If not, augment T by C and select P to be the representative of C.

iii) Repeat ii) until T cannot be augmented.

The number of equivalence classes is at most $(2^k + 4)^k + 1$, so the algorithm will halt. We have therefore shown how to effectively build a finite range homomorphism for irredundance in any k-terminal recursive family.

We now demonstrate the above procedure on the family of rooted trees. Rooted trees are a 1-terminal recursive family of graphs, so the number of equivalence classes for irredundance in rooted trees is $(2^1 + 4)^1 + 1 = 7$. The multiplication table for irredundance in rooted trees is shown in Table 2.4. Since rooted trees have only one composition operation defined on them, we only have one multiplication table. This table is the same as the table built by ad hoc techniques in [1].

•	0	1	2	3	4<1>	5	REJ
0	0	0	0	1	1	2	REJ
1	1	1	1	1	1	REJ	REJ
2	2	2	2	REJ	REJ	REJ	REJ
3	3	3	REJ	3	REJ	REJ	REJ
4<1>	3	4<1>	REJ	5	REJ	REJ	REJ
5	3	5	REJ	5	REJ	REJ	REJ
REJ	REJ	REJ	REJ	REJ	REJ	REJ	REJ

Table 2.4 Multiplication Table For Irredundance In Rooted Trees

Here 0, 1, 2, 3 and 4<1> are accepting classes; 0 and 4<1> are initial classes.

2.5.4. Regularity Of vertex j-independence For Any k-terminal Recursive Family

We have already shown that vertex independence is a regular property for any k-terminal recursive family of graphs. We now show that a generalized version of independence, *vertex j–independence*, is regular for any k-terminal recursive family of graphs. Two vertices in a graph are said to be *vertex j–independent* if the shortest distance (that is the number of edges) between the two vertices is at least j and *vertex j–dependent* otherwise. A subset of vertices in a graph is said to be *vertex j–independent* if the vertices in the set are pairwise j-independent, and *vertex j–dependent* otherwise. Observe that the ordinary independence property is 2-independence. The number of equivalence classes for 2-independence in a k-terminal recursive family was shown to be at most $2^k + 1$.

Clearly, a graph × subgraph pair that is j-dependent can never become j-independent when involved in a composition operation. The class of all graph × subgraph pairs that are j-dependent, will be termed REJ. We now concentrate on graph × subgraph pairs which are j-independent. Observe that, only the vertices in the $j-2$ (including the terminals) that are j-independent can become j-dependent. For each included terminal v, we record the set of terminals which are a distance i from v for all $i \leq j-2$. (Note that all such terminals have to be excluded because otherwise the corresponding graph × subgraph pair is not j-independent.) This information is necessary because j-independence of v is affected by new vertices becoming adjacent to v. So we have $2^{(j-2)(k-1)}$ states for v which completely determines the j-independence behavior of v. We call such states *j-independence states*.

If a terminal v is excluded, then it is automatically j-independent. Hence for each terminal, we record $2^{(j-2)(k-1)}+1$ states. We can record a similar information about the internal vertices, but unfortunately there could be an arbitrary number of such vertices. However, observe that we only need to know whether there exists an internal vertex in a particular j-independence state and not the number of such vertices. There are $2^{2^{(j-2)k}}$ corresponding to this information. As there are k terminals in total, we have $(2^{(j-2)(k-1)}+1)^k)(2^{2^{(j-2)k}})$ classes corresponding to j-independence which have all the graph × subgraph pairs j-independent. We have already classified all graph × subgraph pairs which are j-dependent into REJ. As the number of classes is finite, j-independence is regular for any k-terminal recursive family of graphs Γ. The construction of the multiplication table for vertex j-independence is similar to the the construction of the tables for independence, domination and irredundance.

2.6. Local Properties And Their Regularity

We showed in the last section that certain properties are regular with respect to any k-terminal recursive family. All these properties are *local* in a certain sense. In this section, we give a precise definition of locality of properties and then show that all local properties are regular. Let us examine independence, domination and irredundance more deeply and find out in what sense they are local.

An included vertex v in a graph × subgraph pair is independent if none of the neighbors of v are included. A graph × subgraph pair is independent if *all* of its included vertices are independent. So, to determine if an included vertex is independent, we only need to check the 1-neighborhood (i.e. the immediate neighborhood) of v.

An excluded vertex v in a graph × subgraph pair is dominated if there is an included vertex w in the 1-neighborhood of v. An included vertex is always dominated. A graph × subgraph pair is dominating if all of its

vertices are dominated. Therefore, to check if a vertex v is dominated or not, we only need to check the neighborhood of v.

An included vertex v in a graph × subgraph pair is irredundant if there is an excluded vertex w in the neighborhood of v such that the only vertex that is included in the neighborhood of w is v. A graph × subgraph pair is irredundant if *all* of its included vertices are irredundant. To check if an included vertex is irredundant we only need check the 2–*neighborhood* of v.

The intuitive meaning of locality should now be clear. Independence and domination are 1–*local* properties and irredundance is a 2–*local* property. We now show that it is exactly this local behavior of these properties that makes them regular for any k-terminal recursive family. Before we show the major result of this section, we need some notations and definitions.

We will denote the *neighborhood* of a vertex v, that is, the set of vertices adjacent to v by $N(v)$. The number of included vertices in $N(v)$ will be denoted $inc(v)$ and the number of excluded vertices in $N(v)$ will be denoted $exc(v)$.

Definition : Given a graph × subgraph pair (G, S), we define a vertex property $p_{G,S}$ on $V(G)$ (the set of vertices of G). A vertex property p is said to be *compatible* with a subgraph property P if for all pairs (G, S), $P(G, S)$ is true iff $p_{G,S}(v)$ is true for all vertices $v \in V(G)$. Let l and m be non-negative integers, where m is allowed to be ∞. The *interval* $[l, u]$ is the set $\{i : l \leq i \leq u, i \text{ is an integer}\}$. Let each of I_1, I_2, I_3 and I_4 be a union of a finite number of disjoint intervals. A vertex property p is $\{I_1, I_2, I_3, I_4\}$ 1–*local* if it satisfies the following condition for every graph × subgraph pair (G, S) and every vertex $v \in V(G)$: $p_{G,S}(v)$ is true iff either:

v is included, $inc(v) \in I_1$, and $exc(v) \in I_2$, or

v is excluded, $inc(v) \in I_3$, and $exc(v) \in I_4$.

Examples : The vertex independence property is [0, 0] [0, ∞] [0, ∞] [0, ∞] 1–*local*. The vertex domination property is [0, ∞] [0, ∞] [1, ∞] [0, ∞] 1–*local*.

Definition : A vertex property p is 1–*local* if p is $\{I_1, I_2, I_3, I_4\}$ 1-local for some I_1, I_2, I_3, I_4. A subgraph property P is 1–*local* if it admits of a compatible vertex property p that is 1-local.

Theorem : All 1-local subgraph properties are regular with respect to any k-terminal recursive family of graphs.

Proof : Let P be a 1-local subgraph property and p be its associated compatible vertex property. Let p be $\{I_1, I_2, I_3, I_4\}$ 1-local. Let the least upper bound of I_1 be u_1. If the least upper bound of I_1 is ∞, redefine u_1 to be $q-1$ where q is the lower bound of the interval in I_1 containing ∞. (As I_1 is a union of disjoint intervals, there can only be one interval containing ∞). Define u_2, u_3 and u_4 likewise. Let (G, S) be a graph $\times$ subgraph pair such that the graph belongs to a given k-terminal recursive family of graphs Γ. Then it is clear that all internal vertices in the pair that satisfy p (with respect to the pair) will continue to satisfy p when the pair is involved in a composition operation involving the family. Conversely, if an internal vertex does not satisfy p in the pair then it will never satisfy p when the pair is involved in a composition operation. Consequently, as P is satisfied for the pair only if all the vertices in the pair satisfy p, all pairs that have at least one vertex not satisfying p can be grouped into one class REJ. Any pair that belongs to REJ can only yield a pair belonging to REJ.

Suppose that all internal vertices in the pair satisfy p. p is true for an included vertex v only if the number of included neighbors of $v \in I_1$ and the number of excluded neighbors of $v \in I_2$. If the number of included neighbors of v becomes greater than u_1, the truth or falsehood of $p_{G,S}(v)$ will not be affected by any other included vertex becoming adjacent to v. Similarly, if the number of excluded neighbors of v becomes greater than u_2, the truth or falsehood of $p_{G,S}(v)$ will not be affected by any other excluded vertex becoming adjacent to v. Hence for each included vertex we need to store the pair (i, j) where i is the number of included neighbors of v if the number of included neighbors is at most u_1 and "*" otherwise and j is the number of excluded neighbors v if the number of such neighbors is at most u_2 and "*" otherwise. There are $(u_1+2)(u_2+2)$ such pairs. Similar arguments can be given when the terminal v is excluded. (We call the pair (i, j) a *p-state* of v since it completely determines the p-behavior of v, that is how $p_{G,S}(v)$ changes when new vertices become adjacent to v). In this case, we need to store $(u_3+2)(u_4+2)$ pairs. Hence for each terminal, we have $(u_1+2)(u_2+2)+(u_3+2)(u_4+2)$ states. As there are k terminals, we have $((u_1+2)(u_2+2)+(u_3+2)(u_4+2))^k$ classes. This then proves that there is a finite range homomorphism corresponding to P. Hence P is regular. We can also effectively build the corresponding finite range homomorphism following a similar procedure as enunciated in the proof of regularity of independence and domination. □

We now define recursively, a more general notion of locality.

Definition : Let j be a positive integer. Let α be a *$(j-1)$-local* vertex property and let each of I_1 and I_2 be a union of disjoint intervals. A vertex property p is said to be *$\alpha I_1 I_2$ j-local* if it satisfies the following condition, for all

graph × subgraph pairs (G, S) and all vertices $v \in V(G)$, $p_{G,S}(v)$ is satisfied iff either:

v is included, and the number of vertices in $N(v)$, satisfying α belongs to I_1.

v is excluded, and the number of vertices in $N(v)$, satisfying α belongs to I_2.

A vertex property p is said to be j-local if it satisfies at least one of the following conditions :

i) p is $\alpha I_1 I_2$ j-local for some α, I_1 and I_2.

ii) p is the intersection of two j-local properties

iii) p is the complement of a j-local property.

It is clear that to determine if a given vertex in a graph × subgraph pair (G, S) satisfies a j-local property, we only need to check $N_j(v)$.

Definition : A subgraph property P is said to be *j–local* if it admits of a compatible j-local vertex property.

Theorem : All 2-local properties are regular with respect to any k-terminal recursive family of graphs Γ.

Proof : The proof is abstraction of the proof of regularity of irredundance for any k-terminal recursive family. Let P be a 2-local property and let p be a 2-local vertex property that is compatible with P. Let p be $\alpha I_1 I_2$ 2-local, where α is a $\{I_3, I_4, I_5, I_6\}$ 1-local vertex property. Let u_1 be the least upper bound of I_1. If $u_1 = \infty$, redefine u_1 to be $q - 1$, where q is the lower bound of the interval (in I_1) containing ∞. If I_1 is empty, define u_1 to be -2. Define u_2, u_3, u_4, u_5, u_6 likewise. Let (G, S) be a graph × subgraph pair where the graph belongs to Γ. It is clear that any internal vertex which satisfies α in (G, S) will continue to satisfy α when (G, S) is involved in a composition operation and vice-versa. Also as p is 2-local, an immune vertex which satisfies p in (G, S) will continue to satisfy p when (G, S) is involved in a composition operation and vice-versa. Hence the only vertices for which p is affected are the terminals and the vulnerable vertices(that is the vertices in the neighborhood of the terminals), when (G, S) is involved in a composition operation.

Let v be an included terminal in (G, S). Similar arguments hold when v is an excluded terminal. We now determine how $p_{G,S}(v)$ is affected when (G, S) is involved in a composition operation. Let us record the following information about each included terminal v :

i) the α-state of v. There are $m = (u_3 + 2)(u_4 + 2) + (u_5 + 2)(u_6 + 2)$ such states. (as in the proof of regularity of 1-local properties).

ii) i, where i is the number of internal vertices in $N(v)$ satisfying α, if the number of such vertices is at most u_1 and * otherwise. There are $u_1 + 2$ such i's.

iii) set of terminals that are adjacent to v. There are at most 2^{k-1} such sets.

We record similar information about v when v is an excluded terminal (we replace u_5 by u_6 in ii). Hence, we have $(u_1 + u_2 + 4)m2^{k-1}$ states corresponding to each terminal. We call these states the *p−states* of v. This information completely determines the p-behavior of a terminal v, because ii) determines the number of internal vertices satisfying α (as α is 1-local, α for an internal vertex w does not change when new vertices become adjacent to v). Also, as we record the α-state of each terminal (by i) and the terminals that are adjacent to v, we know completely how the α-behavior of each of the vertices in $N(v)$ changes, when new vertices become adjacent to the terminals in (G, S). Thence, we know completely, the p-behavior of v.

We can record similar information about a vulnerable vertex w. (except that we do not have to record the α-state of w because the α-behavior of w does not change during a composition operation involving (G, S)). This gives us $t2^k$ p-states (where $t = u_1 + u_2 + 4$) for w. Unfortunately, there could be arbitrary number of vulnerable vertices in (G, S). However, note that, for each of these $t2^k$ p-states, we only need to record whether there exists a vulnerable vertex in that state). (P is satisfied for (G, S) only if all the vertices in the pair satisfy p. Hence a single vulnerable vertex that does not satisfy $p_{G,S}$ will witness the falsehood of P). There are 2^{t2^k} possible cases corresponding to this information. We have already shown that $tm2^{k-1}$ p-states are enough for a terminal. As there are k terminals, we have $(tm2^{k-1})^k 2^{t2^k}$ equivalence classes for P. Hence P is a regular property and the multiplication table for P can be constructed in the same way as that for irredundance. □

The theorem on regularity of 2-local properties does not easily generalize to j-local properties for arbitrary j. The reason is that, for a vertex v to satisfy a j-local property p, a certain number of vertices in $N(v)$ should satisfy a $(j-1)$-local property α. Hence if $j > 2$, vertices which become adjacent to v not only affect p but also α of the vertices in $N(v)$. We think that this is but a technical problem and with a little tedium, we may be able to overcome it.

Conjecture: All j-local properties are regular with respect to all k-terminal recursive families of graphs.

2.7. Edge-Induced Subgraph Properties

So far, in this chapter, we have only discussed subgraph properties where the subgraph is a vertex subset. The BLW theory also extends to edge-induced subgraph properties. Before describing our results concerning edge-induced subgraphs, we give a brief survey of the results (edge-induced subgraph properties) obtained by Bern, Lawler and Wong [1].

Bern, Lawler and Wong [1] consider the problem of finding a minimum cardinality maximal matching in rooted trees. The problem asks for a smallest set of edges with disjoint endpoints to which it is impossible to add another edge without violating the disjointness constraint. Minimum cardinality maximal matching is NP-hard for general and bipartite graphs [8]. Bern, Lawler and Wong show that the maximal matching property is regular with respect to the class of rooted trees by constructing a finite range homomorphism.

Since the composition operation o for rooted trees creates a new edge, the extension of o from the class of trees Γ to the set of tree $\times$ subgraph pairs Γ_H must specify whether or not the new edge is chosen to be in the subgraph. This extension is accomplished by creating two special operations for Γ_H : in o_1, the new edge is not chosen, and in o_2, the new edge is chosen. (If an edge is chosen to be in the matching then both its endpoints must also be chosen.)

The equivalence classes for the maximal matching property are as follows :

i) C_1 is the set of those pairs (G, H) in which the root r is unmatched and H is a maximal matching for G.

ii) C_2 is the set of those pairs in which r is matched and H is a maximal matching for G.

iii) C_3 is the set of those pairs in which r is unmatched, H is a matching but not a maximal one and H is a maximal matching for the subgraph of G induced by the vertices in $V(G)-\{r\}$.

iv) C_4 is the set of of remaining pairs.

The multiplication tables for the operators $\bullet_1$ and $\bullet_2$ corresponding to composition operations o_1 and o_2 are given in the tables shown below.

$\bullet_1$	C_1	C_2	C_3	C_4
C_1	C_3	C_1	C_4	C_4
C_2	C_2	C_2	C_4	C_4
C_3	C_3	C_3	C_4	C_4
C_4	C_4	C_4	C_4	C_4

Table 2.5 a) Multiplication Table For Maximal Matching In Rooted Trees For o_1

$\bullet_2$	C_1	C_2	C_3	C_4
C_1	C_2	C_4	C_2	C_4
C_2	C_4	C_4	C_4	C_4
C_3	C_2	C_4	C_2	C_4
C_4	C_4	C_4	C_4	C_4

Table 2.5 b) Multiplication Table For Maximal Matching In Rooted Trees For o_2

The only important difference between the techniques for vertex subset properties and edge-induced subgraph properties is in the extension of composition operators from the class of decomposable graphs Γ to the set of pairs Γ_H. To accomplish this extension for edge-induced subgraph properties, it may be necessary to replace a single composition operation o with as many as 2^l operators on Γ_H, where l is the number of attachment edges created by o. The definitions of homomorphism and regular property are unchanged except that the condition 2) in the definition of homomorphism must hold for every one of the enlarged set of composition operations.

In the remainder of this section, we will prove that *matching* and *cycle* are regular properties for any k-terminal recursive family of graphs. As regular properties are closed under the max operation [1], the regularity of matching for any k-terminal recursive family implies the regularity of maximal matching for any k-terminal recursive family of graphs. It follows directly from these results, that minimum cardinality maximal matching and maximum and minimum length cycles can be found in linear time for any k-terminal recursive family provided the input is given in terms of the parse tree. In general graphs, the problem of finding a minimum cardinality maximal matching and the problem of finding a maximum length cycle are NP-hard [9].

Theorem : Matching is a regular property for any k-terminal recursive family of graphs.

Proof : Let Γ be a k-terminal recursive family of graphs. Let (G, H) be graph $\times$ subgraph pair where $G \in \Gamma$ and H is an edge-induced subgraph of G. Consider a composition operation o in the family which introduces l attachment edges when applied to two graph $\times$ subgraph pairs. Since attachment edges can only be added between terminals, $l \leq k^2$. We extend o to graph $\times$ subgraph pairs by replacing o by 2^l composition operations one for each possible

subset of attachment edges in the subgraph. It is clear that if (G, H) has an improperly matched vertex v, v can never become properly matched when (G, H) is involved in in one of the 2^l composition operations. Hence we group all graph $\times$ subgraph pairs which contain an improperly matched vertex into REJ. Now, if all the vertices in (G, H) are properly matched, then a properly matched internal vertex v will remain properly matched when the (G, H) is involved in a composition operation. Hence, we only need information about terminals to determine the matching behavior of (G, H). For each terminal v, we need to know if the number of edges incident on v and belonging to H is 0 or 1. If the number of such edges is 0, then we know that v is available to be matched; otherwise it is not. As the total number of terminals is k, the number of equivalence classes is at most 2^k. Given this information, we can effectively build the 2^l multiplication tables corresponding to o in a way similar to the way as we built the tables for vertex subset properties. If Γ has more than one composition operation, the construction generalizes in an obvious way. Hence we have shown that matching is a regular property for any k-terminal recursive family of graphs. □

It is interesting to note that matching is a *local* property in an intuitive sense similar to the notion of locality for vertex subset properties. To determine if a vertex v is properly matched, we only need to look at the edges incident on v. If there is more than one edge in the subgraph that is incident on v, then v is not properly matched; otherwise v is properly matched.

Definition : Let (G, H) be a graph $\times$ subgraph pair. A vertex $v \in V(G)$ is said to be *cyclic* if the degree of v in H $d_H(v)$ is 2. A vertex v is said to be *semicyclic* if $d_H(v) = 1$. A vertex v is said to be *non−cyclic* if $d_H(v) = 0$ A property P is said to be a *cycle* property if $P(G, H)$ is true iff H is a cycle in G for all graph $\times$ subgraph pairs (G, H).

Theorem : The cycle property is regular with respect to all k-terminal recursive families of graphs.

Proof : Define G, H, Γ and o as in the previous proof. Let us assume that o introduces l edges during a composition operation. As before $l \leq k^2$, we replace o by 2^l operations when we extend o to graph $\times$ subgraph pairs. It is clear that if there exists a vertex $v \in V(G)$ which is neither cyclic, nor semicyclic nor non-cyclic (that is $d_H(v) \geq 3$), then (G, H) can never satisfy the cycle property when involved in a composition operation. Also if an internal vertex is semicyclic, (G, H) can never satisfy the cycle property. We group all pairs which have such vertices into the class REJ. Now, for all remaining pairs, all internal vertices are either cyclic or non-cyclic and all

terminals are cyclic, semicyclic or non-cyclic. It is easy to see that if all the vertices in (G, H) are either cyclic or non-cyclic, H is either a cycle or a union of disjoint cycles, or H is empty. We group all pairs of first kind (that is, H is a cycle) into the only accepting class ACC, group the pairs of the second kind (i.e., H is a union of cycles) into REJ, and group the pairs of the third kind into class C. It is clear that if more edges are added to H when $(G, H) \in$ ACC, the resulting graph × subgraph pair will belong to REJ. In graph × subgraph pairs which contain semicyclic terminals, the semicyclic terminals occur in pairs in the sense that for every semicyclic terminal v, there exists a semicyclic terminal w such that there is a path from v to w in H (If this is not the case, the path from v in H ends in a semicyclic *internal* vertex which is a contradiction.) Two semicylic terminals which are connected by a path in H are called *mutually amicable*. Among k terminals, there can be $2m$ semicyclic terminals where $m \leq \lfloor k/2 \rfloor$. It is easy to see that for (G, H) not in REJ, ACC or C, if we know all the amicable pairs of terminals and if we know all the cyclic terminals (and hence all the non-cyclic terminals), then we can completely determine the cycle behavior of (G, H). Hence, an equivalence class (except REJ or ACC or C) will be represented by the information which determines all pairs of amicable terminals and all cyclic terminals. The number of ways in which we can choose m pairs of amicable terminals out of k terminals is $\frac{k!}{(2!)^m(k-2m)!m!} = \frac{k!}{(2)^m(k-2m)!m!}$. Of the rest $k-2m$ terminals, there are $2^{(k-2m)}$ ways of choosing all sets of cyclic terminals. The number of equivalence classes of all graph × subgraph pairs which have $2m$ semicyclic terminals is $\frac{k!}{2^m(k-2m)!m!}2^{(k-2m)}$. So, the total number of equivalence classes $= \sum_{m=1}^{\lfloor k/2 \rfloor}(\frac{k!}{2^m(k-2m)!m!})2^{(k-2m)} + 3$. Since the number of equivalence classes is finite, the cycle property is a regular property with respect to any k-terminal recursive family of graphs. □

2.8. Vertex Partition Problems

The BLW theory as presented in [1], can be applied only to subgraph problems. In this section, we show that vertex partition problems like chromatic number can be solved by using a simple extension to the BLW theory. A vertex partition problem has the following form:

Given a graph $G = (V, E)$, find a function $f: V \rightarrow \{1, \ldots, |V|\}$ such that the graph labeled by f satisfies a certain property P and cardinality of the the set $f(V)$ is optimized (maximized or minimized). Here P is called the *property* of the problem.

Definition: Let $f: D \rightarrow R$ be a function. and let $S \subseteq D$, then $f(S)$ is defined to be the set $\{ f(d): d \in D \}$. Let

$f_1: D_1 \rightarrow R_1$ and $f_2: D_2 \rightarrow R_2$ be functions where D_1 and D_2 are disjoint. then the *union* of functions f_1 and f_2, $f_1 \cup f_2$ has the domain $D_1 \cup D_2$ and the co-domain $R_1 \cup R_2$ and is defined as follows :

$f_1 \cup f_2(d) = f_1(d)$ if $d \in D_1$ and

$f_1 \cup f_2(d) = f_2(d)$ if $d \in D_2$

The chromatic number problem could be stated in this form as :
Given a graph $G = (V, E)$, find among all functions $f: V \rightarrow \{1, \ldots, |V|\}$ such that $f(v_1) \neq f(v_2)$ if there is an edge between v_1 and v_2 in G and such that the cardinality of $f(V)$ is minimized.

Definition: Let $G = (V, E)$ be a graph. A function $f: V \rightarrow \{1, \ldots, |V|\}$ is a *bounded labeling* for G, if $1 \leq f(v) \leq l$ for all $v \in V$, for some fixed positive integer l. A vertex partition problem restricted to bounded labelings is called a *bounded partition problem.*

Let G be a graph in a k-terminal recursive family Γ. Let G be labeled by a bounded labeling f. We denote G so labeled by (G, f) and call it a graph-function pair. We extend a composition operation o in Γ to all pairs (G, f) in the natural way:
$(G_1, f_1)\ o\ (G_2, f_2) = (G_1\ o\ G_2, f_1 \cup f_2)$.

Graph-function pairs are more general than graph × subgraph pairs, because we could think of a graph × subgraph pair as a graph-function pair where the function labels the vertices of the graph with 0's and 1's. A vertex that is labeled 1 is in the subgraph and a vertex that is labeled 0 is not in the subgraph. The definitions of a *property* and *regularity of a property* for graph-function pairs are similar to the definitions for graph × subgraph pairs.

Theorem: Let Γ be a k-terminal recursive family. There is a linear time algorithm that solves the bounded partition problem for Γ, if the property P of the problem is regular with respect to Γ and if the input to the algorithm is given in terms of a parse tree for a graph.

Proof: Since P is regular, there is finite range homomorphism h for P. We will use a slightly modified version of the generic linear time algorithm of the BLW theory to solve the bounded partition problem for P. At each leaf of the parse tree (which is a basis graph in Γ), we determine an optimal f (that is an f that optimizes $f(V)$) that satisfies P. At internal nodes of the parse tree, we use the same procedure as in the BLW theory except that instead

of storing the optimal cardinality of a subgraph, we store the optimal $f(V)$ corresponding to each equivalence class. As the given problem is a bounded partition problem, $|f(V)|$ is bounded by a fixed constant. Hence each propagation step in the generic algorithm of the BLW theory can be done in constant time. □

2.8.1. Regularity Of Coloring With Respect To All k-terminal Recursive Families Of Graphs

To check whether a given f is a proper coloring of a graph G, we can independently look at each vertex v and determine whether it has the same color as one of its neighbors. If there is any such v in (G, f), then f is not a proper coloring for G. Hence, in a certain sense, coloring is a local property. Although we have not defined locality for vertex partition problems, our discussion of locality on vertex subsets suggests that coloring is a regular property. But before we prove the regularity result for coloring, we have to show that the minimum coloring problem (i.e. the chromatic number problem) is a bounded partition problem for any k-terminal recursive family of graphs.

Lemma: Let l be the maximum of the chromatic numbers of all basis graphs in a k-terminal recursive family of graphs Γ. Then the chromatic number of any $G \in \Gamma$ is at most $\max\{l, 2k\}$.

Proof : We first show that, if all of the basis graphs in a k-terminal recursive family Γ are $2k$-colorable, then all graphs in Γ are $2k$-colorable. Then, the lemma follows from this fact. Let $G \in \Gamma$, and let o be a composition operation in the family. The proof is by induction on the number of composition operations used to build G. Since all the basis graphs are $2k$-colorable, the basis step is true. For the induction step, observe that, if $G = G_1 \, o \, G_2$, and if G_1 and G_2 are $2k$-colorable, then G is also $2k$-colorable because edges are only added between the $2k$ terminals of G_1 and G_2 when G is composed from G_1 and G_2.

From the above lemma, the following theorem is evident:

Theorem: Minimum coloring is a bounded partition problem.

Definition : Let (G, f) be a graph-function pair in Γ. Then a vertex $v \in V(G)$ is said to be *properly colored* if $f(v) \neq f(v')$ for each neighbor v' of v. Clearly (G, f) is properly colored iff all its vertices are properly colored otherwise v is *improperly colored.*

Theorem: Coloring is a regular property for any k-terminal recursive family of graphs Γ.

Proof: Let l be the maximum of chromatic numbers of all the basis graphs in Γ and let $m = \max\{l, 2k\}$. Let (G, f) be a graph-function pair, where $G \in \Gamma$. A properly colored internal vertex v in (G, f) will remain properly colored when (G, f) is involved in a composition operation. An improperly colored vertex (terminal or an internal vertex) v remains improperly colored when (G, f) is involved in a composition operation. Hence, we group all graph-function pairs which have at least one vertex that is improperly colored into REJ. Now consider graph-function pairs (G, f) all of whose vertices are properly colored. Again a properly colored internal vertex remains properly colored when (G, f) is involved in a composition operation. Therefore, we only need to know the colors assigned to the terminals to completely determine the coloring behavior of (G, f). As G is m-colorable, there are only m states for each terminal. Since there are k terminals, this gives m^k accepting classes. Hence there are $m^k + 1$ classes in total. We have thus proved that coloring is a regular property. It is now easy to effectively build the multiplication tables for coloring from this information. □

2.8.2. Regularity Of Domatic Labeling For k-terminal Recursive Families

Definition : A function $f : V \rightarrow \{1, \ldots, |V|\}$ is called a *proper domatic labeling* for a graph $G = (V, E)$, if for all i, the set $V_i = \{v : f(v) = i\}$ is a dominating set for G.

The domatic number problem can be stated as:
Given a graph $G = (V, E)$, determine a proper domatic labeling f for G such that the cardinality of the set $f(V)$ is maximized. This maximum cardinality is called the *domatic number* of G.

It is known that the domatic number problem is NP-hard for general graphs [9]. We can see that any simple graph in a k-terminal recursive family has $O(n)$ edges where n is the number of vertices in the graph. For any multigraph in the family, the simple graph formed by suppressing all multiple edges has $O(n)$ edges.

Theorem: The domatic number of a graph belonging to a k-terminal recursive family Γ is at most l for some fixed constant l.

Proof: Let $G = (V, E)$ be a graph with n vertices. If G is a multigraph, the domatic number of G is same as the domatic number of the graph constructed from G by suppressing all multiple edges. Hence, we will only consider graphs which do not have multiple edges. The number of edges in such a graph is at most $(m + k^2)n$, where m is

the maximum number of edges in any basis graph of the family. Let f be a proper domatic labeling for G and let the cardinality of $f(V)$ be d. Since each vertex v has to be dominated by at least one vertex labeled by i for all $1 \leq i \leq d$ and since v dominates itself, $degree(v) \geq d-1$. Therefore, the number of edges in the graph is at least $\frac{(d-1)}{2}n$. Hence we have $\frac{(d-1)}{2}n \leq (m+k^2)n$ which implies $d \leq 2(m+k^2)+1 = l$. As f was chosen to be any proper domatic labeling, the domatic number of G is at most l. □

Corollary: The domatic number problem is a bounded partition problem.

Theorem: Proper domatic labeling is a regular property for any k-terminal recursive family of graphs Γ.

Proof: Let $G = (V, E) \in \Gamma$ and let $f: V \rightarrow \{1, \ldots, l\}$ be a bounded labeling on G (where $l = 2(m+k^2)+1$ as in the previous theorem). If (G, f) is properly domatic labeled, then $f(N(v)) = f(V)$ for all vertices $v \in V$. (If (G, f) is properly domatic labeled, then each vertex v in G should be such that it is dominated by at least one vertex of each label in $f(V)$ and therefore the 1-neighborhood of v should be labeled by all the labels in $f(V)$.) Let $IN(v) = N(v) \cup \{v\}$. For an internal vertex v, $f(IN(v))$ does not change when (G, f) is involved in a composition operation. We know that for any bounded domatic labeling, $|f(IN(v))| \leq |f(V)|$ for all vertices $v \in V$. Also $|f(V)|$ can never decrease during a composition operation involving (G, f). (vertices are never deleted during a composition operation, hence the cardinality of $f(V)$ can never decrease during a composition operation.) Hence, if there exists an internal vertex v in (G, f) such that $f(IN(v)) \neq f(V)$, (G, f) is not properly domatic labeled and can never yield a properly domatic labeled graph-function pair. We classify the class of all such graph-function pairs into REJ. Now consider pairs (G, f) such that is such that $f(IN(v)) = f(V)$ for each internal vertex v. For each such (G, f), we need to know $f(V)$ and we need to know $f(v)$ and $f(IN(v))$ for each terminal v in (G, f). This information determines completely the domatic behavior of (G, f). As $f(V) \subseteq \{1, \ldots, l\}$, the number of such $f(V)$'s is bounded by 2^l. Similarly, the number of $f(IN(v))$'s for each terminal v is bounded by 2^l. The number of $f(v)$'s for each terminal v is bounded by l. Hence the total number of equivalence classes corresponding to the property of domatic labeling is bounded by $2^l(l2^l)^k = l^k 2^{l(k+1)}$. Note that we could have done a tighter analysis for the number of equivalence classes as $f(IN(v)) \subseteq f(V)$ for each v. Of these classes, the classes for which $f(IN(v)) = f(V)$ for all terminals v are the accepting classes and the rest are rejecting classes. From this information, it is clear how to build the multiplication table for proper domatic labeling. Hence we have proven that proper domatic labeling is a regular property for any k-terminal recursive family of graphs. □

2.9. Summary

In this chapter, we developed a generalized theory for non-regular properties, showed that it is impossible to effectively construct multiplication tables or finite range homomorphisms for arbitrary computable properties on graph × subgraph pairs, and demonstrated how to effectively build finite range homomorphisms for independence, domination, irredundance and vertex j-independence. We then generalized these results to prove that all local properties are regular. We also showed that edge-induced subgraph properties such as maximal matching and cycle are regular with respect to any k-terminal recursive family of graphs. We extended the BLW theory to vertex partition problems and showed that coloring and proper domatic labeling are both regular with respect to all k-terminal recursive families of graphs.

Bern, Lawler and Wong [1] show that if P and Q are regular properties with respect to a given k-terminal recursive family, then so are P and Q; P or Q; $\max P$ and $\min P$ (A graph × subgraph pair satisfies $\max P$, if it satisfies P and no proper superset of the pair satisfies P. A graph × subgraph pair satisfies $\min P$, if it satisfies P and no proper subset of the pair satisfies P). Given finite range homomorphisms for P and Q, they also show how to effectively build finite range homomorphisms for the 4 properties. We have shown in this chapter how to effectively build finite range homomorphisms for independence, domination and irredundance for any k-terminal recursive family of graphs. Hence we can build finite range homomorphisms for various other properties.

CHAPTER 3

FAST PARALLEL ALGORITHMS FOR REGULAR PROPERTIES

3.1. Introduction

In this chapter, we show how to construct fast parallel algorithms for for finding optimal subgraphs obeying regular properties in k-terminal recursive families of graphs. We adapt the generic linear time sequential algorithm of the BLW theory to yield an $O(log\ n)$ time parallel algorithm using $O(n)$ processors where the input is given in terms of a parse tree and where n is the number of nodes in the parse tree of the graph. The adapted algorithm is also extended to handle the generalized theory of Chapter 2. He [11] has independently obtained fast parallel algorithms for finding minimum covering set (MCS), maximum independent set (MIS), maximum matching set (MMS), and minimum p-dominating set in trees and series parallel graphs by using techniques similar to the ones used by us. However, his results are not as general as ours, as we design a generic parallel algorithm that applies to all regular properties.

In general, the input to the algorithm will not be given in terms of a parse tree. Hence, we need to have fast parallel algorithms for generating parse trees for graphs belonging to a given k-terminal recursive family. Evidently, the procedure for building parse trees for a given family is strongly dependent on the way the family has been defined. So the algorithmic techniques used for generating parse trees for one family may not be applicable to another family. We conclude this chapter with parallel algorithms for building parse trees for rooted trees and series parallel graphs. We have chosen these families because both families can be very simply described and have a simple structure, and linear time sequential algorithms for building parse trees for both the families are known. He [12] has independently designed a fast parallel algorithm for finding parse trees for series parallel graphs. However, he uses a different approach to design such an algorithm.

3.2. Parallel Models Of Computation

We will be assuming an idealized model of parallel computation in which concurrent reads and concurrent writes, when allowed, can be done in constant time. Communication among processors is achieved through a shared global Shared memory models of parallel computation can be divided into the following three categories:

1) **EREW** (Exclusive Read Exclusive Write):

In this model, only one processor can access a particular location in global memory at any given time for read and write operations. However, several processors can access different memory locations simultaneously.

2) **CREW** (Concurrent Read Exclusive Write):

In this model, concurrent reads are allowed, but concurrent writes are not allowed.

3) **CRCW** (Concurrent Read Concurrent Write):

In this model, both concurrent reads and writes are allowed. If different processors try to write different data into the same memory location simultaneously, we get a write-conflict. The conflict can be resolved in one of the following ways:

(1) Require that all processors write the same data into the same location during the concurrent write.

(2) Let an arbitrary processor succeed.

(3) Let the lowest numbered processor succeed.

We will use the CRCW model of parallel computation because this is the most powerful of the three models. There is another reason for using this model. Our algorithm for finding optimal subgraphs obeying regular properties in k-terminal recursive families of graphs is based on a technique called *Parallel Tree Contraction* [4] which uses the CRCW model of parallel computation.

3.3. Parallel Tree Contraction And Applications

Miller and Reif [4] describe a bottom-up technique for computation on trees called **parallel tree contraction**. The basic operation in tree contraction is CONTRACT. CONTRACT is actually a simultaneous application of two operations, RAKE and COMPRESS. RAKE is the operation of removing all of the leaves of a given tree in parallel. If the tree is highly unbalanced, this operation could take $O(n)$ parallel time. Hence a new operation called

COMPRESS is introduced.

Definition : Let $T = (V, E)$ be a rooted tree with n nodes and root r. A sequence of nodes $v_1, \ldots, v_m$ is a *chain* if v_{i+1} is the only child of v_i in T for $1 \leq i < m$, v_m has only one child, and v_m's child is not a leaf. In one parallel step, we *compress* a chain by identifying v_i and v_{i+1} for i odd and $1 \leq i < m$. COMPRESS is the operation on T which compresses all maximal chains of T in one step.

Theorem [4] :

After $\lceil \log_{5/4} n \rceil$ executions of CONTRACT, a tree with n vertices it is reduced to its root.

The naive implementation of COMPRESS requires $O(\log n)$ time since we first determine the parity of each node on a chain by *recursive doubling* and then combine consecutive odd and even nodes pairwise in constant time. Hence, to circumvent this difficulty, Miller and Reif [4] introduce a procedure called *Dynamic Tree Contraction.*

Let T be a rooted tree with node set V of size n and root $r \in V$. Each node which is not a leaf can be viewed as a function for which the children supply the arguments. For each node v with children $v_1, \ldots, v_m$, we set aside k locations $l_1, \ldots, l_m$ in common memory. Initially each l_i is empty or *unticked*. When the value of v_i becomes known, it is assigned to l_i : this is denoted by *tick* l_i. Let $Arg(v)$ denote the number of unticked l_i's associated with v. Initially, $Arg(v) = m$, the number of children of v. Let $P(v)$ be the storage location of parent of v and let $node(P(v))$ be the node associated with the storage location $P(v)$. The following describes one Dynamic Contraction Phase. 1) implements RAKE and 2) implements COMPRESS by recursive doubling.

Procedure Dynamic Tree Contraction

In Parallel for all $v \in V - \{r\}$ **do**

1) If $Arg(v) = 0$ **then** *tick* $P(v)$ and delete v

2) If $Arg(v) = Arg(node(P(v))) = 1$ **then** $P(v) \leftarrow P(node(P(v)))$

od

Theorem [4]: In at most $\lceil \log_{5/4} n \rceil$ applications of dynamic tree contraction, a tree is reduced to its root.

Observe that many nodes are not evaluated, that is, for many v, $Arg(v)$ is never set to 0 during any stage of Dynamic Tree Contraction. Miller and Reif [4] define a new procedure called Dynamic Tree Expansion which will allow the evaluation of all nodes so that each node will eventually have all of its arguments after completion of the

procedure. The Dynamic Tree Contraction algorithm can be modified so that each node keeps an initially empty push down store $Store_v$ which is used to record all previous values of $P(v)$. The following statement is added at the start of the block inside the **do** and **od** of Dynamic Tree Contraction :

0) Push value $P(v)$ onto $Store_v$.

Dynamic Tree Contraction is then applied until the root r has all its arguments. Next the procedure Dynamic Tree Expansion is applied until all the nodes have their arguments :

Procedure Dynamic Tree Expansion

In Parallel for all $v \in V - \{r\}$ **do**

1) $P(v) \leftarrow Pop(Store_v)$

2) **if** $Arg(v) = 0$ **then** *tick* $P(v)$.

od

Theorem [4]: At most $\lceil \log_{5/4} n \rceil$ applications of dynamic tree contraction and $\lceil \log_{5/4} n \rceil$ applications of dynamic tree expansion are needed to tick all nodes.

Miller and Reif also give a Randomized Tree Contraction algorithm which works in $O(log\ n)$ parallel time using $O(n/log\ n)$ processors.

3.4. The Generic Parallel Algorithm Corresponding To The BLW theory

We will now describe the generic parallel algorithm corresponding to the BLW theory. We first describe the algorithm on an example. Let us take the example of domination in rooted trees. The table for domination in rooted trees is given below:

•	0	1	2	REJ
0	REJ	0	1	REJ
1	REJ	1	1	REJ
2	2	2	2	REJ
REJ	REJ	REJ	REJ	REJ

Table 3.1 Multiplication Table For Domination In Rooted Trees

The initial classes are 0 and 2 and the accepting classes are 1 and 2. Class 0 is the class of all graph $\times$ subgraph

pairs such that the root is not included in the subgraph and is also not dominated by any other vertex in the subgraph. All the other vertices in these pairs are dominated. Class 1 is the class of all graph $\times$ subgraph pairs such that the root is not included in the subgraph, but it is dominated by some vertex in the subgraph. Class 2 is the class of all graph $\times$ subgraph pairs such that the root is included in the subgraph and all the other vertices are dominated. Let v be a node in the parse tree and let v_l and v_r be its left and right children. The recurrences for minimum cardinality domination are :

$$a_0(v) = a_0(v_l) + a_1(v_r)$$

$$a_1(v) = \min\{a_0(v_l) + a_2(v_r),\ a_1(v_l) + a_1(v_r),\ a_1(v_l) + a_2(v_r)\}$$

$$a_2(v) = \min\{a_2(v_l) + a_0(v_r),\ a_2(v_l) + a_1(v_r),\ a_2(v_l) + a_2(v_r)\}.$$

We do not need a recurrence for REJ because it is a *sink* state and it is rejecting. The vector $\mathbf{a}(v) = \langle a_0(v), a_1(v), a_2(v)\rangle$ represents the optimal cardinality of a dominating set belonging to classes 0, 1 and 2 respectively, in the tree represented by node v in the parse tree.

We can think of the parse tree of a given rooted tree as an expression evaluation tree by interpreting $\mathbf{a}(v)$ at an internal node v of the parse tree to be a function of two arguments $\mathbf{a}(v_l)$ and $\mathbf{a}(v_r)$. When parallel tree contraction is used on this expression tree, RAKE is the operation of removing all leaves of the tree and evaluating (partially or fully) the vector **a** for the parents in terms of the **a** vectors of its leaves. If a node v becomes a leaf after the RAKE operation, $\mathbf{a}(v)$ gets fully evaluated, otherwise it gets partially evaluated. Observe that the sizes of the expression for $\mathbf{a}(v)$ in terms of the size of the expressions for $\mathbf{a}(v_l)$ and $\mathbf{a}(v_r)$ does not increase during RAKE.

Suppose that v_1 and v_2 are two nodes in a chain of the parse tree and let v_2 be the right child of v_1. Then $\mathbf{a}(v_1)$ has already been evaluated in terms of the **a** vector of its left child. Assume that v_1 and v_2 get identified during a COMPRESS, that v_3 is the only child of v_2 in the chain, and that v_3 is the left child of v_2. This means that the right child of v_2 has been RAKEd and $\mathbf{a}(v_2)$ has been evaluated in terms of the **a** vector of the right child of v_2. Let the **a** vector of the left child of v_1 be $\langle i, j, k\rangle$ and the **a** vector of the right child of v_2 be $\langle l, m, n\rangle$. Then, we have :

$$a_0(v_1) = i + a_1(v_2)$$

$$a_1(v_1) = \min\{i + a_2(v_2),\ j + a_1(v_2),\ j + a_2(v_2)\}$$

$$a_2(v_1) = \min\{k + a_0(v_2),\ k + a_1(v_2),\ k + a_2(v_2)\}$$

$a_0(v_2) = a_0(v_3) + l$

$a_1(v_2) = \min\{a_0(v_3) + n, a_1(v_3) + m, a_1(v_3) + n\}$

$a_2(v_2) = \min\{a_2(v_3) + l, a_2(v_3) + m, a_2(v_3) + n\}$

When v_1 is identified with v_2, $\mathbf{a}(v_1)$ will be evaluated in terms of $\mathbf{a}(v_3)$. The result is :

$a_0(v_1) = \min\{a_0(v_3) + i + n, a_1(v_3) + i + m, a_1(v_3) + i + n\}$

Let $t_1(v_1) = \min\{a_2(v_3) + i + l, a_2(v_3) + i + m, a_2(v_3) + i + n, a_0(v_3) + j + n, a_1(v_3) + j + m\}$.

Then $a_1(v_1) = \min\{t_1(v_1), a_1(v_3) + j + n, a_2(v_3) + j + l, a_2(v_3) + j + m, a_2(v_3) + j + n\}$

Let $t_2(v_1) = \min\{a_0(v_3) + k + l, a_0(v_3) + k + n, a_1(v_3) + k + m, a_1(v_3) + k + n\}$.

Then $a_2(v_1) = \min\{t_2(v_1), a_2(v_3) + k + l, a_2(v_3) + k + m, a_2(v_3) + k + n\}$.

It appears, at first, that during a COMPRESS, the size of the expression for **a** can be squared. However, the size of the expressions can be reduced to constant size as follows:

Let

$r = \min\{i + m, i + n\}$

$s = \min\{i + l, i + m, i + n, j + l, j + m, j + n\}$

$t = \min\{j + m, j + n\}$

$u = \min\{k + l, k + n\}$

$v = \min\{k + m, k + n\}$

$w = \min\{k + l, k + m, k + n\}$

Since i, j, k, l, m, n are known; r, s, t, u, v, w can be computed in constant time. Then, we have :

$a_0(v_1) = \min\{a_0(v_3) + i + n, a_1(v_3) + r\}$.

$a_1(v_1) = \min\{a_0(v_3) + j + n, a_1(v_3) + t, a_2(v_3) + s\}$

$a_2(v_1) = \min\{a_0(v_3) + u, a_1(v_3) + v, a_2(v_3) + w\}$

So, the expressions for $a_0(v_1)$, $a_1(v_1)$ and $a_2(v_1)$, can be bounded by a constant, and the function composition in COMPRESS can be implemented in constant time. Therefore, given a parse tree, we can use dynamic tree contraction to find a minimum dominating set in a rooted tree in $O(\log n)$ time using $O(n)$ processors. The procedure for dominating sets in rooted trees can be generalized to all regular properties in all k-terminal recursive families of

graphs.

Theorem : The cardinality or weight of optimal subsets of vertices obeying any regular property P in a graph belonging to a k-terminal recursive families of graphs can be found in $O(log\ n)$ parallel time using $O(n)$ processors when the input to the algorithm is a parse tree of the graph.

Proof : Without loss of generality, assume that, we want to find maximum cardinality subgraphs obeying P. It is easily seen that the size of the expressions in the recurrences corresponding to a node v in the parse tree of a given graph does not increase during a RAKE of the children of v. Hence, to show that $O(log\ n)$ time is sufficient for finding optimal subgraphs obeying regular properties using $O(n)$ processors, we only need to show that the size of the expressions in the recurrences corresponding to a node v in the parse tree does not increase during a COMPRESS. To show this fact, we use an argument similar to the one in the dominating set example.

Without loss of generality, let v_1 and v_2 be nodes in a chain of the parse tree such that v_2 is the child of v_1. Let us assume that the given regular property P has i classes and that we want to find the maximum cardinality of a subset of vertices satisfying P. At each node v of the parse tree, we store a vector $\mathbf{a}(v) = <a_1(v), \ldots, a_i(v)>$, where $a_j(v)$ represents the cardinality of the optimal subgraph satisfying P when the subgraph is restricted to the graph represented by v. Then as v_1 and v_2 belong to a chain in the parse tree and as v_2 is the child of v_1, v_1 has been evaluated in terms of its other child (which was RAKEd before). Let v_3 be the child of v_2. This implies that the other child of v_2 has been RAKEd and hence $\mathbf{a}(v_2)$ has been evaluated in terms of its other child. In the worst case, each $a_j(v_1)$ for $1 \le j \le i$ is a function of all $a_1(v_2), \ldots, a_i(v_2)$ and each $a_j(v_2)$ for $1 \le j \le i$ is a function of all $a_1(v_3), \ldots, a_i(v_3)$. Then we have the following recurrences :

$a_j(v_1) = \max\{a_1(v_2) + u_{j1}, \ldots, a_i(v_2) + u_{ji}\}$ for all $1 \le j \le i$ where the u_{jl}'s are constants for all $1 \le j, l \le i$

or

$$a_j(v_1) = \max_{m=1}^{i}\{a_m + u_{jm}\} \text{ for all } 1 \le j \le i.$$

and

$a_j(v_2) = \max\{a_1(v_3) + t_{j1}, \ldots, a_i(v_3) + t_{ji}\}$ for all $1 \le j \le i$ where t_{jl} are constants for all $1 \le j, l \le i$.

or

$$a_j(v_2) = \max_{l=1}^{i}\{a_l(v_3) + t_{jl})\} \text{ for all } 1 \le j \le i.$$

When v_1 and v_2 are identified during a COMPRESS, $a(v_1)$ is evaluated in terms of $a(v_3)$ and we get:

$$a_j(v_1) = \max_{m=1}^{i}\{\max_{l=1}^{i}\{a_l(v_3) + t_{ml}\} + u_{jm}\} \text{ or}$$

$$a_j(v_1) = \max_{l=1}^{i}\{\max_{m=1}^{i}\{a_l(v_3) + t_{ml} + u_{jm}\}\} \text{ or}$$

$$a_j(v_1) = \max_{l=1}^{i}\{a_l(v_3) + \max_{m=1}^{i}\{t_{ml} + u_{jm}\}\}.$$

Let $s_{jl} = \max_{m=1}^{i}\{t_{ml} + u_{jm}\}$. Then, we have

$$a_j(v_1) = \max_{l=1}^{i}\{a_l(v_3) + s_{jl}\}.$$

Since all of the t_{jl} and u_{jl} are known constants for all $1 \le j, l \le i$, we can evaluate the expressions $s_{jl} = \max_{m=1}^{i}\{t_{ml} + u_{jm}\}$ in constant time, the expressions for a (v_1) in terms of a (v_3) are of the same size as the expressions for $a(v_1)$ in terms of $a(v_2)$, and the expressions for $a(v_2)$ in terms of $a(v_3)$. Hence, we have shown that the size of the expression for $a(v)$, where v is any node in the parse tree, does not increase during a COMPRESS. □

3.5. Generic Parallel Algorithm For The Generalized Theory

We can modify the generic parallel algorithm of the previous section to yield a parallel algorithm for the generalized theory of Chapter 2. Assume that we have a non-regular computable property P with respect to a given k-terminal recursive family of graphs and we are asked to find maximum or minimum cardinality subset of vertices satisfying P in a given graph G belonging to the family. Further assume that P admits of a generalized homomorphism h_n with $g(n)$ equivalence classes. In the generalized theory, the recurrences are derived from the generalized homomorphism h_n which must be built every time we are given a graph which has a parse tree having n nodes. These recurrences are then propagated up the parse tree in essentially the same way as in the generic algorithm of the BLW theory. The only difference is that the vector stored at each node of the parse tree is of length $g(n)$ rather than being of constant length. Assume that we can build the generalized homomorphism h_n and the corresponding recurrences in constant parallel time using $O(ng^3(n))$ processors. To each node of the parse tree we associate $g^3(n)$ processors. During the RAKE of a leaf node, we need $g^2(n)$ processors to evaluate the parent of the leaf in terms of the leaf. During a COMPRESS, in the parallel algorithm for the BLW theory, we computed the constants r_{jl} for $1 \le j, l \le i$ by finding the optimum over i expressions of constant size where i was the number of

equivalence classes. In the generalized case, the number of equivalence classes is $g(n)$. Finding the maximum or minimum over $g(n)$ numbers can be done in $O(\log g(n))$ time using $O(g(n))$ processors. As there are at most $g^2(n)$ such constants to be evaluated, we need $O(g^3(n))$ processors to do the function composition (required in COMPRESS) in $O(\log g(n))$ time. As there are n nodes in total and as the number of contract operations needed to evaluate the root is $O(\log n)$, we get an $O(\log g(n) \log n)$ time algorithm using $O(ng^3(n))$ processors.

An Example :

We illustrate the above algorithm by finding the cardinality of maximum f-independent sets. Recall from Chapter 2 that the generalized homomorphism h_n for f-independence has $f(n)+1$ classes. There are $f(n)$ accepting classes and 1 rejecting class REJ. Since REJ is a sink class we do not need to store the optimal weight corresponding to this class, so a vector of length $f(n)$ is stored at each node of the parse tree. A representative of the ith accepting class C_i is an f-independent set in which the minimum distance of an included vertex to the root is $i-1$, where $1 \leq i \leq f(n)-1$. The representative of the $f(n)$th accepting class is an f-independent sets and all the included vertices have distance $\geq f(n)-1$ from the root. We can obtain the recurrences from this table in constant time using $O(f^2(n))$ processors. Let $a_j(v)$ be the cardinality (or weight) of the optimal subgraph belonging to the class C_j ($j \leq f(n)$). If we can find each C_l and C_m, such that $C_l \bullet C_m = C_j$ in constant time, then we can find the recurrence corresponding to $a_j(v)$ in constant time if we are allowed to use as many processors as there are such pairs (that is (C_l, C_m)). If a tree $\times$ subgraph pair $(G, S) = (G_1, S_1)\, o\, (G_2, S_2)$, where the included vertex which is nearest to the root is at a distance j from the root in (G_1, S_1) and the included vertex which is nearest to the root is at a distance at least $j-1$ from the root in (G_2, S_2), then the distance of the nearest Therefore, if $j < f(n)$, then all pairs (C_l, C_m) such that $C_l = C_j$, and $C_m = C_k$, for all $k \geq j-1$ satisfy: $C_j = C_l \bullet C_m$. Similarly, if $j < f(n)$,all pairs (C_l, C_m) such that $C_l = C_k$ for all $k > j$ and $C_m = C_{j-1}$, satisfy: $C_j = C_l \bullet C_m$. If $j = f(n)$, then all pairs (C_l, C_m) such that $l \geq f(n)$ and $m \geq f(n)-1$, satisfy $C_j = C_l \bullet C_m$. Hence, we can find the recurrence for $a_j(v)$ in constant time, if we are allowed to use as many processors as there are pairs (C_l, C_m) such that $C_l \bullet C_m = C_j$. There are $f^2(n)$ pairs (C_l, C_m) for all $1 \leq l, m \leq f(n)$. As there is a unique C_j such that $C_l \bullet C_m = C_j$, we can find all the recurrences in constant time using $O(f^2 n)$ processors. Then, we can apply the generic parallel algorithm described in the previous section to yield an $O(\log f(n) \log n)$ time algorithm using $O(nf^3(n))$ processors.

3.6. Parallel Algorithms For Building Parse Trees For Rooted Trees

Recall the recursive definition of rooted trees as described in Chapter 1 :

1 The triple $(\{x\}, \{\}, \{x\})$ is a rooted tree with root x.

2 If $T_1 = (V_1, E_1, r_1)$ and

$T_2 = (V_2, E_2, r_2)$ are rooted trees with roots r_1 and r_2 respectively, then the triple $T = (V, E, r)$ where

$$V = V_1 \cup V_2$$

$$E = E_1 \cup E_2 \cup \{r_1 r_2\}$$

$$r = r_1$$

is a rooted tree.

Let $Parse(v)$ denote the parse tree rooted at vertex v.

Let T be a rooted tree with n vertices and root r. To each vertex v_i, we associate a node $Parse(v_i)$ in the parse tree. This node serves as the pointer to the root of the parse tree of the tree rooted at v_i. To each leaf vertex v_i, we assign a processor and assign $Parse(v_i) \leftarrow v_i$ in parallel. To each vertex v_i which has i_m children $v_1, \ldots, v_m$, we associate i_k processors for a total of $\sum i_m = O(n)$ processors. Then in parallel using $O(\log n)$ time for each vertex v_i, we build a parse tree for the tree rooted at v_i as shown in Figure 3.1.

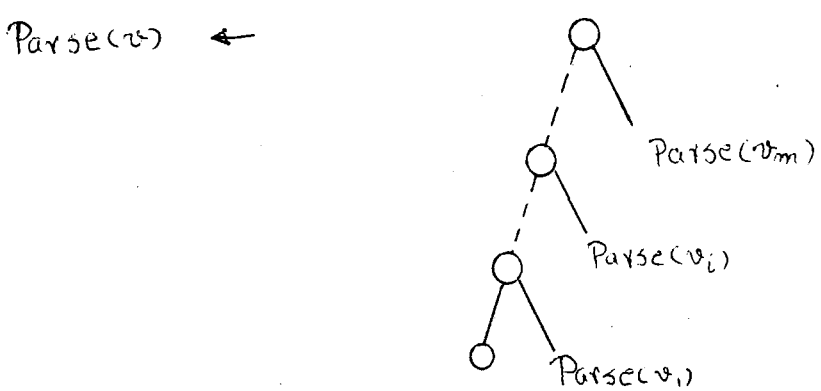

Figure 3.1

3.7. Parallel Algorithm For Finding Parse Trees For Series Parallel Graphs

Recall the definition of series parallel graphs from Chapter 2:

Definition [1]: Series parallel graphs are defined in the following way:

1) The graph $G = (\{l, r\}, \{(l, r)\})$ is series-parallel, with left and right terminals $[l, r]$.

2) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are series-parallel graphs, with left and right terminals $[l_1, r_1]$ and $[l_2, r_2]$ respectively, then :

a) The graph obtained by identifying r_1 and l_2 is a series-parallel graph, with l_1 and r_2 as its left and right terminals respectively. This graph is the *series* composition of G_1 and G_2.

b) The graph obtained by identifying l_1 and l_2 and also r_1 and r_2 is a series-parallel graph called the *parallel* composition of G_1 and G_2. This graph has l_1 $(= l_2)$ and r_1 $(= r_2)$ as its left and right terminals respectively.

Before we describe the parallel algorithm for building parse trees for series parallel graphs, we need some background on the structure of series parallel graphs. There is extensive literature on series parallel graphs. Valdes, Tarjan, and Lawler [10] define *Two Terminal Series Parallel Multidigraphs* (or TTSP multidigraphs) as follows :

Definition :

i) A digraph consisting of two vertices joined by a single edge is a TTSP multidigraphs.

ii) If G_1 and G_2 are TTSP multidigraphs, so is the multidigraph obtained by either of the following operations:

a) *Two terminal parallel composition :* identify the source of G_1 with the source of G_2 and the sink of G_1 with G_2.

b) *Two terminal series composition* : identify the sink of G_1 with the source of G_2.

Notation : Let $G = (V, E)$ be a TTSP multidigraph. A vertex $u \in V$ is an *out−neighbor* of a vertex v if vu is an edge in G. u is an *in−neighbor* if uv is an edge in G . The *outdegree* of v is the number of outneighbors of v. *Indegree* is defined similarly. Let uv be an edge in G. Then u is called the *source* and v the *destination* of uv. The root of the parse tree of the graph which has u as its left terminal and v as its right terminal is called *Parse* (uv).

It is clear from this definition that TTSP multidigraphs are merely the directed versions of series parallel graphs. Alternately, each series parallel graph with left and right terminals l and r respectively can be converted uniquely into a TTSP multidigraph with source l and sink r. Our algorithm for finding parse trees for series parallel graphs will assume the TTSP multidigraph version of series parallel graphs. The sequential algorithm for building parse trees for series parallel graphs uses the transformations for vertices of indegree 1 and outdegree 1 and for multiple edges shown in Figure 3.2.

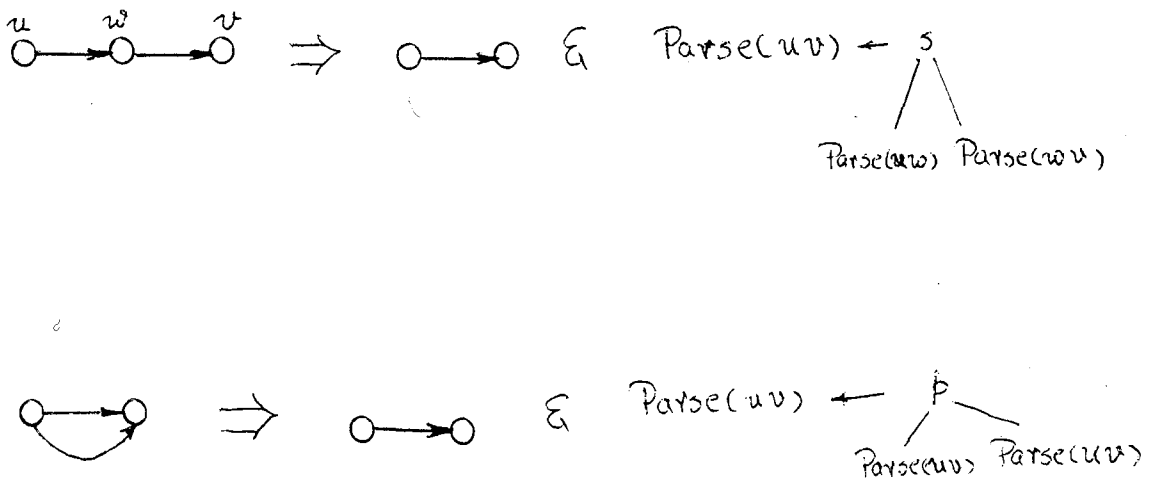

Figure 3.2

Since there is at least 1 vertex of indegree 1 and outdegree 1 or at least one multiple edge [10], each of these reductions reduces the number of edges by 1. If the number of edges in the series parallel graph is m, the time complexity of this algorithm is $O(m)$, so the algorithm is optimal.

3.7.1. The Parallel Algorithm

In this section, we design a parallel algorithm for building parse trees for TTSP multidigraphs. Observe that the straightforward way of parallelizing the sequential algorithm does not yield a fast parallel algorithm. The naive method would try to reduce all indegree 1 outdegree 1 vertices and multiple edges in parallel. This scheme would work if there is an appreciable number of such vertices and edges in the graph. But as the example in Figure 3.3 illustrates, there might be only one vertex of indegree 1 and outdegree 1 or only one multiple edge.

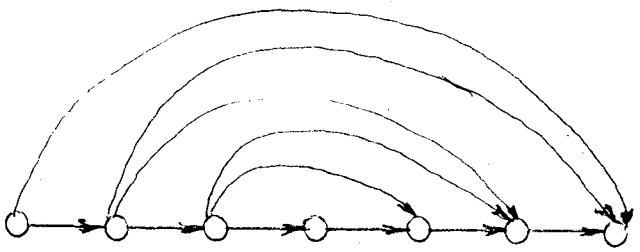

Figure 3.3

In fact, in this example there is only one such vertex or edge after every reduction. Hence the algorithm still requires $\Omega(m)$ time. To find a faster algorithm, we need to find more "structures" in TTSP multidigraphs, which can be easily processed. These "structures" are shown in Figure 3.4. We call such structures *reducible structures*.

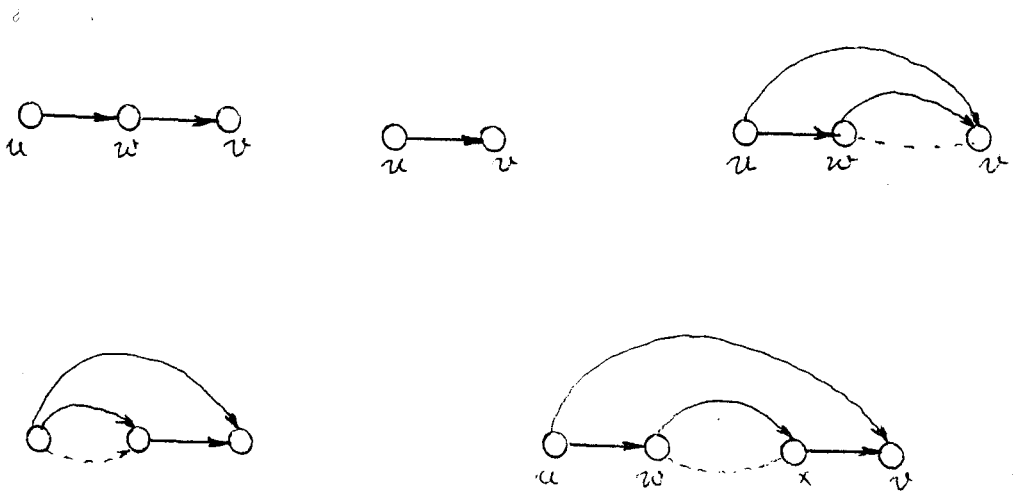

Figure 3.4

Assume that we have a TTSP multidigraph with m edges and n vertices. We reduce this to a TTSP digraph with no multiple edges using m processors in $O(\log m)$ time. We now have a TTSP digraph with $O(n)$ edges.

The input to the algorithm is an adjacency list for a graph. So we have a list of the out-neighbors and in-neighbors of each vertex. We assign a processor to each edge and a processor to each vertex. If e is an edge in G, we initialize $Parse(e) \leftarrow e$. Initially, using $O(m)$ processors, we suppress all the multiple edges and build the corresponding parse trees in $O(\log m)$ time. The resulting digraph has $O(n)$ edges.

We now explain what happens in one step of the algorithm :

We call a vertex of indegree 1 and outdegree 1 a *reducible* vertex. We call a sequence of vertices $v_1, \ldots, v_m$ a *chain* if each of v_i $1 \leq i \leq m$ is reducible, the in-neighbor of v_1 is not reducible, the out-neighbor of v_m is not reducible and v_{i+1} is the out-neighbor of v_i for all $1 \leq i \leq m$. Figure 3.5 gives an example of a chain. A reducible vertex can be eliminated by the transformation shown in Figure 3.2.

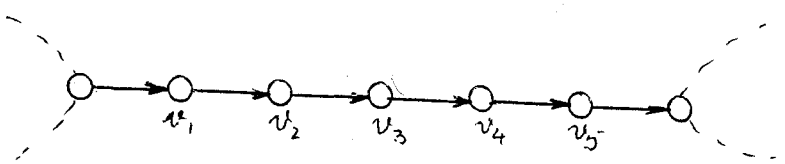

Figure 3.5

It is clear that two vertices u and v which are on a chain such that v is the out-neighbor of u cannot be simultaneously reduced. Hence we need to identify alternate vertices on each chain which are to be reduced. This can be done by recursive doubling in $O(\log n)$ time using $O(n)$ vertex processors. The reduction can then be done in constant time. Each reduction deletes $k-1$ edges in a chain of length k and introduces $\lceil k/2 \rceil$ new edges. There could be more than one parse trees $Parse(uv)$ at any particular time depending on the multiplicity of the edge uv. These parse trees will be all combined by the edge processors into one using the parallel operation p. This can be done in $O(\log k)$ time using k processors where k is the number of parse trees. As k is at most n, the reduction of all reducible vertices can be done in $O(\log n)$ time using $O(n)$ processors.

For each processor uv, we check for the following 3 cases :

i) If there is a reducible structure of the form shown in Figure 3.6, then $outdegree(u) = 2$ or 3 and v is an out-neighbor of both u and one of the other out-neighbors of u. Let us call this other out-neighbor w. Also $indegree(w) = 1$. This case can be checked in constant time.

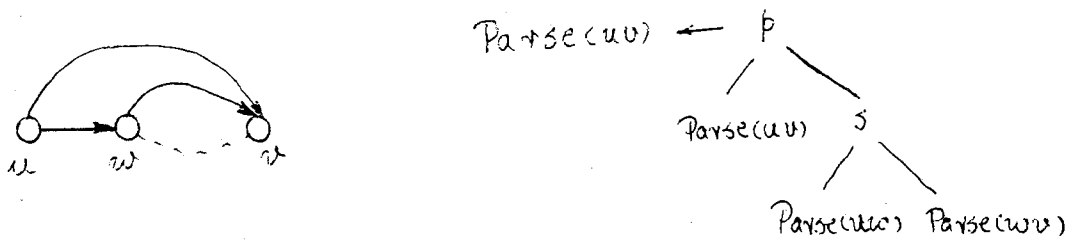

Figure 3.6

We then delete uw and w from the out-neighborhood of u. Deletion of uw creates an articulation point at v. The biconnected component containing v and x is a TTSP multidigraph with w as the source and v as the destination. Hence we need to separately process the component and build its corresponding parse tree *Parse* (xv). To achieve this goal, we will have to identify the component containing v and x. We store the information that v has been created as an articulation point and w is in the biconnected component containing v and defer the identification of the component until all edge processors have finished the procedure.

ii)If there a reducible structure of the form shown in Figure 3.7, then *indegree* (v) = 2 or 3 and there is an edge from u to one of the other in-neighbors of v. Let us call this in-neighbor w. Also *outdegree* (w) = 1. This case is symmetric to case 1 and is handled in a similar way.

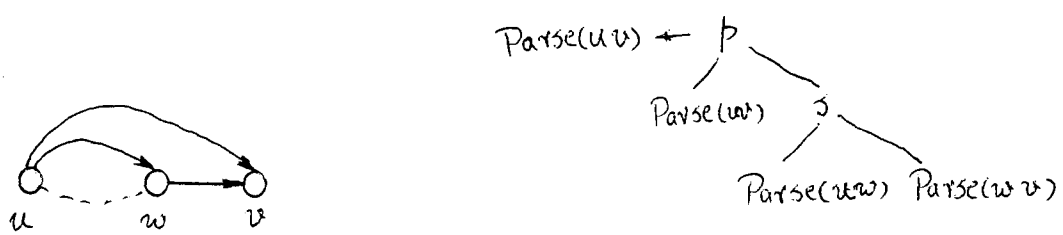

Figure 3.7

iii) Is there a reducible structure of the form shown in Figure 3.8 ? If so, *outdegree* (u) = *indegree* (v) = 2. Let w be the out-neighbor of u and let x be the in-neighbor of v. Then *indegree* (w) = *outdegree* (x) =1. wx is an edge. This case can be checked in constant time. We can update the parse tree *Parse* (uv) in terms of *Parse* (uw), *Parse* (wx), *Parse* (xv) and partially evaluated *Parse* (uv).

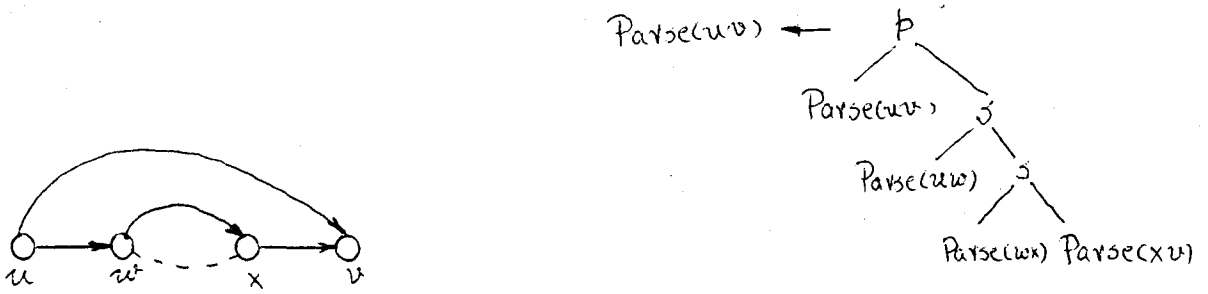

Figure 3.8

We then delete uw and xv and we also delete w from the out-neighborhood of u and x from the in-neighborhood of v. We can then process *Parse* (wx) disjointly from *Parse* (uv).

After every edge processor has executed this step, we find the biconnected components which contain the articulation vertices v and the corresponding vertices w created in cases i) and ii) in the edge-processor execution. Let n be the number of vertices and m the number of edges in the graph. The biconnected components of a general graph can be found in $O(\log n)$ time using $O(n+m)$ processors [5]. As $m = O(n)$ in a TTSP digraph which does not have multiple edges, we can execute the above procedure in $O(\log n)$ time. using $O(n)$ processors.

Hence we see that each step incrementally builds a parse tree *Parse* (uv) for each edge uv of the graph. Also we see that each step requires $O(\log n)$ time using $O(n)$ processors. We conjecture that a TTSP digraph (that is a graph with no multiple edges) with $e = O(n)$ edges has at least $e/10$ structures. To support our conjecture, we have the following evidence:

We claim that the TTSP digraphs shown in Figure 3.9 have the minimum number of reducible structures among all TTSP digraphs with same number of edges.

The TTSP digraphs shown are obtained by recursing over *level number* j. Let the number of edges in these graphs be u_j and the number of reducible structures be r_j. Then we have :

$$u_j = 2u_{j-1} + 5 \text{ and}$$

$$r_j = 2r_{j-1}$$

with $u_0 = 5$ and $r_0 = 1$.

Solving these recurrences yields:

$$u_j = 10(2^j) - 5 \text{ and}$$

$r_j = 2^j$.

Therefore, the ratio $\frac{r_j}{u_j} = \frac{2^j}{10(2^j)-5} \geq \frac{1}{10}$.

The number of edges deleted by the algorithm is at least half the total number of reducible structures. So the number of edges can be reduced by $e/20$ after during each parallel step. If $t(e)$ is the maximum number of parallel steps for a TTSP digraph with e edges (e is $O(n)$), then

$t(e) \leq t((1-1/20)e) + 1$, so

$t(e) = \log_{\frac{20}{19}} e = O(log\ e) = O(log\ n)$.

Hence the algorithm will halt after $O(log\ n)$ steps and the total time for the algorithm is $O(log^2 n + \log m)$ using $O(n+m)$ processors. (This time bound includes the initial preprocessing time for suppressing multiple edges).

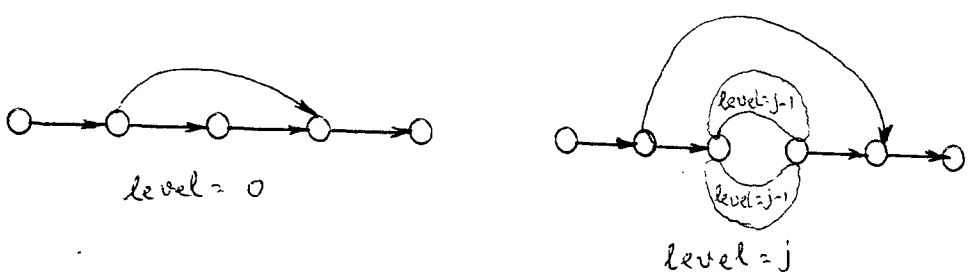

Figure 3.9

CHAPTER 4

CONCLUSIONS AND OPEN PROBLEMS

The BLW theory is a uniform theory in the sense that it provides a uniform framework for designing linear time algorithms for certain graph optimization problems. The framework has two key ideas:k-terminal recursive families and regular properties. However, the theory does not explain why certain properties are regular, and the building of finite range homomorphisms (multiplication tables) is left to human ingenuity. What we have attempted to do in this research is to address these issues. We introduced a notion of locality and showed that all local properties are regular. Hence in essence, we explain why certain properties are regular with respect to certain families of graphs. We have also shown that non-regular properties exist and have modified the BLW theory to handle non-regular properties. The generic algorithm for the generalized theory is not linear (as expected) but is reasonably fast for certain non-regular properties. We also adapted the generic algorithms of the BLW theory and the generalized theory to yield $O(log\ n)$ time parallel algorithms using $O(n)$ processors.

There is one important question that neither the BLW theory nor our research addresses in detail. The generic algorithms for the BLW theory and for the generalized theory (introduced in this research) take as input the parse trees of the graphs belonging to a k-terminal recursive family. Since the input to the algorithms could be given in terms of adjacency lists or adjacency matrices of the corresponding graph instead in terms of parse trees, we need to design fast algorithms for building parse trees for different k-terminal recursive families of graphs. Linear time sequential algorithms for building parse trees for rooted trees and series parallel graphs [10] are known. In this research, we designed an $O(log\ n)$ time parallel algorithm for building parse trees for rooted trees, using $O(n)$ processors. We also designed an $O(log^2 n + \log m)$ time parallel algorithm using $O(m+n)$ processors, for building the parse trees of series parallel graphs with m edges and n vertices. (However, we have not been able to provide the proof of complexity, for the parallel parse tree algorithm for series parallel graphs).

A few open problems emerging as a result of this research are as follows:

1) It is clear that the notion of locality, as introduced in this research, does not completely characterize regularity.

However, it is quite possible that a more general notion of locality could completely characterize regularity. We know that regular properties are closed under operations such as union, intersection, complementation, and max and min operations. So any generalized notion of locality that does not satisfy all of these closure properties is clearly not equivalent to regularity. To actually prove that all regular properties are *local* (in a generalized sense), requires us to develop a stronger version of the Pumping Lemma proved in Chapter 2. The lemma should be strong enough to prove that if a property P is not *local* (in the generalized sense), then it is not regular.

2) Although we have restricted ourselves to vertex subsets when defining locality, it is very likely that such a notion of locality could be extended to edge-induced subgraph properties and to vertex partitioning problems.

3) It is easily seen that any graph in a k-terminal recursive family has $O(n)$ edges (where n is the number of vertices in the graph) unless the graph is a multigraph. The BLW theory and hence our theory, is only applicable to k-terminal recursive families of graphs. A more general theory which can be applied to recursive families of graphs which have more that $O(n)$ edges will be a significant achievement in this regard. A naive approach to such a generalization would be to make the terminal set of arbitrary size. But following this approach, we could characterize the class of all graphs by such a family. Hence this approach will not yield any useful results (Most of the graph optimization problems we deal with are NP-hard for general graphs.) Another approach that could be followed is to restrict the types of composition operations for a family. This approach seems to be a more promising one as we have many ways to restrict the set of composition operations in the family.

4) The BLW theory shows that if a property P is regular with respect to a k-terminal recursive family of graphs Γ, then there is a linear time algorithm for the optimization problem (maximum or minimum cardinality or weight) for P in Γ. We showed in Chapter 2 that there are properties that are not regular, but which have linear time algorithms for solving them (e.g. the property of being a center or a bicenter of a tree). Hence regularity does not completely characterize properties which have linear time optimization algorithms. It would therefore be interesting to investigate a more generalized framework which would facilitate this characterization.

5) As we have already indicated, a uniform theory for developing efficient algorithms for building parse trees for graphs belonging to k-terminal recursive families of graphs would be a breakthrough. The question seems quite difficult because it requires us to determine some structure common to all k-terminal recursive families, before finding efficient algorithms for building parse trees for such families.

BIBLIOGRAPHY

[1] M.W.Bern, E.L.Lawler, A.L.Wong; 'Why Certain Subgraph Computations Require Only Linear Time' , 26th Foundations Of Computer Science Symposium (1985), pp. 117-125

[2] T.V.Wimer, S.T.Hedetniemi, Laskar; "A Methodology For Constructing Linear Graph Algorithms", to appear

[3] Y.Shiloach, U.Vishkin; "An O(log n) Parallel Connectivity algorithm", Journal Of Algorithms 3 (1982), pp. 57-67

[4] G.Miller, J.Reif; "Parallel Tree Contraction And Its Applications", 26th Foundations Of Computer Science Symposium (1985), pp. 478-489 (1985)

[5] R.E.Tarjan, U.Vishkin; "Finding Biconnected Components and Computing Tree Functions In Logarithmic Parallel Time", 25th Foundations Of Computer Science Symposium (1984), pp. 12-20

[6] S.C.Kwan, W.L.Ruzzo; "Adaptive Parallel Algorithms For Minimum Spanning Trees",

[7] J.E.Hopcroft, J.D.Ullman; "Introduction To Automata Theory, Languages and Computation", pp. 201-203

[8] M.Yannakakis, F.Gavril; "Edge Dominating Sets In Graphs", Unpublished Manuscript.

[9] M.R.Garey, D.S.Johnson; "Computers And Intractability: A Guide To The Theory Of NP-Completeness", W.H. Freeman and Co. (1979)

[10] J.Valdes, R.E.Tarjan, E.L.Lawler; "The Recognition Of Series-Parallel digraph", Proc. 11th Annual Symposium On Theory Of Computing, Atlanta,Ga., 1979, pp.1-12

[11] X. He; "Binary Tree Algebraic Computations And Parallel Algorithms For Simple Graphs", Tech. Report (1986); Department Of Computer And Information Science, The Ohio State University.

[12] X. He; "Parallel Recognition and Decomposition Of Two Terminal Series Parallel Graphs", 24th Annual Allerton Conference On Communication, Control, And Computing, 1986.