INFORMATION TO USERS

While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. For example:

- Manuscript pages may have indistinct print. In such cases, the best available copy has been filmed.

- Manuscripts may not always be complete. In such cases, a note will indicate that it is not possible to obtain missing pages.

- Copyrighted material may have been removed from the manuscript. In such cases, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or as a 17"x 23" black and white photographic print.

Most photographs reproduce acceptably on positive microfilm or microfiche but lack the clarity on xerographic copies made from the microfilm. For an additional charge, 35mm slides of 6"x 9" black and white photographic prints are available for any photographs or illustrations that cannot be reproduced satisfactorily by xerography.
On efficient parallel algorithms for solving graph problems

He, Xin, Ph.D.

The Ohio State University, 1987
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages ______
2. Colored illustrations, paper or print ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy ______
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages ______
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) ____________ lacking when material received, and not available from school or author.
12. Page(s) ____________ seem to be missing in numbering only as text follows.
13. Two pages numbered ______. Text follows.
14. Curling and wrinkled pages ______
15. Dissertation contains pages with print at a slant, filmed as received ______
16. Other ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________

University Microfilms International
ON EFFICIENT PARALLEL ALGORITHMS
FOR SOLVING GRAPH PROBLEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By
Xin He, M.S.

* * * *

The Ohio State University
1987

Dissertation Committee:
Y. Yesha, Advisor
E. M. Gurari
T. J. Long

Approved by

Yaacov Yesha
Advisor
Department of Computer and
Information Science
To My Parents
ACKNOWLEDGMENTS

I express sincere appreciation to Dr. Yaacov Yesha, my dissertation advisor, for his guidance and insight throughout the research. Thanks go to the members of my advisory committee, Dr. Eitan M. Gurari and Dr. Timothy J. Long, for their suggestions and comments. Dr. Ming Li and Dr. William F. Ogden deserve my special thanks for their encouragement and interests in my research. My gratitude also goes to Professor Jerome Rothstein, Dr. Bruce W. Weide, and Dr. Neelamegam Soundararajan.

To my wife, Hwa, I offer deep thanks for her unshakable faith in me, for her sacrifices, and for all experience we have shared.

Finally, I thank Department of Computer and Information Science, Graduate School and Office of Research and Graduate Studies of The Ohio State University for financial support.
VITA

April 8, 1953 ............................................. Born - Beijing, China.
1977 - 1978 ............................................... Lanzhow University, Lanzhow, China.
1978 - 1980 ............................................... The University of Science and Technology of China, Beijing, China.
1981 ............................................................ M.S., Mathematics, The Ohio State University.
1984 ............................................................ M.S., Computer and Information Science, The Ohio State University.

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................................................................................ iii
VITA ....................................................................................................................................... iv
TABLE OF CONTENTS .......................................................................................................... vi
LIST OF FIGURES .............................................................................................................. ix

CHAPTER PAGE

I. INTRODUCTION ................................................................. 1
   1.1 Why Parallel Computation .................................................... 1
   1.2 Parallel Computation Model ................................................. 2
   1.3 Parallel Algorithm Design and the NC Complexity Class .... 5
   1.4 Dissertation Organization ..................................................... 9

II. BINARY TREE ALGEBRAIC COMPUTATION ......................... 12
   2.1 Definitions ........................................................................... 12
   2.2 A Parallel BTAC Algorithm ................................................. 15

III. PARALLEL ALGORITHMS FOR SOLVING TREE PROBLEMS 28
   3.1 Input Representation ............................................................ 29
   3.2 Minimum Covering Set Problem ......................................... 31
   3.3 Maximum Independent Set and Maximum Matching Set Problems .............................................. 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Minimum r-Dominating Set Problem</td>
<td>38</td>
</tr>
<tr>
<td>IV.</td>
<td>R-DOMINATING SET AND P-CENTER PROBLEMS ON TREES</td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>Definitions and Background</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>The Outline of the r-Dominating Set Algorithm</td>
<td>54</td>
</tr>
<tr>
<td>4.3</td>
<td>Representation of the Unary Functions</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>Parallel Algorithm for p-Center Problem</td>
<td>68</td>
</tr>
<tr>
<td>V.</td>
<td>RECOGNITION AND DECOMPOSITION OF TWO TERMINAL SERIES PARALLEL GRAPHS</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Definitions</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>Reducing Components of Two Terminal Multidags</td>
<td>81</td>
</tr>
<tr>
<td>5.4</td>
<td>Decomposition of Chain Type TTSP Graphs</td>
<td>87</td>
</tr>
<tr>
<td>5.5</td>
<td>Decomposition of General TTSP Graphs</td>
<td>91</td>
</tr>
<tr>
<td>5.6</td>
<td>Implementation</td>
<td>102</td>
</tr>
<tr>
<td>VI.</td>
<td>PARALLEL ALGORITHMS FOR SOLVING TTSP GRAPH PROBLEMS</td>
<td>111</td>
</tr>
<tr>
<td>6.1</td>
<td>Generalized BTAC Problem</td>
<td>112</td>
</tr>
<tr>
<td>6.2</td>
<td>Weighted Maximum Matching Set Problem</td>
<td>115</td>
</tr>
<tr>
<td>6.3</td>
<td>Maximum Path and Maximum Cycle Problems</td>
<td>120</td>
</tr>
<tr>
<td>VII.</td>
<td>DEPTH FIRST SPANNING TREE PROBLEM IN PLANAR GRAPHS</td>
<td>125</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A chain is broken into two sub-chains</td>
<td>21</td>
</tr>
<tr>
<td>2. The binary function $f_{MCS}$</td>
<td>32</td>
</tr>
<tr>
<td>3. An example of the MCS algorithm</td>
<td>35</td>
</tr>
<tr>
<td>4. A tree $T_1$ is converted to b binary tree $T$</td>
<td>35</td>
</tr>
<tr>
<td>5. Label modification rules</td>
<td>44</td>
</tr>
<tr>
<td>6. The binary function $f_{2-MDS}$</td>
<td>45</td>
</tr>
<tr>
<td>7. An example of the $r$-MDS algorithm with $r = 2$</td>
<td>45</td>
</tr>
<tr>
<td>8. The cycle defining the cyclic order</td>
<td>60</td>
</tr>
<tr>
<td>9. Representation of 1-fold functions</td>
<td>62</td>
</tr>
<tr>
<td>10. Representation of the composite function in case 1</td>
<td>63</td>
</tr>
<tr>
<td>11. Representation of the composite function in case 2</td>
<td>63</td>
</tr>
<tr>
<td>12. The composite function of $F(u)$ and $F(v)$</td>
<td>67</td>
</tr>
<tr>
<td>13. A TTSP graph and its decomposition tree</td>
<td>80</td>
</tr>
<tr>
<td>14. A TTSP graph is decomposed into two reducing components</td>
<td>80</td>
</tr>
<tr>
<td>15. The forbidden subgraph $W$</td>
<td>83</td>
</tr>
<tr>
<td>16. The proof of Lemma 5.3</td>
<td>83</td>
</tr>
<tr>
<td>17. The proof of Lemma 5.4</td>
<td>86</td>
</tr>
</tbody>
</table>
18. The jump tree of a chain type TTSP graph ...........................................88
19. The proof of Lemma 5.5 .........................................................................88
20. The proof of Lemma 5.8 ..........................................................................92
21. The resolution graph ..............................................................................95
22. The proof of Lemma 5.9 ..........................................................................97
23. Constructing a normal spanning tree ....................................................106
24. The multiplication table for the weighted MMS problem ..................117
25. The multiplication table for the MC and MP problems ......................123
26. An illustration of Lemma 7.2 .................................................................129
27. The enclosing spanning tree and the dual tree ....................................134
28. The proof that $T^*$ is connected .........................................................134
29. The proof that $T^*$ has no cycle ..........................................................136
30. The type 2 and type 3 nodes .................................................................140
31. The computation of the first pass ...........................................................140
32. The computation of the second pass ......................................................142
33. The path $P$ divides the boundary of the face $f$ .................................145
34. The construction of the separating cycle ..............................................145
35. Computing the number of children in a tree ......................................157
CHAPTER I
INTRODUCTION

1.1 Why Parallel Computation

During the past decade, due to the rapid development in very large scale integrated circuit technology, the size and the cost of computer hardware components have been decreasing dramatically. Therefore, in order to substantially increase the computing power of modern computers, it is becoming technically and economically feasible to build a computer by putting a large number of similar components together and making them working in parallel. It is no surprise that parallel computing is such an active and dynamic area of research.

The study of parallel computation plays an important role in computer science both theoretically and practically. Theoretically, it is a natural extension of classical complexity theory. The discoveries made in the study of parallel computing will give us a better understanding of the nature of computation. Practically, the results obtained from the study of parallel computing will provide guidance for the design and use of real parallel computers.

The significance of the study in parallel computing is perhaps best stated by Professor R. M. Karp of UC Berkeley (member of National Academy of Science) in his Turing Award interview [Fr]: "I'm extremely interested in parallel computation, and I think it's a fascinating area. There are several strains of
research that have not yet completely come together: There's the study of various parallel architectures; there are many questions about what the processors should look like and how they should be interconnected. There are numerical analysis issues, complexity issues, and algorithm design issues."

This research is concerned with the development of new methods for designing parallel algorithms and the design of efficient parallel algorithms for solving various graph problems.

1.2 Parallel Computation Model

In order to study parallel algorithms quantitatively, we must fix a parallel computation model for our research. Parallel algorithms are designed for running on this specific model. The model must be chosen so that we can define measurements for the quality of the parallel algorithms running on the model and these measurements should reasonably reflect the nature of the parallel computation.

The parallel computation model used in this research is a parallel random access machine (PRAM for short). The model consists of a number of identical processors and a common memory. Each processor is a random access machine which is capable of accessing any common memory cell. In each time unit, a processor can read from a common memory cell, perform an ordinary arithmetic or a logic operation, and write into a common memory cell. Each processor is given a processor number. All processors have the same program (certain statements of the program may depend on the processor number.) The input to the PRAM is given in the common memory. After the computation is finished, the
output is left in the common memory.

According to how the concurrent access to the common memory is resolved, the PRAMs are divided into several sub-models:

(a) **Exclusive Read Exclusive Write PRAM** (EREW PRAM): Neither concurrent read from nor concurrent write into a common memory cell by different processors is allowed.

(b) **Concurrent Read Exclusive Write PRAM** (CREW PRAM): Concurrent read from a common memory cell is allowed. Concurrent write into a common memory cell is not permitted.

(c) **Concurrent Read Concurrent Write PRAM** (CRCW PRAM): Both concurrent read from and concurrent write into a common memory cell are allowed.

In a CRCW PRAM, if several processors try to write into the same common memory cell, we say that a write conflict occurs. According to how the write conflicts are resolved, the CRCW PRAMs are divided into:

(c1) **Common CRCW PRAM**: In a write conflict, the processors trying to write into the same memory cell must write the same value. Otherwise, the content of the memory cell being written is not defined. The programmer must make sure that the programs running on this model satisfy this requirement.

(c2) **Arbitrary CRCW PRAM**: In a write conflict, an arbitrary processor succeeds, but we do not know which one in advance. The programmer must make sure that the results of the computation are correct regardless of which processor succeeds in concurrent writes.
(c3) **Priority CRCW PRAM:** Each processor is indexed by a processor number according to its priority. In a write conflict, the lowest indexed (the highest priority) processor succeeds.

The *resources* used by a parallel algorithm running on a PRAM is measured by two quantities: the number of processors, and the time units needed to solve a problem. These two quantities are very important in practice since the primitive goal of parallel computing is to substantially speed up the computation by increasing the number of processing units. Naturally, we want to solve the problem as fast by using as few processors as possible. These two quantities are also very interesting in the theory of computation: the trade-off between them is of fundamental importance in complexity theory ([CD], [FMRW], [LY1], [LY2], and [FRW]).

The power of these PRAM models can be ordered as follows (in increasing order): EREW, CREW, Common CRCW, Arbitrary CRCW, Priority CRCW. An algorithm running on a lower power model can also run on a higher power model without any change. The reverse, however, is not true. Namely, one step of a higher power PRAM of \( N \) processors might not be simulated in constant time by a lower power PRAM of \( N \) processors. (See [CD], [FMRW], [LY1], [LY2], and [FRW]).

The PRAM models are found very suitable for the study of parallel computation for the following two reasons. First, the PRAM model can be efficiently simulated by other parallel computation models. Thus, once efficient parallel algorithms for the PRAM models have been developed, they can be converted into efficient algorithms for other parallel computation models. Second, the
power of the PRAM models allows us to concentrate on intrinsic properties of the problems to be solved. It is these intrinsic properties that determine if a problem can be solved efficiently in parallel. Therefore, the PRAM models are widely used in literature for implementing parallel algorithms. (See [AIS], [BH], [Col], [KUW], [KR], and [KW] etc.)

1.3 Parallel Algorithm Design and the NC Complexity Class

Given a PRAM model and a particular problem \( Q \), our goal is to design a parallel algorithm solving \( Q \) as efficiently as possible. Without an efficient parallel algorithm, the power of parallel computers can not be fully utilized.

The quality of a parallel algorithm is measured by the resources used by the algorithm. Namely, the number of processors and the number of time units used in solving the given problem.

We use the standard "big \( O \)" notation ([AHU]) in complexity theory to define these two quantities. Let \( n \) be the size of the problem instance. Let \( f(n) \) be a fixed non-negative valued function and let \( g(n) \) be another function on \( n \). If there is a constant \( c > 0 \) and an integer \( n_0 \) such that \( g(n) \leq cf(n) \) for all \( n \geq n_0 \), we write \( g(n)=O(f(n)) \) and call \( g(n) \) an order \( f(n) \) function. The number of processors and the number of time units used by a parallel algorithm is specified by this notation: we often say that a parallel algorithm takes \( O(T(n)) \) time using \( O(P(n)) \) processors where \( T(n) \) and \( P(n) \) are two functions of \( n \). By using the big \( O \) notation, we may concentrate on the asymptotic complexity which, as we will see, reflects the intrinsic properties of the parallel computation.
For each $k \geq 1$, define:

$$NC^k = \{ \text{the problems solvable in } O(\log^k n) \text{ time using polynomially many processors} \};$$

and

$$NC = \bigcup_{k \geq 1} NC^k.$$

$NC$ is generally accepted as the class of problems that can be solved on a parallel computer very fast (polylogarithmic time) and with reasonably many (polynomial) processors. $NC$ is first defined by N. Pippenger [Pi] and extensively studied by many authors (see [Coo2] and [CSV] for a survey). The definition described here is slightly different from the original $NC$ definition given by Pippenger. However, it is widely used by the researchers in the field of parallel algorithm design.

$NC$ is an important complexity class, and there exists an interesting relationship between $NC$ and the well known complexity class $P$. ($P$ is the class of problems solvable by a single processor computer in polynomial time [AHU]). If a problem can be solved on a PRAM with $N$ processors in $T$ time units, the same problem can be solved in $T \ast N$ time units by a single processor computer, since one step of a PRAM with $N$ processors can be simulated by a single processor computer in $N$ steps. Thus, it is easy to see that $NC \subseteq P$. One fundamental question in parallel computing is: $NC = \neq P$. Intuitively we are asking: Is there a fundamental difference between the problems that can be efficiently solved sequentially and the problems that can be efficiently solved in parallel? It is widely believed that $NC \neq P$ (although it seems that we are far away from actually proving this). As a consequence, for a given problem $Q \in P$, it is not necessarily the case that $Q \in NC$. 
Given a problem $Q$, the first important question is: does $Q$ belong to $NC$? One can show $Q \in NC$ by designing a parallel algorithm for $Q$ which takes $O(\log^k n)$ time units using polynomially many processors on a PRAM. Keep in mind that the design of efficient parallel algorithms often requires completely different approaches from the corresponding sequential algorithms and sometimes could be very challenging. For such a problem $Q$, our goal is to show that $Q$ is in $NC$.

If $Q$ is indeed in $NC$, the second important question is: How efficiently can $Q$ be solved in parallel? If a parallel algorithm solves $Q$ in $g(n)$ time using $h(n)$ processors, we can convert this parallel algorithm to a sequential algorithm with running time $g(n)h(n)$ since one processor can always simulate $n$ processors in $n$ steps. Suppose the best sequential algorithm solves $Q$ in $O(f(n))$ time. If $g(n)h(n) = O(f(n))$, we say the parallel algorithm is optimal since this is the best resource bounds we can get without improving the best sequential algorithm. Our second goal is to design a parallel algorithm for $Q$ which is as fast and as "nearly" optimal as possible.

In order to illustrate the difficulties involved in parallel algorithm design, we consider the problem of computing the greatest common divisor (gcd) of two integers $A$ and $B$. On a single processor computer, the gcd of $A$ and $B$ can be computed by using the classical Euclidean algorithm as follows:

$$A = B \times q_1 + r_1;$$

$$B = r_1 \times q_2 + r_2; \cdots$$

If both $A$ and $B$ are $n$ bit numbers, it is easy to show that the gcd of $A$ and $B$ can be found by performing at most $O(n)$ integer divisions. When we
try to solve the gcd problem in parallel, it turns out very hard to reduce the running time substantially. The best known parallel algorithm for the gcd problem takes $O(n^2 \log^2 n)$ processors and runs in $O\left(\frac{n \log \log n}{\log n}\right)$ time [KMR]. We pay a very high price in the number of processors, and achieve only a slight improvement in running time.

The reason for this ineffectiveness is simple: the Euclidean algorithm proceeds in a step by step fashion. The computation of the next step cannot start before the current step finishes. Thus, even with more processors on hand, most of them have to sit idle since they must wait for the results produced by other processors. In our terminology, the gcd problem is not known to be in $NC$. On the other hand, no one has proved that the gcd problem is not in $NC$. It is quite possible that one can design an algorithm, completely different from the Euclidean algorithm, which solves the gcd problem in $O(\log^k n)$ time using polynomially many processors on a PRAM and places the gcd problem into $NC$. Nevertheless, the design of such an algorithm presents a great intellectual challenge for computer scientists and mathematicians.

The computer scientists have identified several problems for which efficient sequential algorithms are well known, but efficient parallel algorithms are very hard to develop. The gcd problem mentioned above is one of them. Two other such problems are from graph theory. One is the maximum matching set (MMS) problem, and another one is the depth first spanning tree (DFST) problem. In spite of many attempts, these problems are still not known to be in $NC$.

The design of parallel algorithms for solving combinatorial problems in graph theory is a particularly active research area. Graph theory arises from
many applied science disciplines [BM]. A sizable portion of problems solved by computers are formulated as graph problems. Therefore, the design of efficient parallel algorithms for graph problems is very important and challenging. Hundreds of papers have been published on designing parallel algorithms for various graph problems (see Bibliography).

In this research, we will develop efficient parallel algorithms for solving various combinatorial problems for different classes of graphs. Our algorithms either improve the best previously known parallel algorithms, or prove, for the first time, that a problem is in $NC$. In particular, we will show that the MMS problems for trees and two terminal series parallel graphs and the DFST problems for planar graphs are all in $NC$.

Our algorithms are efficient in the sense that all of them take only linear number of processors and run in $O(\log n)$ or $O(\log^2 n)$ time (depending on the problem being solved). Thus, compared with the best sequential algorithms, they are all within a polylogarithmic factor of optimal.

1.4 Dissertation Organization

We will first developed a general methodology, called the binary tree algebraic computation (BTAC), for designing efficient parallel algorithms for solving a broad class of graph problems. In Chapter II, the BTAC problem is defined and an efficient parallel algorithm for this problem is developed. Using this methodology as a basic tool, we will develop efficient parallel algorithms for solving various combinatorial problems on Trees (Chapters III and IV), Two Terminal Series Parallel Graphs (Chapters V and VI), and Planar Graphs
(Chapter VII).

The BTAC problem is an algebraic problem. However, it is closely related to many combinatorial problems on graphs. In particular, many important problems on trees can be directly converted into instances of the BTAC problem. Therefore, the BTAC algorithm can be applied to solve them efficiently. In Chapter III, the following problems on trees are solved by using this method: the minimum covering set, maximum independent set, maximum matching set, and minimum r-dominating set problems.

In Chapter IV, a more general version of the r-dominating set problem on trees is considered. By using a sophisticated application of the BTAC methodology, a more efficient parallel algorithm for this problem is presented. A parallel algorithm for another related problem, the p-center problem, on trees is also discussed.

The BTAC methodology can be generalized to design parallel algorithms for solving similar problems on other classes of special graphs, provided that the structure of these special graphs can be represented by a tree. In particular, we will study the class of Two Terminal Series Parallel (TTSP) graphs. In Chapter V, we develop an efficient parallel recognition and decomposition algorithm for TTSP graphs. This algorithm determines if a given graph $G$ is a TTSP graph or not. If $G$ is a TTSP graph, the algorithm also constructs a decomposition tree for $G$ which represents the structure of $G$. Applying a generalized version of the BTAC algorithm on the decomposition tree produced by the recognition and decomposition algorithm, many important combinatorial problems for TTSP graphs can be solved efficiently in parallel. These algorithms are discussed in
Chapter VI.

As mentioned in Section 1.3, the design of an efficient parallel algorithm for constructing a depth first spanning tree (dfst) in a general graph is an important and well-known open problem. In spite of many attempts, this problem is still not known to be in $NC$. Recently, Smith discovered a parallel algorithm for constructing a dfst in a planar graph in $O(\log^3 n)$ time using $O(n^4)$ processors [Sm]. Smith’s work is important since it is the first result showing that the dfst problem for planar graphs is in $NC$. However, due to the importance of the dfst in solving graph problems, it is very desirable to have a more efficient parallel algorithm for this problem. In Chapter VII, we develop such an algorithm which takes only $O(\log^2 n)$ time with $O(n)$ processors, a substantial improvement of Smith’s algorithm.

For most of the graph problems discussed in this dissertation, we concentrate on the derivation of the properties which make fast parallel computation possible, rather than the implementation details. Some commonly used subroutines are presented in the Appendices. Some of the algorithms developed in this research use the existing parallel algorithms as subroutines. In such a case, we only cite the reference, and omit the details.
CHAPTER II

BINARY TREE ALGEBRAIC COMPUTATION

In this chapter, we define the binary tree algebraic computation (BTAC) problem on a finite set $S$ (section 2.1) and develop an efficient parallel algorithm for solving the BTAC problem (section 2.2). The BTAC algorithm runs in $O(k\log n)$ time with $n$ processors on an EREW PRAM (the most restricted PRAM model) where $k$ is the cardinality of $S$ and $n$ is the number of vertices in the input binary computation tree. As a direct application of the BTAC algorithm, we show that any algebraic expression on a finite set $S$ can be evaluated in $O(k\log n)$ time with $n$ processors on an EREW PRAM where $k$ is the cardinality of $S$ and $n$ is the total number of operands and operators in the expression. The BTAC algorithm will be extensively used in this research for designing efficient parallel algorithms for various graph problems.

2.1 Definitions

Most definitions in graph theory used in this dissertation are standard (for example, as defined in [BM] and [Har]).

An (undirected) graph $G = (V,E)$ consists of a set $V$ of vertices and a set $E$ of edges. We usually use letter $n$ to denote the cardinality of $V$, $m$ to denote the cardinality of $E$. We also denote the cardinality of $V$ by $|G|$, called the size
of $G$. An edge $e = (x, y)$ is an (unordered) pair of vertices. A (simple) path in $G$ is a sequence of vertices $P = v_1, v_2, \cdots, v_l$ such that $(v_i, v_{i+1}) \in E$ ($i = 1, \cdots, l-1$), and $\{v_i\}_{1 \leq i \leq l-1}$ are distinct and $\{v_i\}_{2 \leq i \leq l}$ are distinct. $P$ is called a path from $v_1$ to $v_l$. If for any pair of vertices $x, y \in V$ there exists a path from $x$ to $y$, $G$ is connected.

A path $P = v_1, \cdots, v_l$ ($l \geq 3$) is called a (simple) cycle if $v_1 = v_l$. If $G$ contains no cycle, it is called a forest. A connected forest is called a tree. We usually use letter $T$ to denote a tree. It is well known that if a tree $T$ consists of $n$ vertices, then there are $n - 1$ edges in $T$ [BM]. If $x$ and $y$ are two vertices in a tree $T$, then there exists a unique path in $T$ from $x$ to $y$. (The path exists because $T$ is connected. The path is unique because $T$ contains no cycle).

Let $T$ be a tree. If we designate a vertex $r$ as the root, $T$ becomes a rooted tree. Let $x$ be a vertex in a rooted tree $T$ and let $y$ be a vertex on the path from $x$ to the root $r$. $y$ is an ancestor of $x$ and $x$ is a descendent of $y$. If $y$ is the immediate ancestor of $x$ (i.e. $(x, y)$ is an edge), $y$ is the parent of $x$ and $x$ is a child of $y$. We use $P(x)$ to denote the parent of $x$. The parent of $x$'s parent (i.e. $P(P(x))$) is called the grand-parent of $x$. If two vertices $x_1$ and $x_2$ share the same parent, they are brothers. A vertex with no children is called a leaf. A vertex with at least one child is called an internal vertex.

A binary tree is a rooted tree in which each vertex either has two children, or has no children. (In some literature, it is called a full binary tree). If $u$ is an internal vertex of a binary tree, its two children are called the left child and the right child, respectively.
We next define a *binary computation tree* on a given finite set \( S \). Let \( S = \{s_1, \cdots, s_k\} \) be a finite set with \( k \) elements. We denote by \( \Psi_S = \{g: S \times S \to S\} \) the set of binary functions defined on \( S \). A binary computation tree on \( S \) is a binary tree \( T \) such that each leaf \( v \) of \( T \) is labeled by an arbitrary element \( L(v) \in S \), and each internal vertex \( u \) of \( T \) is labeled by an arbitrary binary function \( F(u) \in \Psi_S \).

The *algebraic computation* on a binary computation tree \( T \) is defined as follows: Suppose \( u \) is an internal vertex of \( T \) whose left child \( v_1 \) and right child \( v_2 \) are both leaves. We compute a *final label* \( L(u) = F(u)(L(v_1), L(v_2)) \) for \( u \), and delete \( v_1, v_2 \). (Thus, \( u \) becomes a new leaf). Repeat this operation until all vertices of \( T \) receive a final label \( L(u) \in S \). (Notice that we need to compute the final labels for all vertices, not only the root of \( T \) since the applications of this computation require all final labels being evaluated).

The *binary tree algebraic computation* (BTAC) problem on \( S \) is to perform the algebraic computation on an arbitrary binary computation tree \( T \) on \( S \).

As another description of the BTAC problem, we consider the (fully parenthesized) *algebraic expressions* on a finite set \( S \) defined as follows:

(a) Each \( s \in S \) is an algebraic expression on \( S \).

(b) If \( e_1 \) and \( e_2 \) are two algebraic expressions on \( S \), so is the formula \( (e_1 * e_2) \) where \( * \in \Psi_S \) is an arbitrary binary operator defined on \( S \).

(c) These are the only algebraic expressions on \( S \).

A binary computation tree \( T \) naturally corresponds to a fully parenthesized algebraic expression \( e \) on \( S \). A leaf of \( T \) corresponds to an operand in \( e \). An internal vertex of \( T \) corresponds to an operator in \( e \). The
size of $T$ equals the total number of operands and operators in $e$. The BTAC problem is equivalent to the evaluation of the expression and all sub-expressions in it. For example, if $S = \{0,1\}$, we are dealing with the fully parenthesized Boolean expressions in which any of the sixteen Boolean functions may occur.

### 2.2 A Parallel BTAC Algorithm

In this section, we present a parallel algorithm for solving the BTAC problem. The input to the algorithm is an arbitrary binary computation tree $T = (V,E)$ on $S$ given as follows: For each vertex $v$ of $T$, the parent of $v$ is given by the parent pointer $P(v)$. (The parent pointer of the root $r$ is set to $P(r) = \text{nil}$). Each leaf $v$ is given a label $L(v) \in S$. Each internal vertex $u$ is associated with a binary function $F(u) \in S$. $F(u)$ may be given by any encoding that describes the evaluation rules of the binary function. For simplicity, we assume that $F(u)$ is given by a $k \times k$ table in which the $(i,j)$th entry is the value of $F(u)(s_i, s_j)$. This table is called the multiplication table of $F(u)$. In our applications of the BTAC algorithm, the size of $S$ is usually quite small. The values of $P(v)$, $L(v)$, and $F(v)$ are stored in common memory cells associated with $v$. The output of the algorithm is the final labels $L(v)$ for all $v \in V$ as defined in the last section.

Miller and Reif [MR] developed a parallel tree contraction scheme for solving tree related problems. They applied this scheme to solve the binary tree arithmetic computation problem. This problem is similar to the BTAC problem except that the leaves of the binary computation tree are labeled by numbers and the internal vertices of the binary computation tree are labeled by the
arithmetic operator $+$ or $\ast$. We will use the parallel tree contraction scheme to solve the BTAC problem. In our case, the internal vertices of the binary computation tree are labeled by arbitrary binary functions on $S$ which may be neither commutative nor associative.

The tree contraction scheme makes uses of two operations: RAKE and COMPRESS. These two operations are performed on a tree $T$ in order to reduce the size of $T$.

In our algorithm, each vertex $v$ of $T$ is assigned a processor $P_v$ and is associated with certain common memory cells to store information necessary for the computation. This information includes the values of $P(v)$, $L(v)$, and $F(v)$ as described above. Other necessary information will be introduced later.

The algorithm consists of two phases. The first phase repeatedly applies the RAKE and COMPRESS operations on $T$ until $T$ is "reduced" to its root $r$. During the execution of the first phase, the shape of $T$ keeps changing, and may be no longer a binary tree. (Namely, some vertices of $T$ will have only one child). When an operation is performed on a vertex $v$, the processor assigned to $v$ modifies the information associated with $v$ and $v$'s parent. When the first phase terminates, some vertices of $T$ will have already received their final labels $L(v)$ and other vertices will not have received their final labels as of yet. The second phase of the algorithm will compute these uncomputed final labels by using the information prepared in the first phase.

The first phase is a repeat loop. Each execution of the loop body applies the RAKE and COMPRESS operations on $T$ simultaneously. We discuss these two operations separately.
The RAKE operation simultaneously deletes all leaves of $T$. Depending on different conditions on the leaves to be deleted, there are three cases for the RAKE operation.

**R1:** Suppose $u$ is an internal vertex of $T$ and both its left child $v_1$ and its right child $v_2$ are leaves with final labels $L(v_1)$ and $L(v_2)$, respectively. In this case, the leaves $v_1$ and $v_2$ are deleted by the RAKE operation and the final label $L(u) = F(u)(L(v_1), L(v_2))$ is evaluated immediately. $v_1$ and $v_2$ "report" the values of $L(v_1)$ and $L(v_2)$ to $u$. Then by looking up the multiplication table of $F(u), L(u)$ can be computed in constant time by the processor $P_u$.

**R2:** Suppose $u$ is an internal vertex of $T$ such that one of its children (say the left child $v_1$) is a leaf with a final label $L(v_1)$, and another child (the right child $v_2$ in this case) is not a leaf. In this case, the leaf $v_1$ is deleted by the RAKE operation. After $v_1$ is deleted from $T$, $u$ has only one child $v_2$. Since the final label $L(v_2)$ is not known yet, the final label $L(u)$ can not be evaluated immediately. In order to make the subsequent evaluation possible, we associate $u$ with a unary function $F^1(u):S \rightarrow S$ defined by: $F^1(u)(z) = F(u)(L(v_1), z)$ where the argument $z$ is the unknown final label $L(v_2)$. (If the right child $v_2$ is a leaf, $F^1(u)$ is defined by: $F^1(u)(x) = F(u)(x, L(v_2))$). Since $|S| = k$ is a fixed constant, $F^1(u)$ can be represented as a $k$-element vector $(s_1, \cdots, s_k)$ such that $F^1(u)(s_j) = s_{ij}$ for $j = 1, \cdots, k$. This vector is in fact a proper column (or a row) of the multiplication table of $F(u)$ and can be computed by the processor assigned to $u$ in $O(k)$ time and stored in the common memory cells associated with $u$. 
**R3:** Suppose \( u \) is an internal vertex of \( T \) with only one child \( v \) which is a leaf. (Since the original computation tree \( T \) is a binary tree, this case does not occur when the RAKE operation is applied on \( T \) for the first time. However, after a number of applications of the RAKE operation, this case might occur). In this case, the leaf \( v \) is deleted by the RAKE operation. Before the deletion of \( v \), the final label \( L(v) \) is known and \( u \) is associated with a unary function \( F^1(u) \). So, the final label \( L(u) = F^1(u)(L(v)) \) can be evaluated immediately. This evaluation can be done by the processor \( P_u \) in constant time as follows: In the vector representation of \( F^1(u) \), find the \( L(v) \)th element. This element is the value \( F^1(u)(L(v)) \).

If \( T \) is highly unbalanced, it may take \( O(n) \) applications of the RAKE operation in order to compute all final labels. The COMPRESS (COM for short) operation is introduced to speed-up the computation.

A *chain* of a tree \( T \) is a sequence of vertices \( C = v_1 \cdots v_l \) such that \( v_i \) is the only child of \( v_{i+1} \) for \( 1 \leq i \leq l-1 \) and \( v_1 \) itself has exactly one child. \( C \) is a *maximal chain* if it is a chain and is not a sub-chain of longer chains. The COM operation shortens all maximal chains with length at least 3 as follows. Let \( C = v_1 \cdots v_l \) \((l \geq 3)\) be a maximal chain of \( T \). For each even \( i \), the COM operation deletes \( v_i \) and makes \( v_{i+1} \) the parent of \( v_{i-1} \). We say \( v_{i-1} \) *jumps* to its *grand-parent* \( v_{i+1} \). (If \( l \) is even, the parent of \( v_{l-1} \) remains to be \( v_l \)). Before this operation is performed, each vertex \( v_i \) \((1 \leq i \leq l)\) on \( C \) is associated with a unary function \( F^1(v_i) \). In order to avoid changing the subsequent evaluation, \( v_{i+1} \) will be associated with the composite function \( F^1(v_{i+1}) \circ F^1(v_i) \). Suppose that the function \( F^1(v_{i+1}) \) is represented by a \( k \)-element vector \( \mathbf{v}_{i+1} = (a_1, \ldots, a_k) \) \((a_i \in S)\).
for $i = 1, \cdots, k$) and the function $F^1(v_i)$ is represented by another $k$-element vector $\mathbf{V}_2 = (b_1, \cdots, b_k)$ ($b_j \in S$ for $j = 1, \cdots, k$). The composite function can be represented by a vector $\mathbf{V} = (c_1, \cdots, c_k)$ as follows: The $l$th element $c_l$ in $\mathbf{V}$ is the $t$th element $a_t$ in $\mathbf{V}_1$ where $t = b_l$. Thus, the vector representation of the composite function can be computed by the processor assigned to $v_{i+1}$ in $O(k)$ time.

When the COM operation is performed on a maximal chain $C$, the even indexed vertices $v_i$ on $C$ are deleted without their final labels $L(v_i)$ having been evaluated. However, $L(v_i)$ only depends on the final label $L(v_{i-1})$ and the unary function $F^1(v_i)$. In order to compute the final label $L(v_i)$ at a later time, we associate each vertex $v$ with a $stack(v)$ stored in the common memory. When the vertex $v_i$ is deleted by the COM operation, $v_i$ and the vector representation of the unary function $F^1(v_i)$ is pushed on $stack(v_{i-1})$. Since $F^1(v_i)$ is represented by a $k$-element vector, the push operation can be done by the processor assigned to $v_{i-1}$ in $O(k)$ time. When the final label $L(v_{i-1})$ becomes available later, $stack(v_{i-1})$ will be popped and the final label $L(v_i) = F^1(v_i)(L(v_{i-1}))$ will be evaluated. (This evaluation will be performed in the second phase).

There is one problem remaining for the implementation of the first phase: How does the algorithm know which action (R1, R2, R3, or COM) should be performed on a vertex $v$ at a particular step?

The conditions for performing the RAKE operation (actions R1, R2, and R3) on $v$ involve only the local information of $v$. Namely, the number of children of $v$, the number of children of $v$'s parent, and whether $v$'s brother is a
leaf or not. For each vertex $v$, our algorithm maintains a variable $D(v)$ in a common memory cell associated with $v$. $D(v)$ records the number of $v$'s children. $v$ is a leaf iff $D(v) = 0$. At the beginning of each execution of the first phase, each vertex $v$ "reports" the value $D(v)$ to its parent. Hence, by reading the information associated with $v$ and the parent of $v$, the processor $P_v$ can learn the number of children of $v$, the number of children of $v$'s parent, and whether $v$'s brother is a leaf or not. So, the conditions for performing the RAKE operation on $v$ can be checked in constant time by the processor assigned to $v$.

The condition for performing the COM operation on $v$ is harder to determine. The algorithm can check if $v$ is on a chain $C$ in constant time ($v$ is on a chain iff $D(v) = 1$). However, the determination of whether $v$ is an odd or an even indexed vertex on $C$ involves global information. The direct determination of the index of $v$ on $C$ will take $O(\log n)$ parallel time and increase the running time of the algorithm by a factor of $O(\log n)$.

In order to save this $O(\log n)$ factor, we use the unconditional jump technique of [MR] in the COM operation. We will refer this new COM operation as the unconditional jump COM operation. Let $C = v_1 \cdots v_l$ ($l \geq 3$) be a maximal chain of $T$. When the unconditional jump COM operation is performed on $C$, every vertex $v_i$ on $C$ jumps to its grand-parent $v_{i+2}$, regardless of whether $i$ is even or odd. After the unconditional jump COM operation is performed, $C$ is broken into two sub-chains. (Figure 1 shows such an example. Notice that since the vertex $x$ has two children, it is not in the chain and does not participate the unconditional jump COM operation). One of them (with $v_1$ as the end vertex)
is the chain obtained in the original COM operation. Another chain (with $v_2$ as the end vertex) consists of the vertices $v_i$ ($i$ even) which should have been deleted by the original COM operation. The second chain is called a *pending chain*. In subsequent steps, the unconditional jump COM operation can still be performed on a pending chain. However, since the end vertex of a pending chain has no physical children, the RAKE operation can no longer be performed on a pending chain in subsequent steps. Consequently, the existence of the pending chains has no effect on the correctness of the algorithm.

![Figure 1. A chain is broken into two sub-chains.](image)

When the first phase terminates, some vertices have already received their final labels. Some vertices have not received their final labels yet (they are the vertices deleted by the original COM operation). However, all necessary information for computing these final labels is stored in the $\text{stack}(v)$ where the final
labels $L(v)$ have been computed. These uncomputed final labels are evaluated in the second phase (which is also a repeat loop). Each execution of the loop body of the second phase will simultaneously pop the $\text{stack}(v)$ where $L(v)$ have been computed, and evaluate the final labels for the vertices just popped. The loop is repeated until all $\text{stack}(v)$ are empty. At this point, all final labels will have been evaluated.

We are now ready to present our BTAC algorithm. In this algorithm, the variables $P(v)$, $L(v)$, $F(v)$, and $D(v)$ are already described in the preceding paragraphs. The values of $P(v)$, $L(v)$, and $F(v)$ are given as input. As initialization, the algorithm determines, for each vertex $v$, the number of children of $v$, the left and the right child of $v$ (see Appendix A). For each leaf $v$, $D(v)$ is initialized to 0. For each internal vertex $u$, $D(u)$ is initialized to 2. In addition to these variables, certain common memory cells are allocated to $v$ to store the vector representation of the unary function $F^1(v)$ and the $\text{stack}(v)$. Each vertex $v$ is assigned a processor $P_v$. During the execution of the algorithm, $P_v$ modifies the information associated with $v$ and $v$'s parent. As mentioned before, the conditions for performing the R1, R2, R3, or the unconditional jump COM operation on $v$ can be determined in constant time. For clarity, the checking of these conditions is omitted in the following algorithm. The action "delete $v$" can be implemented as "$P(v)\text{-nil}$" by the processor assigned to $v$. The BTAC algorithm is as follows:

**Algorithm 2.1: BTAC:**

**Input:** A binary computation tree $T=(V,E)$ given by parent pointers $P(v)$.

   Each leaf $v$ of $T$ is given a label $L(v)\in S$. Each internal vertex $u$ of $T$ is
given a binary function $F(u) \in \Psi_\delta$.

Initialization: Compute $D(v)$ for each $v \in V$;

Phase 1: repeat

In parallel for all $u \in V$ do:

(R1) If both the left child $v$ and the right child $w$ of $u$ are leaves, then

compute $L(u) = F(u)(L(v), L(w))$;

$D(u) - 0$;

delete $v, w$;

(R2) If the left child $v$ of $u$ is a leaf, and the right child $w$ of $u$ is not a leaf,

(if $v$ is not a leaf and $w$ is a leaf, perform similar actions), then

compute the vector representation of the unary function $F^l(u)$

defined by: $F^l(u)(x) = F(u)(x, L(v))$;

$D(u) - 1$;

delete $v$;

(R3) If $u$ has only one child $v$ and $v$ is a leaf, then

compute $L(u) = F^l(u)(L(v))$;

$D(u) - 0$;

delete $v$;

(COM) If $u$ has only one child $v$, and $v$, $u$, $P(u)$ are all on a chain, then

$F^l(P(u)) = F^l(F^l(u)) \circ F^l(u)$;

push $(u, F^l(u))$ on stack($v$);
\[ P(v) - P(u); \]

(null) If none of above, then do nothing on \( u \);

Until only the root \( r \) of \( T \) is left (end of Phase 1);

Phase 2: repeat

In parallel for all \( v \in V \) do:

If \( L(v) \) has been computed and \( stack(v) \) is not empty, then

\[ (u, F^1(u)) - pop stack(v); \]

compute \( L(u) = F^1(u)(L(v)) \); \n
Until every \( L(v) \) has been computed (end of Phase 2);

End BTAC.

We next analyze the resource bounds of the BTAC algorithm. Since one processor is assigned to one vertex of \( T \), the algorithm clearly takes \( n \) processors. In order to analyze the time bound of the BTAC algorithm we need the following lemmas. (Similar result was proved in [MR]).

**Lemma 2.1:** Let \( T = (V, E) \) be a rooted tree such that every vertex \( v \in V \) has either 0, 1, or 2 children in \( T \), and every maximal chain of \( T \) has at most 2 vertices. Then the number of leaves in \( T \) is at least \(|T|/6\).

**Proof:** Define:

\[ L = \{ v \in V \mid v \text{ has no children in } T \}; \] (leaves).

\[ C = \{ v \in V \mid v \text{ has one child in } T \}; \] (vertices on chains).

\[ I = \{ v \in V \mid v \text{ has two children in } T \}; \] (internal vertices).

Each vertex in \( C \) is on a maximal chain of \( T \). We replace each maximal chain by a new single edge (thus, deleting all vertices in \( C \)). The resulting
graph is a binary tree $T'$ (i.e. each vertex of $T'$ has either 0 or 2 children). So, the vertex set of $T'$ is $V' = L \cup I$. The leaves of $T'$ is the set $L$ and the internal vertices of $T'$ is the set $I$. Let $E'$ denote the edge set of $T'$.

Since $T'$ is a binary tree, we have [AHU]:

$$ |L| = |I| + 1; \quad \text{and} \quad |E'| = |V'|-1 = |L| + |I| - 1. $$

Since each maximal chain of $T$ corresponds to an edge of $T'$ and contains at most 2 vertices, we have: $|C| \leq 2|E'| = 2(|L| + |I| - 1)$. Thus:

$$ |L| = \frac{|L| + |I| + 1}{2} = 2\left(\frac{|L| + |I| - 1}{2} + \frac{4}{4}\right) \geq \frac{|C| + 4}{4}. $$

So, $6|L| = 2|L| + 4|L| \geq 2|L| + |C| + 4 = |L| + |I| + 1 + |C| + 4$.

Hence: $|L| \geq \frac{|L| + |I| + |C| + 5}{6} = \frac{|T| + 5}{6} > \frac{|T|}{6}$ as to be shown. □

**Lemma 2.2:** Let $T=(V,E)$ be a rooted tree such that every vertex $v \in V$ has either 0, 1 or 2 children. Then the simultaneous application of the RAKE and COMPRESS operations on $T$ deletes at least $|T|/6$ vertices.

**Proof:** Let $C=\{v \in V \mid v$ is on a maximal chain of $T$ with length $\geq 3\}$. Suppose $C_1, \ldots, C_k$ (for some $k$) are all maximal chains of $T$ with length $\geq 3$. Thus, $C = \bigcup_{1 \leq i \leq k} C_i$.

If $|C_i|$ is an odd number, then the COMPRESS operation will delete $\frac{|C_i| - 1}{2} \geq \frac{|C_i|}{6}$ vertices in $C_i$ from $T$.

If $|C_i|$ is an even number, then the COMPRESS operation will delete $\frac{|C_i| - 2}{2} \geq \frac{|C_i|}{6}$ vertices in $C_i$ from $T$. 
Hence, the COMPRESS operation deletes at least \( \sum_{i=1}^{k} \frac{|C_i|}{6} = \frac{|C|}{6} \) vertices in \( C \) from \( T \).

In \( T \), replace each \( C_i \) by a new single edge. The resulting tree \( T' \) satisfies the conditions in Lemma 2.1. The vertex set of \( T' \) is \( V' = V - C \). The set \( L \) of leaves of \( T' \) is also the set of leaves of \( T \). \( L \) will be deleted from \( T \) by the RAKE operation. By Lemma 2.1, \( |L| \geq \frac{|V'|}{6} \).

Therefore, the total number of vertices deleted by the RAKE and COMPRESS operations is at least \( \frac{|V'|}{6} + \frac{|C|}{6} = \frac{|T|}{6} \) as to be shown. \( \square \)

It follows from Lemma 2.2 that the first repeat loop (Phase 1) of the BTAC algorithm is executed at most \( \log_{\frac{8}{5}} n \) iterations. Hence, the maximal depth of \( \text{stack}(v) \) \( (v \in V) \) is at most \( \log_{\frac{8}{5}} n \). The second repeat loop (Phase 2) of the BTAC algorithm keeps popping \( \text{stack}(v) \) for each \( v \in V \). Suppose that \( M \leq \log_{\frac{8}{5}} n \) is the maximal depth of all \( \text{stack}(v) \). From the BTAC algorithm, it is easy to see that the \( \text{stack}(v) \) with depth \( M \) are popped starting from the first execution of the second repeat loop, the \( \text{stack}(v) \) with depth \( M - 1 \) are popped starting from the second execution \( \cdots \) and so on. Hence, the second repeat loop is executed at most \( \log_{\frac{8}{5}} n \) iterations. We have already shown that the body of the first repeat loop can be executed in \( O(k) \) time. By using similar techniques, it is easy to show that the body of the second repeat loop can also be implemented in \( O(k) \) time. Therefore, the BTAC algorithm takes \( O(k \log n) \) time.

During the computation, two processors associated with the children of a vertex \( u \) may need to modify (or access) the information associated with \( u \) at
the same time. Since \( u \) has only two children, we can let the processor associated with the left child modify (access) the information first and let the processor associated with the right child modify (access) the information second. By using this arrangement, concurrent writes (reads) from the common memory can be avoided (the running time is increased by a factor of 2). Thus, the BTAC algorithm can be executed on an EREW PRAM. As a summary, we have:

**Theorem 2.3:** For any finite set \( S \) of \( k \) elements, the BTAC problem on \( S \) can be solved in \( O(k \log n) \) time with \( n \) processors on an EREW PRAM where \( n \) is the number of vertices in the input binary computation tree. \( \square \)

Applying Theorem 2.3 to the evaluation of the algebraic expressions on the set \( S \), we have:

**Corollary 2.4:** Let \( S \) be a finite set of \( k \) elements and let \( e \) be any algebraic expression on \( S \) given by binary tree form. Then \( e \) can be evaluated in \( O(k \log n) \) time with \( n \) processors on an EREW PRAM where \( n \) is the total number of operators and operands in \( e \). \( \square \)
CHAPTER III
PARALLEL ALGORITHMS FOR SOLVING
TREE PROBLEMS

The BTAC problem is an algebraic computation problem. However, it is closely related to many combinatorial problems on graphs. In this chapter, we will apply the BTAC algorithm to solve some important combinatorial problems on trees.

The trees form a basic class of graphs. Many problems for other classes of graphs can be reduced to the tree problems. The problems for trees also find practical applications. For example, the underline graph of some transportation network may be a tree. In this case, the tree problems correspond to various strategic location problems on the network. For these reasons, the sequential algorithms for solving tree problems have been extensively studied (see [BLW], [BM], [CT1], [CGH], [FJ], [Hak1], [Han], [KH] etc.) We will develop parallel algorithms for these problems in this chapter.

Our parallel algorithms will work for any unrooted tree given by edge list representation. However, the algorithms are easier to describe if the input is a rooted tree given by adjacency list representation. In Section 3.1 we discuss how to convert the edge list of an unrooted tree to the adjacency list of the corresponding rooted tree. In Section 3.2, a parallel algorithm for solving the
minimum covering set (MCS) problem on trees is presented. In Section 3.3, parallel algorithms for two related problems, the maximum independent set (MIS) and the maximum matching set (MMS) problems on trees, are discussed. In Section 3.4 we present a parallel algorithm for solving the minimum r-dominating set (r-MDS) problem on trees.

The importance and the applications of the problems listed above are well-known [BM]. For general graphs, these problems are NP-complete (except for the MMS problem). Restricted to trees, linear time sequential algorithms exist. However, parallel algorithms for them have not appeared in literature. Our parallel algorithms for the MDS, MIS, and MMS problems run in \( O(\log n) \) time with \( O(n) \) processors on an EREW PRAM. The algorithm for the r-MDS problem runs in \( O(r\log n) \) time with \( O(n) \) processors on an EREW PRAM. Hence, they are all within an \( O(\log n) \) factor of optimal.

### 3.1 Input Representation

All parallel algorithms presented in this chapter will work for arbitrary unrooted trees (not only binary trees). However, the algorithms are easier to describe if the input is a rooted tree. In this section, we discuss how to convert an unrooted tree to a rooted tree and how to prepare the input so that it is suitable for our parallel algorithms.

Let \( T = (V, E) \) be an unrooted tree. The edge list representation of \( T \) is the list of the edges of \( T \) in arbitrary order.

Let \( T = (V, E) \) be a rooted tree. \( T \) can be described by specifying the parent \( P(v) \) for each vertex \( v \in V \). The parent pointer representation of \( T \) is the
list of the edges \{ (P(v),v) \mid P(v) \text{ is the parent of } v \} given in arbitrary order.

The adjacency list representation of a rooted tree \( T \) is the list of the edges \{ (P(v),v) \mid P(v) \text{ is the parent of } v \} arranged in the order such that edges whose first vertex are the same appear consecutively in the list. Another equivalent description for this representation is that for each internal vertex \( u \) of \( T \), a list consisting of all children of \( u \) is given.

Let \( T=(V,E) \) be an unrooted tree given by the edge list representation and let \( r \) be an arbitrary vertex of \( T \). By using the Euler Tour technique of Tarjan and Vishkin [TV1], \( T \) can be converted to a rooted tree with \( r \) as the root. The parent pointer representation of the rooted version of \( T \) can also be constructed. The resource bound for this conversion is \( O(\log n) \) time with \( O(n) \) processors on an EREW PRAM [TV1].

Let \( T=(V,E) \) be a rooted tree given by the parent pointer representation. The adjacency list representation of \( T \) can be constructed as follows: Perform a parallel sort on the edge list \{ (P(v),v) \mid P(v) \text{ is the parent of } v \} according to the first vertex of each edge. The result is the adjacency list of \( T \). Sorting can be done in \( O(\log n) \) time with \( O(n) \) processors on an EREW PRAM by using the algorithm in [Col].

In the following sections, we assume that the conversions described above have been performed on \( T \) and the input to our parallel algorithms are rooted trees given by the adjacency list representation. Since these conversions can be performed in \( O(\log n) \) time with \( O(n) \) processors on an EREW PRAM, this preprocessing does not affect the resource bounds of our parallel algorithms.
3.2 Minimum Covering Set Problem

A covering set of a tree $T = (V, E)$ is a subset $C \subseteq V$ such that for any edge $(i, j) \in E$, $\{i, j\} \cap C \neq \emptyset$. Intuitively, every edge of $T$ is covered by a vertex in $C$. The minimum covering set (MCS) problem is to find a covering set of $T$ with minimum cardinality.

We actually solve the following slightly more general problem. Given a tree $T = (V, E)$, a labeling on $T$ is a mapping $L : V \rightarrow \{F, R\}$. A mixed covering (mc for short) set of $T$ with respect to the given labeling $L$ is a subset $C \subseteq V$ such that $C$ is a covering set of $T$ and $\{v \mid L(v) = R\} \subseteq C$. Intuitively, a mixed covering set of $T$ is a covering set of $T$ which includes all vertices with a label $R$. By this definition, a covering set of $T$ is just a mixed covering set of $T$ with respect to the labeling $L$ defined by $L(v) = F$ for all $v \in V$. We denote by $mc(T, L)$ the cardinality of a minimum mc set of $T$ with respect to $L$.

The parallel algorithm for finding a minimum mc set of $T$ with respect to a given labeling $L$ is developed as follows. We first design a sequential algorithm for the problem. Then we show that this sequential algorithm can be converted into an instance of the BTAC problem defined in Chapter II. Thus, the BTAC algorithm can be used to solve this new problem.

**Lemma 3.1:** Let $T = (V, E)$ be a tree with a labeling $L : V \rightarrow \{F, R\}$. Let $v$ be a leaf of $T$, and let $u$ be the parent of $v$. Define $T' = T - \{v\}$.

1. If $L(v) = F$, let $L'$ be the labeling of $T'$ defined by: $L'(w) = L(w)$ for $w \in V - \{u, v\}$ and $L'(u) = R$. Then $mc(T, L) = mc(T', L')$.
2. If $L(v) = R$, let $L'$ be the restriction of $L$ to $T'$. Then $mc(T, L) = mc(T', L') + 1$.
Proof: We only prove (2). The proof of (1) is similar.

Let \( C' \) be a minimum mc set of \( T' \) with respect to \( L' \). Define \( C = C' \cup \{v\} \).

We will show that \( C \) is a mc set of \( T \) with respect to \( L \). For any \( w \in V - \{v\} \) such that \( L(w) = R \), we have \( L'(w) = R \) by the definition of \( L' \). This implies \( w \in C' \) and hence \( w \in C \) (since \( C' \subseteq C \)). Since \( v \) is also in \( C \), we have \( \{w \mid L(w) = R\} \subseteq C \). Let \((i,j)\) be an edge of \( T \). If \((i,j) \neq (u,v)\), \((i,j)\) is also an edge of \( T' \). Hence, \( \{i,j\} \cap C \supseteq \{i,j\} \cap C' \neq \emptyset \). If \((i,j) = (u,v)\), we trivially have \( \{i,j\} \cap C \neq \emptyset \) since \( v \in C \). So, \( C \) is a mc set of \( T \) with respect to the labeling \( L \).

Thus, \( mc(T,L) \leq |C| = |C'| + 1 = mc(T',L') + 1. \)

Let \( C \) be a minimum mc set of \( T \) with respect to \( L \). Define \( C' = C - \{v\} \).

We now show that \( C' \) is a mc set of \( T' \) respect to \( L' \). Let \( w \neq v \) be a vertex in \( T' \). If \( L'(w) = R \), then \( L(w) = R \) and \( w \in C \). Hence, \( w \in C' \). Let \((i,j)\) be any edge of \( T' \). Since \((i,j)\) is also an edge of \( T \) and \( C \) is a mc set of \( T \), we have \( \{i,j\} \cap C \neq \emptyset \). Since \((i,j)\) can not be the edge \((u,v)\), we have \( \{i,j\} \cap C' \neq \emptyset \). So, \( C' \) is a mc set of \( T' \) with respect to \( L' \). Thus, \( mc(T',L') \leq |C'| = |C| - 1 = mc(T,L) - 1. \)

\[ \begin{array}{c|c|c|c} f_{MCS} & F & R \\ \hline F & R & F \\ \hline R & R & R \end{array} \]

Figure 2. The binary function \( f_{MCS} \).
Define $S = \{F,R\}$ and define a binary function $f_{MCS}: S \times S \rightarrow S$ as in Figure 2. The row index of the table is the first argument of $f_{MCS}$ and the column index of the table is the second argument of $f_{MCS}$. The interpretation of $f_{MCS}$ is as follows: Let $T$ be a tree with a given labeling $L$. Suppose $v$ is a leaf of $T$ and $u$ is its parent. If $L(u) = y$ and $L(v) = x$, then $f_{MCS}(y, x)$ is the label received by $u$ after deleting $v$ from $T$ as in Lemma 3.1.

The following sequential algorithm computes a minimum mc set of a tree $T$ with respect to a given labeling $L$. The correctness of the algorithm easily follows from Lemma 3.1.

Algorithm 3.1. Sequential MCS:

Input: A rooted tree $T = (V,E)$ given by adjacency list. Each vertex $v$ of $T$ is given a label $L(v) \in S$.

$C \leftarrow \emptyset$;

Repeat:

   Arbitrarily select a leaf $v$ and let $u$ be its parent; (the algorithm works for any selection of $v$. In order to make it deterministic, we may let $v$ be the leaf with the largest vertex number);

   If $L(v) = R$ then $C \leftarrow C \cup \{v\}$;

   $L(u) \leftarrow f_{MCS}(L(u), L(v))$;

   Delete $v$ from $T$;

Until only the root $r$ of $T$ remains;

End Sequential MCS.
In order to compute an ordinary minimum covering set of $T$, we give every vertex $v$ of $T$ a label $L(v) = F$ and run the Sequential MCS algorithm on $T$.

Figure 3 is an example of the sequential MCS algorithm. Figure 3 (1) shows a tree $T$ with 11 vertices. We label all vertices of $T$ by $L(v) = F$. When Algorithm 3.1 is performed on $T$, the vertices 11, 10, \ldots, 2, 1 are deleted in that order. Figure 3 (2) shows the final labels of the vertices after Algorithm 3.1 terminates. A minimum covering set of $T$ is $C = \{1, 3, 4, 9\}$.

Notice that the minimum mc $C$ computed in Algorithm 3.1 is completely determined from the final labels: $C = \{v \mid$ the final label of $v$ is $R\}$. So, the MCS problem is equivalent to the computation of these final labels. We will convert this computation into an instance of the BTAC problem on $S = \{F, R\}$.

Let $T_1 = (V, E)$ be an arbitrary tree. In order to compute the final labels for $T_1$ as in Algorithm 3.1, we convert $T_1$ into a binary computation tree $T$ on $S$ as follows. Suppose $u$ is a vertex of $T_1$ with children $v_1, \ldots, v_k$. We introduce $k$ dummy vertices $u_0, u_1, \ldots, u_k$. Make $u_i$ the left child of $u_{i+1}$ ($0 \leq i \leq k - 1$ where $u_k = u$). Make $v_i$ the right child of $u_i$ ($1 \leq i \leq k$ where $u_k = u$). After performing this operation on all internal vertices of $T_1$, we get a binary tree $T$ (Figure 4 is an example of this conversion). If $u_0$ is a leaf of $T$ and is a dummy vertex introduced for the vertex $u$ in $T_1$, we give $u_0$ the label $L(u)$ from $T_1$. If $v$ is a leaf of $T$ and also a leaf of $T_1$, we give $v$ the label $L(v)$ from $T_1$. The internal vertices of $T$ are labeled by the binary function $f_{MCS}$. Thus, $T$ becomes a binary computation tree on $S$. It is easy to see that after performing the algebraic computation on $T$ the final labels of the original vertices of $T_1$ are exactly the final labels computed by Algorithm 3.1.
Figure 3. An example of the MCS algorithm.

Figure 4. A tree $T_v$ is converted to a binary tree $T$. 
The conversion from $T_1$ to $T$ can be done as follows: First we compute the number of children of each vertex $v \in V$. This can be done in $O(\log n)$ time using $O(n)$ processors on an EREW PRAM. (See Appendix A). Then the dummy vertices are introduced and the parent pointers are set properly. This completes the construction of $T$. Since the total number of dummy vertices introduced into $T$ equals the number of edges in $T_1$, we have $|T| = 2|T_1| - 1$. Thus, this conversion affects the time and processor bounds of the algorithm only by a factor 2.

We now have the following parallel MCS algorithm: First convert the input tree $T_1$ to a binary tree $T$. Label vertices of $T$ as described above. Then perform the BTAC algorithm on $T$. Upon termination, a MCS of $T_1$ will be the set $C = \{v \mid v$ is a vertex in $T_1$ and the final label of $v$ is $R\}$. Since both the conversion from $T_1$ to $T$ and the application of the BTAC algorithm on $T$ can be done in $O(\log n)$ time with $O(n)$ processors, we have:

**Theorem 3.2:** The MCS problem on trees can be solved in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM where $n$ is the number of vertices in the input tree. □

### 3.3 Maximum Independent Set and Maximum Matching Set Problems

In this section, we consider the maximum independent set (MIS) and the maximum matching set (MMS) problems on trees. Both of them are closely related to the MCS problem.

An independent set of a tree $T = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent. The MIS problem is to find an independent set of $T$
with maximum cardinality. It is well known that a subset $C \subseteq V$ is a minimum covering set of $T$ iff the subset $V - C$ is a maximum independent set of $T$ [BM].

So, the MCS algorithm developed in Section 3.2 can be used to solve the MIS problem. For example, a maximum independent set of the tree $T$ in Figure 3 is the set $\{2, 5, 6, 7, 8, 10, 11\}$. Thus, we have:

**Theorem 3.3:** The MIS problem on trees can be solved in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM. $\square$

A matching set of a tree $T = (V, E)$ is a subset $M \subseteq E$ such that no two edges in $M$ share a common vertex. The MMS problem is to find a matching set of $T$ with maximum number of edges.

**Lemma 3.4:** Let $T = (V, E)$ be a tree with all vertices labeled by $F$. Let $L(v) (v \in V)$ be the final labels computed by Algorithm 3.1. Then:

1. Suppose $u$ is an internal vertex of $T$ and $v_1, \ldots, v_k$ are its children. If $L(u) = R$, at least one $v_i \ (1 \leq i \leq k)$ has a final label $L(v_i) = F$;

2. For each $u$ with $L(u) = R$, let $v_u$ be a child of $u$ with a final label $F$. The collection of edges $M = \{ (u, v_u) \mid L(u) = R \}$ is a MMS of $T$.

**Proof:** (1) From the multiplication table of $f_{MCS}$ (Figure 2), a vertex $u$ may get a final label $R$ only if at least one of its children gets a final label $F$.

(2) From the definition of $M$, $M$ is clearly a matching set of $T$. It is well-known that in a bipartite graph, in particular in a tree, the cardinality of a MCS equals the cardinality of a MMS [BM]. Since the cardinality of $M$ equals the cardinality of the minimum covering set computed by Algorithm 3.1, $M$ is a maximum matching set of $T$. $\square$
In Figure 3, a maximum matching set of the tree $T$ is the set $M = \{(9,11),(3,7),(4,6),(1,2)\}$.

Lemma 3.4 suggests the following parallel MMS algorithm: Label all vertices of $T$ by $F$. Apply the parallel MCS algorithm to compute the final labels $L(v)$ for all $v \in V$. For each vertex $u$ with a final label $R$, select a child with a final label $F$. Then a maximum matching set of $T$ can be constructed as in Lemma 3.4. The selection can be done as follows. For each internal vertex $u$, we sort the children of $u$ according to their final labels. (Namely, the children with a final label $F$ appear in the sorted list first, and the children with a final label $R$ appear in the sorted list second). Then we select the first child in the sorted list. Sorting can be done in $O(\log n)$ time using $O(n)$ processors on an EREW PRAM [Col]. In summary we have:

**Theorem 3.5:** The MMS problems on trees can be solved in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM. □

### 3.4 Minimum $r$-Dominating Set Problem

Let $T = (V,E)$ be a tree and let $r$ be a fixed integer. For any $u,v \in V$, the *distance* between $u$ and $v$ is the number of edges on the unique path from $u$ to $v$ in $T$. We denote by $d_T(u,v)$ (or simply $d(u,v)$ if $T$ is understood) the distance between $u$ and $v$ in $T$. If $d_T(u,v) \leq r$, we say $v$ is *$r$-dominated by $u$*. An *$r$-dominating set* of $T$ is a subset $D \subseteq V$ such that every vertex $v$ of $T$ is $r$-dominated by some vertex $u \in D$. The *minimum $r$-dominating set* ($r$-MDS) problem is to find an $r$-dominating set of minimum cardinality.
We will follow the same approach as in Section 3.2 to design a parallel $r$-MDS algorithm: First we develop a sequential algorithm for the problem. Then we show that this sequential algorithm can be converted to an instance of the BTAC problem and be solved by the BTAC algorithm. Slater has a similar linear time sequential algorithm for this problem [Sl]. However, our algorithm is more suitable for parallel implementation.

For any vertex $v \in V$, the subtree of $T$ rooted at $v$ is denoted as $T(v)$. $[-r,r]$ denotes the set of integers between $-r$ and $r$.

The sequential algorithm performs a "bottom up" search on $T$, starting from the leaves and moving toward the root $t$ of $T$. During the search, vertices of $T$ are added to the desired minimum $r$-dominating set $D$ in an optimal way until all vertices of $T$ are covered by some vertex in $D$ within distance $r$. To do this, we associate a label $L(v)$ with each vertex $v$. $L(v)$ are initialized to $-r$ for all $v \in V$. During the computation, $L(v)$ are modified but always have a value in $[-r,r]$. When the computation terminates, the minimum $r$-dominating set of $T$ is determined from these final labels.

In order to understand the intuitive meaning of the label $L(v)$, we consider a complete binary tree $T$ with height $2r$ (i.e. every leaf of $T$ has distance $2r$ to the root $t$ of $T$). We say the root $t$ is the level 0 vertex, the children of the root are level 1 vertices, $\cdots$, the leaves are level $2r$ vertices. When the sequential $r$-MDS algorithm is performed on $T$, the level $2r$ vertices (leaves) will have a final label $-r$, the level $2r-1$ vertices will have a final label $-r+1$, $\cdots$, the level $r$ vertices will have a final label $0$, $\cdots$, the level 0 vertex (root $t$) will have a final label $r$. The minimum $r$-dominating set of $T$ consists of all vertices
with a final label 0 (i.e. the level \( r \) vertices). The intuitive meaning of the final labels \( L(v) \) is as follows. If \( L(v) = -a \) for some \( a > 0 \), then \( v \) is dominated from "above" by a vertex in the \( r \)-dominating set within distance \( a \). If \( L(v) = b \) for some \( b > 0 \), then \( v \) is dominated from "below" by a vertex in the \( r \)-dominating set within distance \( b \). If \( L(v) = 0 \), then \( v \) itself is in the \( r \)-dominating set.

Since our algorithm has to deal with arbitrary trees, the meaning of the label \( L(v) \) and the way they are modified are more complicated than that described above.

\( L(v) \) are modified as follows. Suppose \( v \) is a leaf with a label \( L(v) = x \) and \( u \) is \( v \)'s parent with a label \( L(u) = y \). We delete \( v \) and call \( x \) the final label of \( v \). If \( x = 0 \), \( v \) is added to \( D \). \( u \) is given a new label \( L(u) \) determined from \( x \) and \( y \) according to certain label modification rules. More precisely, \( u \) is relabeled by a new label \( L(u) = f_{r-MDS}(y, x) \) where \( f_{r-MDS} : [-r, r] \times [-r, r] \to [-r, r] \) is a binary function describing the label modification rules (\( f_{r-MDS} \) will be defined later). This operation is repeated until \( T \) is reduced to the root \( t \). If \( t \) gets a final label \( L(t) \leq 0 \), \( t \) is also added to \( D \). The sequential \( r \)-MDS algorithm is as follows:

Algorithm 3.2. Sequential \( r \)-MDS:

Input: A rooted tree \( T = (V, E) \) given by adjacency list.

Label all vertices \( v \in V \) by \( L(v) = -r \);

\( D = \emptyset \);

Repeat:

Arbitrarily select a leaf \( v \) and let \( u \) be its parent; (the algorithm works for any selection of \( v \). In order to make it deterministic, we may let \( v \) be the leaf with the largest vertex number);
If \( L(v) = 0 \) then \( D \leftarrow D \cup \{v\} \);

\[ L(u) \leftarrow f_{r-MDS}(L(u), L(v)) ; \]

Delete \( v \) from \( T \);

Until only the root \( t \) of \( T \) remains;

If \( L(t) \leq 0 \) then \( D \leftarrow D \cup \{t\} \);

End Sequential r-MDS.

We next define the function \( f_{r-MDS} \) and explain how the algorithm works. Since the leaves of \( T \) are deleted repeatedly in Algorithm 3.2, the shape of \( T \) keeps changing. At any stage of the computation, the tree remaining at the stage is called the current tree and the vertices in the current tree are called current vertices. Let \( v \) be a current vertex. Let \( v_1, \ldots, v_k \) be the children of \( v \) in the original \( T \), but not in the current tree. Namely, \( v_i (1 \leq i \leq k) \) have been deleted during the execution of the algorithm. The subtrees \( T(v_i) (1 \leq i \leq k) \) have been deleted by the algorithm. Let \( S(v) \) denote the union of \( v \) and the vertices in these deleted subtrees. (If no subtree rooted at a child of \( v \) has been deleted, \( S(v) \) consists of \( v \) itself). In order to make the algorithm work, we require that at any stage of the computation and for any vertex \( v \) in the current tree, \( L(v) \) satisfies the following two conditions:

(a) If all vertices in \( S(v) \) (including \( v \) itself) have already been "covered" by a vertex in \( S(v) \cap D \) within distance \( r \), \( v \) will have a positive label \( L(v) \) such that \( L(v) \) is the distance from \( v \) to the nearest vertex \( w \) which is in \( S(v) \) and is already included in \( D \). More precisely, \( L(v) = \min_w \{ d_T(w, v) \mid w \in S(v) \cap D \} \).
(b) If there are some vertices \( w \in S(v) \) that have not been covered by any vertex in \( S(v) \cap D \) within distance \( r \), \( v \) will have a non-positive label \( L(v) \leq 0 \) such that \(-L(v)\geq 0\) is the maximum distance from \( v \) within which a vertex must be included into \( D \) in order to cover the uncovered vertices in \( S(v) \). More precisely, \(-L(v)=r-\max\{d_{T}(w,v)\mid w \in S(v)\text{ and } w \text{ is not covered by any vertex in } S(v) \cap D \text{ within distance } r\}\). In particular, if \( L(v)=0 \), then \( v \) itself must be included into \( D \) in order to cover the uncovered vertices in \( S(v) \).

If \( L(v) \) satisfies the above two conditions for all vertices in the current tree, \( L \) is called a **proper labeling** for the current tree.

Initially, we assign \( L(v)=-r \) for all \( v \in V \). Since no vertices have been deleted, \( S(v)=\{v\} \) for all \( v \in V \), and condition (b) is clearly satisfied for all \( v \). So, this initial labeling is a proper labeling for the original tree \( T \).

The label modification rules are described by a binary function \( f_{r-MDS}:[-r,r] \times [-r,r] \rightarrow [-r,r] \). Suppose \( v \) is a leaf in the current tree with a final label \( L(v) \) and \( u \) is \( v \)'s parent with a current label \( L(u) \). The function value \( f_{r-MDS}(L(u),L(v)) \) will be the new label of \( u \). \( f_{r-MDS} \) is defined by the following formulas:

1. For each fixed \( i \geq 0 \):
   
   (1.1) \( f_{r-MDS}(i,x)=x+1 \), if \(-i \leq x \leq i-1\);
   
   (1.2) \( f_{r-MDS}(i,x)=i \), if \(-r \leq x \leq -(i+1) \) or \( i \leq x \leq r\);

2. For each fixed \(-i < 0\):
   
   (2.1) \( f_{r-MDS}(-i,x)=x+1 \), if \(-i \leq x \leq i-1\);
\[(2.2) f_{r-MDS}(-i,x) = -i, \text{ if } -r \leq x \leq -(i+1) \text{ or } i \leq x \leq r.\]

We will show that the label modification rules defined by $f_{r-MDS}$ preserve the properness of the labeling. We only consider the formulas (1.1) and (1.2). The formulas (2.1) and (2.2) can be interpreted similarly.

Suppose $v$ is a leaf with a final label $L(v)$ and $u$ is its parent with a current label $L(u)$. Assume that the labeling is proper for the current tree $T$. We must show that after delete $v$ and relabel $u$ by $f_{r-MDS}(L(u),L(v))$, the new labeling is proper for the resulting tree.

First consider the case $i=0$. Since the current label $L(u)=i=0$, $u$ must be included into $D$ in order to cover the uncovered vertices in $S(t)$. So, no matter what the final label $L(v)=x$ is, the new label $L(u)$ must remain 0. Thus, formula (1.2) returns correct new label $L(u)=0$ (Figure 5 (a)).

Next consider the case $i>0$. For (1.1) we distinguish three cases.

Case 1: $L(v)=x=0$. Since $L(v)=0$, $v$ is in $D$. Thus, $u$ is covered by $v$ within distance $d(v,u)=1$. So, the new label should be $L(u)=1$ and (1.1) returns correct value (Figure 5 (b)).

Case 2: $0<L(v)=x\leq i-1$. Since $L(u)=i>0$, there is a $w \in S(u) \cap D$ such that $d(w,u)=i$. Since $L(v)=x>0$, there is a $w' \in S(v) \cap D$ such that $d(w',v)=x$. Since $d(u,w')=d(u,v)+d(v,w')=1+x\leq i=d(u,w)$, $w'$ is closer to $u$ than $w$ is. Therefore, the new label should be $L(u)=d(u,w')=x+1$ and (1.1) returns correct value (Figure 5 (c)).
Figure 5. Label modification rules.
\begin{figure}
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
$f_{2-MDS}$ & -2 & -1 & 0 & 1 & 2 \\
\hline
-2 & -1 & 0 & 1 & 2 & -2 \\
\hline
-1 & -1 & 0 & 1 & -1 & -1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 \\
\hline
2 & -1 & 0 & 1 & 2 & 2 \\
\hline
\end{tabular}
\caption{The binary function $f_{2-MDS}$.}
\end{figure}

\begin{figure}
\centering
\begin{tikzpicture}
\node (1) at (0,0) {1}
child {node (3) {3}
child {node (8) {8}
child {node (9) {9}}
child {node (11) {11}}}
child {node (7) {7}}}
child {node (2) {2}};
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
\node (1) at (0,0) {1}
child {node (0) {0}
child {node (-2) {-2}}
child {node (-1) {-1}}}
child {node (-1) {-1}};
\end{tikzpicture}
\caption{An example of the r-MDS algorithm with $r=2$.}
\end{figure}
Case 3: \(-i \leq x = L(v) < 0\). Since \(L(u) = i > 0\), there is a \(w \in S(u) \triangleq D\) such that \(d(u, w) = i\). Since \(L(v) = x < 0\), \(v\) must be covered by a vertex in \(D\) within distance \(-L(v)\). Since \(-L(v) = -x \leq i < i + 1 = d(u, w) + d(v, u) = d(v, w)\), \(w\) can not cover \(v\) within distance \(-L(v)\). In order to cover \(v\) within distance \(-L(v)\), \(u\) must be covered by a vertex within distance \(-L(v) - 1\). Therefore, the new label \(L(u)\) must satisfy \(-L(u) = -L(v) - 1\). Thus, (1.1) returns correct value \(L(u) = L(v) + 1 = x + 1\) (Figure 5 (d)).

For (1.2) we consider two cases.

Case 4: \(i \leq x = L(v) \leq r\). Since \(L(u) = i > 0\), there is a \(w \in S(u) \triangleq D\) such that \(d(w, u) = i\). Since \(L(v) = x > 0\), there is a \(w' \in S(v) \triangleq D\) such that \(d(w', v) = x\). Since \(d(w', u) = d(u, v) + d(v, w') = 1 + x > i = d(w, u)\), \(w\) is the vertex in \(S(u) \triangleq D\) closest to \(u\). So, the new label \(L(u)\) should remain \(i\) which is correctly returned by (1.2) (Figure 5 (e)).

Case 5: \(-r \leq x = L(v) \leq -(i + 1)\). Since \(L(u) = i > 0\), there is a \(w \in S(u) \triangleq D\) such that \(d(w, u) = i\). Since \(L(v) = x < 0\), \(v\) must be covered by a vertex in \(D\) within distance \(-L(v)\). Since \(-L(v) = -x \geq i + 1 = d(w, u) + d(u, v) = d(w, v)\), \(w\) covers \(v\) within distance \(-L(v)\). Thus, the new label \(L(u)\) should remain \(i\) which is correctly returned by (1.2) (Figure 5 (f)).

Figure 6 shows the multiplication table of the binary function \(f_{2-MDS}\) for the 2-MDS problem. The row index is the first argument of \(f_{2-MDS}\) and the column index is the second argument of \(f_{2-MDS}\). Figure 7 shows an example of the Sequential \(r\)-MDS algorithm with \(r = 2\). Figure 7 (1) is a tree \(T\) with 11 vertices. When we run Algorithm 3.2 on \(T\), the vertices 11, 10, \(\cdots\), 2, 1 are deleted in that order. Figure 7 (2) shows the final labels of the vertices after Algorithm
3.2 terminates. A minimum 2-dominating set of \( T \) is the set \( \{3,8\} \). From this example, the intuitive meaning of Algorithm 3.2 is clear.

In order to give a rigorous proof of the correctness of Algorithm 3.2, we use similar techniques as in Section 3.2.

Let \( T = (V,E) \) be a tree. A labeling on \( T \) is a mapping \( L : V \to [-r,r] \). A mixed \( r \)-dominating set (md set) of \( T \) with respect to \( L \) is a subset \( D \subseteq V \) such that:

(a) \( L(v) = 0 \) implies \( v \in D \);

(b) \( L(v) = -i \) implies that either there exists a \( u \in V \) such that \( L(u) = j \) and \( j + d(u,v) \leq i \); or there exists a \( u \in D \) such that \( d(u,v) \leq i \). (In either case, we say \( v \) is bound to \( u \) in \( D \)).

(c) A vertex \( v \) with \( L(v) = j \) for some \( j \) needs not be included in \( D \), but may be included in \( D \) in order to bound the vertices with a label \( -i \).

By this definition, an \( r \)-dominating set of \( T \) is just a md set of \( T \) with respect to the labeling \( L \) where \( L(v) = -r \) for all \( v \in V \). Let \( md(T,L) \) denote the cardinality of a minimum md set of \( T \) with respect to a given labeling \( L \).

**Lemma 3.6:** Let \( T = (V,E) \) be a tree and \( L : V \to [-r,r] \) be a labeling on \( T \). Suppose \( v \) is a leaf of \( T \) and \( u \) is its parent. Let \( T' = T \setminus \{v\} \). Define a labeling \( L' \) on \( T' \) by: \( L'(u) = f_{r-MDS}(L(u),L(v)) \) and \( L'(w) = L(w) \) for other vertex \( w \) in \( T' \). Then

(1) If \( L(v) = 0 \), then \( md(T,L) = md(T',L') + 1 \);

(2) If \( L(v) \neq 0 \), then \( md(T,L) = md(T',L') \).
Proof: We only prove (1). The proof of (2) is similar (but much more tedious).

Let $D$ be a minimum md set of $T$ with respect to $L$. Define $D' = D - \{ v \}$. We will show that $D'$ is a md set of $T'$ with respect to $L'$.

If $L(u) = 0$, then $L'(u) = 0$ (from 1.2 or 2.2). $D'$ is clearly a md set of $T'$ with respect to $L'$.

If $L(u) \neq 0$, then $L'(u) = 1$ (from 1.1 or 2.1). Consider any vertex $w$ in $T'$ with $L'(w) = -i$ for some $i$. If $w$ is bound to $v$ in $D$, we have $d_T(w,v) \leq i$ and $d_{T'}(w,u) + 1 = d_T(w,v) \leq i$. Hence $w$ is bound to $u$ in $D'$. Thus, $D'$ is a md set of $T'$ with respect to $L'$.

So, $md(T',L') \leq |D'| = |D| - 1 = md(T,L) - 1$.

Now let $D'$ be a minimum md set of $T'$ with respect to $L'$. Define $D = D' \cup \{ v \}$. We need to show $D$ is a md set of $T$ with respect to $L$.

If $L(u) = 0$, then $L'(u) = 0$ (from 1.2 or 2.2). $D$ is clearly a md set of $T$ with respect to $L$.

If $L(u) \neq 0$, then $L'(u) = 1$ (from 1.1 or 2.1). Consider any vertex $w$ in $T$ with $L(w) = -i$ for some $i$. If $w$ is bound to $u$ in $D'$, we have $d_T(w,u) + 1 \leq i$ and $d_T(w,v) = d_T(w,u) + 1 \leq i$. Hence $w$ is bound to $v$ in $D$. Thus, $D$ is a md set of $T$ with respect to $L$.

So, $md(T,L) \leq |D| = |D'| + 1 = md(T',L') + 1$. □

The correctness of Algorithm 3.2 easily follows from Lemma 3.6.

The structure of the Sequential $r$-MDS algorithm (Algorithm 3.2) is almost identical to the Sequential MCS algorithm. The only difference is that the
function $f_{MCS}$ is used for the MCS algorithm and the function $f_{r-MDS}$ is used for the $r$-MDS algorithm. Thus, by using the same techniques as in Section 3.2, the Sequential $r$-MDS algorithm can be converted into an instance of the BTAC problem on the set $S = [-r, r]$ and solved by the BTAC algorithm. Since the cardinality of $S$ is $2r + 1$, by Theorem 2.3, we have:

**Theorem 3.7:** The $r$-MDS problem on trees can be solved in $O(r \log n)$ time with $O(n)$ processors on an EREW PRAM.

**Remark:** The time bound $O(r \log n)$ of the parallel $r$-MDS algorithm depends on $r$. If $r$ is large (say $r = n^{1/2}$), this algorithm becomes less efficient. In the next chapter, we will consider more general forms of the $r$-dominating set problem and provide a more sophisticated parallel $r$-MDS algorithm whose time bound does not depend on $r$. 

CHAPTER IV
R-DOMINATING SET AND P-CENTER PROBLEMS ON TREES

In this chapter, we consider more general forms of the $r$-dominating set problem and a related $p$-center problem on trees.

In Section 4.1, we introduce the definitions of these problems and give a historical review. In Sections 4.2 and 4.3, we develop an efficient parallel algorithm for the $r$-dominating set problem. Using the $r$-dominating set algorithm as a subroutine, we present an efficient parallel algorithm for the $p$-center problem in Section 4.4. The algorithm for the $r$-dominating set problem runs in $O(\log n \log \log n)$ time with $O(n)$ processors. The algorithm for the $p$-center problem runs in $O(\log n \log^2 n)$ time with $O(n)$ processors. These algorithms are within an $O(\log n \log \log n)$ factor of optimal.

The model used in this chapter is a CREW PRAM which is stronger than the EREW PRAM model used in Chapter III. The reason for using this stronger model is that a parallel merging algorithm [BH] is used in our algorithms as a subroutine and the merging algorithm requires concurrent reads from common memory. All other steps of our algorithms can be implemented on an EREW PRAM.
4.1 Definitions and Background

Let $T = (V, E)$ be a tree with $|V| = n$. We assume $T$ is embedded in Euclidean plane. The edges of $T$ are represented by straight line segments whose end points are vertices. Each edge $e \in E$ is given a length $l(e)$. This embedding enables us to talk about the points, not necessarily the vertices, on the edges. Let $A$ denote the set of all points of $T$. For any $x, y \in A$, $d(x, y)$ denotes the distance, measured along the edges of $T$, between $x$ and $y$.

We specify a set of supply points $X \subseteq A$ and a set of demand points $Y \subseteq A$ and consider following problems.

(a) The $X/Y/r$-dominating set problem: Let $r$ be a positive real number. A finite set of points $D \subseteq X$ is a $X/Y/r$-dominating set of $T$ if for all $y \in Y$

$$\min_{x \in D} d(x, y) \leq r.$$  

The $X/Y/r$-dominating set problem is to find a $X/Y/r$-dominating set $D^*$ of minimum cardinality. The dominating number, $M_T(r)$, of $T$ with radius $r$ is defined as the cardinality of $D^*$.

(b) The $X/Y/p$-center problem: Let $p$ be a positive integer. For any $p$-element subset $C = \{x_1, \ldots, x_p\} \subseteq X$, define $R_T(C) = \max_{y \in Y} \{\min_{1 \leq i \leq p} d(y, x_i)\}$. The $p$-radius of $T$ is define as $r_p = \min_C \{R_T(C) | C \subseteq X \text{ and } |C| = p\}$. A $p$-element subset $C^* \subseteq X$ such that $R_T(C^*) = r_p$ is called a $X/Y/p$-center of $T$. The $X/Y/p$-center problem is to find such a $C^*$.

Intuitively, the $X/Y/r$-dominating set problem is to find a minimum cardinality subset $D^*$ of $X$ such that every point in $Y$ is covered by a point in $D^*$ within distance $r$. The $X/Y/p$-center problem is to find a $p$-element subset $C^*$ of $X$ which minimizes the maximum of the minimum distance from any point in $Y$ to the points in $C^*$. These problems find obvious applications in fields such as
transportation science and network design. (See [CT1], [CT2], [DF], [FJ], [Hak1], [Hak2], [Han], [KH], [MTZC], [MT] etc.) The r-dominating set problem discussed in Section 3.4 is just a special case of the \( V/V/r \)-dominating set problem where the edge length \( l(e) = 1 \) for all \( e \in E \).

The study of these problems was initiated by Hakimi ([Hak1] and [Hak2]) two decades ago. Four cases of these problems are most interesting: \( X \) and \( Y \) are taken to be either the set of vertices \( V \) or the set of points \( A \). Sequential algorithms for these cases have been extensively studied. Cockayne et al. [CGH] presented a linear time algorithm for a special case of the \( V/V/1 \)-dominating set problem where \( l(e) = 1 \) for all \( e \in E \). Slater [SI] developed a linear time algorithm for the \( V/V/r \)-dominating set problem where \( r \) is an integer and \( l(e) = 1 \) for all \( e \in E \). Linear time algorithms for general \( V/V/r \) and \( A/V/r \)-dominating set problems were obtained by Kariv and Hakimi [KH]. Chandrasekaran and Tamir [CT1] developed linear time algorithms for the \( V/A/r \) and \( A/A/r \)-dominating set problems.

The \( p \)-center problems are more difficult. For \( p = 1 \) or 2, the \( p \)-center problems can be solved in linear time ([DF], [Go], [Han], and [KH]). For general \( p \), Kariv and Hakimi [KH] gave an \( O(n^2 \log n) \) algorithm for the \( V/V/p \) and \( A/V/p \)-center problems. Chandrasekaran and Tamir [CT1] presented an \( O((n \log p)^2) \) algorithm for the \( A/A/p \)-center problem. Megiddo et al. [MTZC] improved the \( V/V/p \), \( A/V/p \) and \( V/A/p \)-center algorithms to \( O(n \log^2 n) \) and gave an \( O(n \min\{p \log^2 n, n \log p\}) \) algorithm for the \( A/A/p \)-center problem. Later, Frederickson and Johnson [FJ] developed an \( O(n \log n) \) algorithm for the \( V/V/p \), \( A/V/p \) and \( V/A/p \)-center problems and an \( O(pn \log n) \) algorithm for the \( A/A/p \)-center problem. Megiddo and Tamir [MT] designed another efficient
A/A/p-center algorithm which runs in $O(n \log^3 n)$ time.

On the other hand, parallel algorithms for these problems have not been studied. The goal of this chapter is to design efficient parallel algorithms for solving these problems.

In Sections 4.2 and 4.3 we will present a parallel algorithm for solving a special case of the $V/V/r$-dominating set problem where $r$ is an integer and $l(e) = 1$ for all $e \in E$. This special case is the same problem we discussed in Section 3.4. We will simply refer this special case as the $r$-dominating set problem in Sections 4.2 and 4.3. Our new algorithm runs in $O(\log n \log n)$ time with $O(n)$ processors on a CREW PRAM. The time bound of this algorithm is independent of $r$. Thus, for large $r$, it is more efficient than the algorithm in Section 3.4.

Although we only discuss a special case for the $X/Y/r$-dominating problems, all necessary techniques for solving general $X/Y/r$-dominating set problems are presented in the algorithm for this special case. Algorithms for other $X/Y/r$-dominating set problems within the same resource bounds can be obtained by using similar techniques.

Using the parallel $X/Y/r$-dominating set algorithm as a subroutine, we will parallelize the sequential $X/Y/p$-center algorithm of Frederickson and Johnson [FJ] in Section 4.4. Our parallel algorithm for the $V/V/p$, $A/V/p$ and $V/A/p$-center problems runs in $O(\log n \log^2 n)$ time with $O(n)$ processors on a CREW PRAM. The algorithm for the $A/A/p$-center problem runs in $O(p \log n \log^2 n)$ time with $O(n)$ processors on a CREW PRAM. Compared with the time bounds of the best known sequential algorithms, all parallel algorithms presented in this
chapter are within an $O(\log \log n \log n)$ factor of optimal.

### 4.2. The Outline of the $r$-Dominating Set Algorithm

We first review the parallel $r$-dominating set algorithm from Section 3.4. In order to compute a minimum $r$-dominating set for an arbitrary tree $T_1$, first covert $T_1$ to a binary tree $T$. Next, label each leaf $v$ of $T$ by a label $L(v) = -r$ and each internal vertex $u$ of $T$ by the binary function $f_{r-MDS}$ defined in Section 3.4. Then perform the BTAC algorithm (Algorithm 2.1) on $T$ to compute the final label $L(v)$ for each vertex $v$. The minimum $r$-dominating set of $T_1$ is determined from these final labels.

The only step of this algorithm that requires $O(r \log n)$ time with $O(n)$ processors is the application of the BTAC algorithm on $T$. All other steps can be performed in $O(\log n)$ time with $O(n)$ processors. Thus, in order to obtain a more efficient algorithm, we only need to consider the following problem: Perform the BTAC algorithm on a binary tree $T$ where each leaf of $T$ is labeled by $-r$ and each internal vertex of $T$ is labeled by the function $f_{r-MDS}$.

Recall that the BTAC algorithm (Algorithm 2.1) consists of two phases. The phase 1 is a repeat loop and is executed in $K$ steps for some $K = O(\log n)$. In each step one of the actions R1, R2, R3 or COM is performed on each vertex $v$ depending on certain conditions on $v$.

Two difficulties arise in the implementation of Algorithm 2.1. The first one is timing: how does the algorithm know which action should be performed on a vertex $v$ at step $t$ of the phase 1? (More precisely, if the COMPRESS operation should be performed on $v$ at step $t$?) In Algorithm 2.1, this difficulty is resolved
by using the "unconditional jump COM" operation. The drawback of this approach is that after a vertex \( v \) becomes a pending vertex, the processor assigned to \( v \) is still associated with \( v \). Consequently, many processors are "bound" to pending vertices without doing real work. (This is the price we paid to save an \( O(\log n) \) factor in running time). In order to obtain a more efficient parallel algorithm, the processors that are "bound" to pending vertices must be "released" so that they can do some real work.

The second difficulty is a representation problem: how to represent the unary functions induced by \( f_{r-MDS} \) so that the evaluation and the composition of these unary functions can be done efficiently? In Algorithm 2.1, these unary functions are represented by vectors of \( 2r + 1 \) elements and the function value on each \( x \in [-r, r] \) is explicitly recorded. This is the reason why the algorithm takes \( O(r \log n) \) time. If the internal vertices of the computation tree are labeled by arbitrary functions, this vector recording seems the only way to represent these unary functions. In the \( r \)-dominating set problem, however, the internal vertices of \( T \) are labeled by a special function \( f_{r-MDS} \). By exploring the structure of \( f_{r-MDS} \), it is possible to find a more compact representation for these unary functions and subsequently reduce the running time of the algorithm.

We resolve the first difficulty in this section. The representation issue will be discussed in Section 4.3.

We will refer the COM action in Algorithm 2.1 as the "unconditional jump COM" operation. In this operation every vertex on a maximal chain "jumps" to its grand-parent (namely the parent of its parent). The operation in which only the odd indexed vertices "jump" to their grand-parent will be referred as the
"original COMPRESS" (or simply COMPRESS) operation. Our goal in this section is to find a technique so that the original COMPRESS operation is implemented in Algorithm 2.1 without the $O(\log n)$ penalty in running time.

In order to do so, we preprocess $T$ to compute an array $A_v[1..K]$ ($K$ is the number of iterations in phase 1) for each $v \in V$. The entry $A_v[t]$ will be in one of three forms. If $A_v[t] = \text{null}$, no action is performed on $v$ at step $t$ of phase 1 of Algorithm 2.1. If $A_v[t] = R$, one of the actions R1, R2 or R3 is performed on $v$ at step $t$ of phase 1 of Algorithm 2.1. If $A_v[t] = (J,u)$ ($J$ means "jump", $u$ is the parent of $v$), the original COMPRESS operation is performed on $v$ at step $t$ of phase 1 of Algorithm 2.1. Namely, $v$ is an odd indexed vertex on a maximal chain and $v$ jumps to its grand-parent, and $v$'s parent $u$ is pushed on $\text{stack}(v)$.

The following Timing algorithm computes the entries of $A_v$ ($v \in V$). In this algorithm the variable $D(v)$ records the number of children of $v$. $D(v)$ is initialized to 0 for the leaves of $T$ and 2 for the internal vertices of $T$. As mentioned in Section 2.2, the conditions for performing the actions R1, R2, R3, and "unconditional jump COM" can be checked from the values of $D(v)$ in constant time. For clarity, the checking for these conditions is omitted in the following Timing algorithm.

**Algorithm 4.1. Timing:**

Input: A binary tree $T = (V,E)$.

$t \leftarrow 0$;

Loop 1: repeat:

$t \leftarrow t + 1;$
In parallel for all $v \in V$ do:

(R1) If $v$ is a leaf with parent $u$ and its brother is also a leaf then

$$A_v[t] = R; \quad D(u) = 0; \quad \text{delete } v;$$

(R2) If $v$ is a leaf with parent $u$ and its brother is not a leaf then

$$A_v[t] = R; \quad D(u) = 1; \quad \text{delete } v;$$

(R3) If $v$ is a leaf and is the only child of its parent $u$ then

$$A_v[t] = R; \quad D(u) = 0; \quad \text{delete } v;$$

(COM) If $v$ is on a maximal chain and $u$ is the parent of $v$ then

$$P(v) = P(u); \quad A_v[t] = (J, u);$$

(null) If none of above then $A_v[t] = \text{null};$

Until only the root $r$ of $T$ remains (end of Loop 1);

$K = t;\quad$

label the root $r$ "alive";

Loop 2: for $t = K$ downto 1 do:

In parallel for all $v \in V$ do:

(a) If $A_v[t] = R$, then label $v$ "alive";

(b) If $A_v[t] = (J, u)$ and $v$ is "alive", then label $u$ "alive";

(c) If $A_v[t] = (J, u)$ and $v$ is not "alive", then $A_v[t] = \text{null};$

End Loop 2;

End Timing.

The Timing algorithm consists of two loops. The first loop simulates phase 1 of Algorithm 2.1. When an action is performed on a vertex $v$ at step $t$, the
entry $A_v[t]$ is filled with a value indicating the action performed. After a vertex $v$ becomes a pending vertex, the RAKE operation (actions R1, R2, or R3) can no longer be performed on $v$ in subsequent steps (since the end vertex of a pending chain has no physical children). However, the unconditional jump COM operation can still be performed on a pending chain in subsequent steps (since the condition for action COM can not distinguish a pending chain from a real chain). This means that some entries in the arrays $A_v$ are filled with wrong information. These wrong entries will be corrected in the second loop.

A vertex $v$ is called "alive" at step $t$ if it neither has been deleted by the RAKE operation nor becomes a pending vertex after step $t$ of phase 1 of Algorithm 2.1. An "alive" vertex $v$ becomes "dead" at step $t$ if it is deleted either by the RAKE or by the original COMPRESS operation (or equivalently, it becomes a pending vertex in the unconditional jump COM operation) at step $t$ of phase 1 of Algorithm 2.1.

The loop 2 of the Timing algorithm traces phase 1 of Algorithm 2.1 backward in order to identify "alive" vertices at each step $t$. At step $K$ (the last step of the loop 1), only the root $r$ of $T$ is "alive". If $A_v(t)=R$, $v$ is deleted at step $t$ by the RAKE operation and is "alive" at step $t-1$. So, it is labeled as "alive" by the statement (a) of the loop 2. If $A_v[t]=(J,u)$ and $v$ is "alive" at step $t$, the COM operation is performed on $v$ at step $t$ and its parent $u$ is deleted by COM at step $t$. Thus, $u$ is "alive" at step $t-1$. So, $u$ is labeled "alive" by the statement (b). If $A_v[t]=(J,u)$ and $v$ is not "alive" at step $t$, $v$ is a pending vertex at step $t$ and no action should be performed on $v$ at step $t$. So, the entry $A_v[t]=(J,u)$ is wrong and the statement (c) sets $A_v[t]$ to null.
Thus, after performing the Timing algorithm on $T$, the entry $A_v[t]$ indeed indicates which action should be performed on $v$ at step $t$ of phase 1 in Algorithm 2.1. If we assign one processor for each vertex, both loop bodies of the Timing algorithm can be performed in constant time. So, the Timing algorithm runs in $O(\log n)$ time with $O(n)$ processors.

In the next section, we assume that the Timing algorithm has been performed on $T$. Hence, the algorithm knows which operation (RAKE or the original COMPRESS) should be performed on each $v \in V$ at step $t$ of phase 1 in Algorithm 2.1. In particular, if a vertex $v$ is deleted by the original COMPRESS operation at step $t$, the processor assigned to $v$ will be "released" from $v$ and it will be assigned to another vertex to do some real work.

### 4.3 Representation of the Unary Functions

In this section, we discuss how to represent the unary functions induced by $f_{r-MDS}$ so that the evaluation and the composition of these unary functions can be performed efficiently.

Define $g_i(x) = f_{r-MDS}(i,x)$ and $h_i(x) = f_{r-MDS}(x,i)$ for $i \in [-r,r]$. Let $\Psi = \{g_i, h_i \mid -r \leq i \leq r\}$ be the set of unary functions induced by $f_{r-MDS}$. Let $\Pi = \{G \mid G = f_1 \circ f_2 \circ \cdots \circ f_k \text{ for some } k, \text{ and } f_j \in \Psi \text{ for } 1 \leq j \leq k\}$. Each $G \in \Pi$ is an unary function $G: [-r,r] \rightarrow [-r,r]$. If $G$ is a composition of $k$ unary functions in $\Psi$, $G$ is called a $k$-fold function.

For each $G \in \Pi$, we partition the domain set $[-r,r]$ of $G$ into ordered sub-intervals $\{[a_1, b_1], \ldots, [a_k, b_k]\}$ so that $G$ can be represented by a simple formula on each $[a_i, b_i]$ ($1 \leq i \leq k$). We will show that it is always possible to find a
partition such that \( G \) restricted to \([a_i, b_i]\) is either a constant or a translation by an integer \( \alpha \) (namely \( G(x) = x + \alpha \) for all \( x \in [a_i, b_i] \)). Thus, \( G \) can be represented as \( \{<[a_1, b_1], r_1>, \ldots, [<a_k, b_k], r_k>\} \) where \( \{[a_i, b_i]\}_{1 \leq i \leq k} \) is an ordered partition of the interval \([-r, r]\) and each \( r_i \) is either a constant or of the form \( +\alpha \) for some integer \( \alpha \). If \( r_i \) is a constant, \( G \) takes the constant value \( r_i \) on \([a_i, b_i]\). If \( r_i \) is of the form \( +\alpha \), \( G \) is a translation by \( \alpha \) on \([a_i, b_i]\). This representation is called the interval representation of \( G \) and denoted by \( I(G) \). An entry of the form \( <[a_i, b_i], +\alpha> \) is called a translation interval.

A sequence of integers \( x_1, \ldots, x_l \) (\( x_l \in [-r, r] \), not necessarily distinct) is cyclically ordered if we can travel the circle shown in Figure 8 in counterclockwise direction and meet \( x_1, x_2, \ldots, x_l \) in that order.

\[ \text{Figure 8. The circle defining the cyclic order.} \]

**Lemma 4.1:** Any \( k \)-fold function \( G \in \Pi \) can be represented by \( I(G) = \{<[a_1, b_1], r_1>, \ldots, <[a_q, b_q], r_q>\} \) such that \( q \leq 2k + 1 \) and there is at most one translation interval in \( I(G) \). In addition, the sequence \( r_1, \ldots, r_q \) is
cyclically ordered. (If \(<[a_i, b_i], r_i>\) is a translation interval with \(r_i = +\alpha, r_i\) represents two numbers \(a_i + \alpha\) and \(b_i + \alpha\) in the sequence).

**Proof:** The lemma is proved by induction on \(k\). For \(k = 1\), we consider all 1-fold functions in \(\Pi\). The function \(f_{r-MDS}\) is defined in formulas (1) and (2) in Section 3.4 by fixing the first variable. It can also be defined by fixing the second variable as follows:

(1') For each fixed \(i \geq 0\):

1.1') \(f_{r-MDS}(x,i) = x\), if \(-i \leq x \leq i\);

1.2') \(f_{r-MDS}(x,i) = i + 1\), if \(-r \leq x \leq -i - 1\) or \(i + 1 \leq x \leq r\).

(2') For each fixed \(-i < 0\):

2.1') \(f_{r-MDS}(x,-i) = x\), if \(-i + 1 \leq x \leq i - 1\);

2.2') \(f_{r-MDS}(x,-i) = -i + 1\), if \(-r \leq x \leq -i\) or \(i \leq x \leq r\).

From these formulas, we have following interval representations:

\(I(g_0) = \{<[-r, r], 0]\};\)

\(I(g_i) = \{<[-r, -(i + 1)], i>, <[-i, i - 1], i + 1>, <[i, r], i>\};\)

\(I(g_{-i}) = \{<[-r, -(i + 1)], -(i)>, <[-i, i - 1], i + 1>, <[i, r], -(i)>,\};\)

\(I(h_0) = \{<[r, -1], 1>, <[0, 0], 0>, <[1, r], 1>\};\)

\(I(h_i) = \{<[r, -(i + 1)], (i + 1)>, <[-i, i], 0>, <[i + 1, r], (i + 1)>,\};\)

\(I(h_{-i}) = \{<[r, -i], -(i + 1)>, <[-i + 1, i - 1], 0>, <[i, r], -(i + 1)>,\};\)

These representations are shown in Figure 9. The claims of the lemma are clearly true for these functions.
Figure 9. Representations of 1-fold functions.
Figure 10. Representation of the composite function in case 1.

Figure 11. Representation of the composite function in case 2.
Let \( G = f_1 \circ \cdots \circ f_k \) be a \( k \)-fold function in \( \Pi \) and let \( H = f_2 \circ \cdots \circ f_k \). So, \( G = f_1 \circ H \). By the induction hypothesis, \( H \) can be represented as

\[ I(H) = \{ [< a_1, b_1 >, r_1 >, \cdots , [< a_q, b_q >, r_q >] \} \]

where \( q \leq 2(k-1) + 1 \), \( I(H) \) has at most one translation interval and the sequence \( r_1 \cdots r_q \) is cyclically ordered.

Suppose \( I(f_1) = \{ [c_1, d_1], s_1 >, [< c_2, d_2 ], + 1 >, [< c_3, d_3 ], s_3 > \} \). (If \( I(f_1) \) is of other form, the proof is similar).

Case 1: \( I(H) \) has no translation interval and each \( r_i (1 \leq i \leq q) \) is a constant.
In this case, \( I(G) = \{ [< a_1, b_1 ], t_1 >, \cdots , [< a_q, b_q ], t_q > \} \) where \( t_i = s_i \) if \( r_i \in [c_1, d_1] \); \( t_i = r_i + 1 \) if \( r_i \in [c_2, d_2] \); \( t_i = s_3 \) if \( r_i \in [c_3, d_3] \). The representation of \( G = f_1 \circ H \) is shown in Figure 10. Clearly, \( I(G) \) has \( q < 2k + 1 \) intervals, \( I(G) \) has no translation intervals and the sequence \( t_1, \cdots , t_q \) is cyclically ordered.

Case 2: \( I(H) \) has one translation interval, say \( [< a_i, b_i >, + \alpha >] \). Consider the intersection of the interval \( [a_i + \alpha, b_i + \alpha] \) with \( [c_j, d_j] \) \( (j = 1,2,3) \). Suppose \( c_1 \leq a_i + \alpha \leq d_1 \) and \( c_3 \leq b_i + \alpha \leq d_3 \) (other cases are similar). We break \( [a_i, b_i] \) into three sub-intervals \( [a_i, d_1 - \alpha] \), \( [c_2 - \alpha, d_2 - \alpha] \) and \( [c_3 - \alpha, b_i] \) (see Figure 11). Then \( G \) can be represented as:

\[ I(G) = \{ [< a_i, b_i ], s_1 >, \cdots , [< a_i - 1, b_i - 1 ], s_1 >, \]
\[ [< a_i, d_1 - \alpha ], s_1 >, \]
\[ [< c_2 - \alpha, d_2 - \alpha ], + (\alpha + 1 ) >, \]
\[ [< c_3 - \alpha, b_i ], s_3 >, \]
\[ [< a_i + 1, b_i + 1 ], s_3 >, \cdots , [< a_q, b_q ], s_3 > \} \]. So, \( I(G) \) has \( q - 1 + 3 = q + 2 \leq 2k + 1 \) intervals and one translation interval \( [< c_2 - \alpha, d_2 - \alpha ], + (\alpha + 1 ) > \). It is easy to see that the sequence \( s_1, \cdots , s_1, (c_2 - \alpha), + (\alpha + 1 ), (d_2 - \alpha), + (\alpha + 1), s_3, \cdots , s_3 \) is cyclically ordered. This completes the induction. □

We are now ready to implement phase 1 of Algorithm 2.1. The unary functions induced by \( f_{r-MDS} \) are represented by their interval representation. Originally each vertex \( v \) is assigned a processor \( P_v \). Suppose that during
execution, some vertex $v$ is deleted by COMRESS and its parent $u$ is associated with the composite function $F(u)\circ F(v)$. Whenever this occurs, the processors assigned to $v$ are reassigned to $u$. Hence, at any step during the computation, if a vertex $u$ is associated with a $k$-fold function, $k$ processors will be assigned to $u$.

If action R1 is performed on a vertex $v$, both final labels $L(v)$ and $L(w)$ ($w$ is $v$'s brother) are known. The final label $L(u)$ of $v$'s parent $u$ can be computed in constant time by the processor $P_u$.

If action R2 is performed on $v$, $L(v)=i$ (for some $i$) is known. After $v$ is deleted, its parent $u$ will be associated with an unary function $F^1(u)$ which is either $g_i$ or $h_i$. The interval representation $I(F^1(u))$ is computed in constant time by the processor $P_u$.

If action R3 is performed on $v$, $v$'s parent $u$ is associated with a $k$-fold function $F^1(u)$ for some $k$ and the final label $L(v)$ is known. In order to compute the final label $L(u)=F^1(u)(L(v))$, we search the interval representation $I(F^1(u))$ to find the entry $<[a_i,b_i],r_i>$ such that $L(v)\in[a_i,b_i]$. Then $L(u)$ can be determined from $r_i$. Since $k$ processors are assigned to $u$ and $I(F^1(u))$ has at most $2k+1$ intervals, this search can be done in constant time by the processors assigned to $u$.

We next consider the COMRESS operation. Let $v$ and $u$ be two vertices on a maximal chain $C$ where $u$ is the parent of $v$ and $v$ is deleted by the COMRESS operation. Suppose $v$ is associated with a $k$-fold function $F^1(v)$, $u$ is associated with an $m$-fold function $F^1(u)$. We must compute the interval representation $I(F^1(u)\circ F^1(v))$ from $I(F^1(v))$ and $I(F^1(u))$. Suppose $I(F^1(v))=$
Furthermore, suppose \(<[a_1, b_1], \alpha_1>\) is the translation interval of \(I(F^1(v))\). There are \(k\) processors assigned to \(v\). There are \(p \leq 2k + 1\) (\(q \leq 2m + 1\)) intervals in \(I(F^1(v))\). The intervals \([c_j, d_j]\) (\(1 \leq j \leq q\)) such that \([c_j, d_j] \cap [a_i + \alpha, d_i + \alpha] \neq \emptyset\) can be identified in constant time by the processors assigned to \(u\). Suppose that they are \([c_1, d_1], \ldots, [c_q, d_q]\). We break the interval \([a_1, b_1]\) into sub-intervals \([a_1, d_1], [c_1 - \alpha, d_1 - \alpha], \ldots, [c_q - \alpha, b_q]\) (Figure 12). The value of the composite function on these intervals can be determined in constant time from \(s_i, \ldots, s_j\). Next we determine the values of the composite function on the intervals \([a_1, b_1], \ldots, [a_{i-1}, b_{i-1}], [a_i, b_{i+1}], \ldots, [a_p, b_p]\). By using a slight modification of the parallel merging algorithm of Borodin and Hopcroft [BH], we can merge the cyclically ordered sequence \(r_1, \ldots, r_i, \ldots, r_{i+1}, \ldots, r_p\) into the ordered list of intervals \([c_1, d_1], \ldots, [c_q, d_q]\).

On a CREW PRAM, the merging can be done in \(O(\log\log(\max\{p, q\}))\) time with \(m + k\) processors assigned to \(u\) and \(v\) [BH]. (This is the only place where the CREW PRAM model is required. Otherwise, our algorithm would be able to run on an EREW PRAM). Therefore, the algorithm knows, for each \(l\) (\(l = 1, \ldots, i - 1, i + 1, \ldots, p\)), the interval \([c_{j_l}, d_{j_l}]\) containing \(r_l\). Hence, the value of the composite function on \([a_1, b_1]\) can be determined from \(s_{j_1}\). So, \(I(F^1(u) \circ F^1(v))\) can be computed in \(O(\log\log n)\) time with \(m + k\) processors assigned to \(u\) and \(v\). Since \(I(F^1(v))\) has at most \(2k + 1\) intervals and there are \(k\) processors assigned to \(v\), \(I(F^1(v))\) can be pushed onto the proper stack in constant time.
The implementation of the phase 2 of Algorithm 2.1 is similar. The processors are assigned to the vertices in a way that if \((u,I(F^1(u)))\) is popped from \(stack(v)\) and \(F^1(u)\) is a \(k\)-fold function, \(k\) processors will be assigned to \(u\). The pop operation and the evaluation of \(L(u) = F(u)(L(v))\) can be done in constant time by the processors assigned to \(u\).

![Figure 12. The composite function of \(F(u)\) and \(F(v)\).](image)

Since both loop bodies of Algorithm 2.1 can be implemented in \(O(\log \log n)\) time with \(O(n)\) processors and each loop is executed at most \(O(\log n)\) iterations, we have:

**Theorem 4.2:** The \(V/V/r\)-dominating set problem on trees where all edges have length \(l(e) = 1\) can be solved in \(O(\log \log n \log n)\) time with \(O(n)\) processors on a CREW PRAM. □

**Remark:** The method presented above can be used to design parallel algorithms for other \(X/Y/r\)-dominating set problems. Linear time sequential algorithms for the \(V/A/r\), \(A/V/r\) and \(A/A/r\)-dominating set problems were
presented in [CT1] and [KH]. These algorithms can be developed by using similar approach as presented in Section 3.4. (The function \( f_{r-MDS} \) describing the label modification rules is different for each case and the edge length \( l(e) \) must be considered in \( f_{r-MDS} \).) Thus, a slight modification of the parallel algorithm we developed above can solve these problems within the same time and processor bounds.

4.4 Parallel Algorithm for p-Center Problem

In this section we discuss the parallel algorithm for the \( X/Y/p \)-center problem. We first review the sequential algorithm developed in [FJ] and then sketch the parallel algorithm for this problem.

All efficient sequential algorithms for the \( X/Y/p \)-center problem are based on a fundamental observation: the set \( R \) of candidates for the \( p \)-radius \( r_p \) is a nicely structured finite set, and \( r_p \) can be computed by an efficient search in \( R \). The following results are proved in [CT1], [FJ], [KH], and [MTZC]:

(a) For the \( V/V/p \)-center problem, \( R = \{d(x,y) | x, y \in V \} \);
(b) For the \( A/V/p \)-center problem, \( R = \{d(x,y)/2 | x, y \in V \} \);
(c) For the \( V/A/p \)-center problem, \( R = \{d(x,y), d(x,y)/2 | x, y \in V \} \);
(d) For the \( A/A/p \)-center problem, \( R = \{d(x,y)/2k | x, y \in V, 1 \leq k \leq p \} \).

For a fixed tree \( T \), the dominating number \( M_T(r) \) (as defined in Section 4.1) is obviously a non-increasing function of \( r \). So, the \( p \)-radius \( r_p \) of \( T \) can be computed as follows. Select the median \( r_1 \) in \( R \), and use the \( X/Y/r \)-dominating set algorithm to compute \( M_T(r_1) \). If \( M_T(r_1) \leq p \) (in this case we say \( r_1 \) is feasible), then \( r_p \leq r_1 \). Hence the members in \( R \) that are greater than \( r_1 \) can be
discarded. If $M_T(r_1) > p$ (in this case we say $r_1$ is *infeasible*), then $r_p > r_1$ and the members in $R$ that are less than or equal to $r_1$ can be discarded. The algorithm continues in a binary search fashion until only one member in $R$ remains. This member is $r_p$. The $p$-center is then computed by using the $X/Y/r$-dominating set algorithm with $r = r_p$.

The cardinality of $R$ is $O(n^2)$ ($O(pn^2)$ for the $A/A/p$-center problem). If the members of $R$ are explicitly computed, the sequential time needed for solving the problem is at least $O(n^2)$. A considerable effort has been made in order to improve this time bound. The basic strategy is to find a good data structure for $R$ so that the members of $R$ are not explicitly listed and $R$ can be searched efficiently.

The best sequential $p$-center algorithm is developed in [FJ]. It runs in $O(n \log n)$ time for $V/V/p$, $A/V/p$, $V/A/p$-center problem and in $O(pn \log n)$ time for the $A/A/p$-center problem. By using the existing parallel selection, sorting, and merging algorithms, this sequential algorithm can be modified to run on a PRAM. We assume that readers are familiar with [FJ].

Our parallel algorithm for the $V/V/p$, $A/V/p$, $V/A/p$-center problem runs in $O(p \log \log n \log^2 n)$ time with $O(n)$ processors on a CREW PRAM. The parallel algorithm for the $A/A/p$-center problem runs in $O(p \log \log n \log^2 n)$ time with $O(n)$ processors on a CREW PRAM. We only discuss the $V/V/p$-center algorithm. Algorithms for other cases are similar.

Given a tree $T = (V, E)$, we first build a data structure $R(T)$ for the set $R = \{d(x, y) | x, y \in V\}$. Let $v$ be a vertex of $T$. We want to "break" $T$ into two subtrees at $v$. More precisely, $v$ is replaced by two new vertices $v_1, v_2$ and each
edge \((v,w)\in E\) is replaced either by \((v_1,w)\) or \((v_2,w)\), then two new trees \(T_1\) and \(T_2\) result. Let \(T_1=(V_1,E_1)\) and \(T_2=(V_2,E_2)\) where \(v_1\in V_1\) and \(v_2\in V_2\). We generate the data structure \(R(T)\) as follows: First construct the set of intervertex distances for pairs in \((V_1-\{v_1\})\times(V_2-\{v_2\})\). Then apply the procedure recursively on \(T_1\) and \(T_2\).

In order to represent the set \(R=\{d(x,y)|x\in V_1-\{v_1\}, y\in V_2-\{v_2\}\}\), we introduce the following concepts. Given two lists \(A\) and \(B\) of numbers, the **cartesian matrix** of \(A\) and \(B\), denoted by \(A+B\), is the set \(\{a+b|a\in A, b\in B\}\). The data structure for \(A+B\) is simply a pair of lists \((A,B)\). (Note that the cardinality of the set \(A+B\) is \(|A|\cdot|B|\) while the cardinality of the representation \((A,B)\) is only \(|A|+|B|\).) If both \(A\) and \(B\) are sorted lists, \(A+B\) is called a **sorted cartesian matrix**. For any \(x\in V_1-\{v_1\}\) and \(y\in V_2-\{v_2\}\),
\[d(x,y)=d(x,v_1)+d(y,v_2).\]
So, \(R\) can be represented by a sorted cartesian matrix \(L_1(v)+L_2(v)\) where \(L_1(v)\) is the sorted list of \(\{d(w,v_1)|w\in V_1-\{v_1\}\}\) and \(L_2(v)\) is the sorted list of \(\{d(w,v_2)|w\in V_2-\{v_2\}\}\). Thus, \(R(T)\) can be defined by the recursive formula:
\[R(T)=R(T_1)\cup R(T_2)\cup (L_1(v),L_2(v)).\]

In order to achieve an efficient representation for \(R(T)\), \(T\) must be decomposed into two subtrees \(T_1\) and \(T_2\) of about equal size. A decomposition \((v,T_1,T_2)\) of \(T\) is called a **balanced decomposition** if \(n/3\leq |T_1|\leq 2n/3\). The following lemma will help us to find a balanced decomposition of \(T\).

**Lemma 4.3:** For any tree \(T\) of \(n\) vertices there exists a vertex \(v\) in \(T\) such that either:

1. \(n/3\leq |T(v)|\leq 2n/3\) \((T(v)\) denotes the subtree of \(T\) rooted at \(v\)); or
(2) \(|T(v)| > 2n/3\) and if \(v_1, \ldots, v_k\) are children of \(v\) then \(|T(v_i)| < n/3\) for \(1 \leq i \leq k\).

**Proof:** Initialize \(v\) to be the root \(t\) of \(T\). Suppose \(v_1, \ldots, v_k\) are children of \(v\). If there is a \(v_i\) such that \(|T(v_i)| > 2n/3\), then replace \(v\) by \(v_i\) and repeat the procedure on \(v_i\). If there is a \(v_i\) such that \(n/3 \leq |T(v_i)| \leq 2n/3\), then \(v_i\) satisfies (1). Otherwise \(|T(v_i)| < n/3\) for \(1 \leq i \leq k\), and \(v\) satisfies (2). \(\Box\)

We next describe how to find a balanced decomposition of \(T\) in parallel. By using the algorithm of Tarjan and Vishkin [TV1], the set of numbers \(|T(u)|\) for \(u \in V\) can be computed in \(O(\log n)\) time with \(O(n)\) processors. Then we sort the set \(|T(u)|\) for \(u \in V\). This can be done in \(O(\log n)\) time with \(O(n)\) processors [Col]. The first vertex \(v\) in the sorted list satisfying lemma 4.3 can be selected in constant time with \(O(n)\) processors. If (1) of lemma 4.3 is true, let \(T_1 = T(v)\) and \(T_2\) be the tree obtained from \(T\) by deleting \(T_1\). Then \((v, T_1, T_2)\) is a balanced decomposition. If (2) of lemma 4.3 is true, let \(v_1, \ldots, v_k\) be children of \(v\). Since \(|T(v_i)| < n/3\) for all \(i\), there is an integer \(m\) such that \(n/3 \leq \sum_{i=1}^{m} \frac{n}{2} \leq 2n/3\). \(m\) can be found in \(O(\log n)\) time with \(O(n)\) processors by using the well known "partial sum" algorithm [TV1]. Let \(T_1 = \bigcup_{1 \leq i \leq m} T(v_i) \cup \{v\}\) and let \(T_2\) be the tree obtained from \(T\) by deleting \(T_1\). Then \((v, T_1, T_2)\) is a balanced decomposition of \(T\).

For \(T_1\), the set \(|d(w, v)| w \in V_1 - \{v\}\) can be computed in \(O(\log(|T_1|))\) time with \(O(|T_1|)\) processors [TV1]. These distances can be sorted in parallel in \(O(\log(|T_1|))\) time with \(O(|T_1|)\) processors [Col]. Similar operation is also performed on \(T_2\). Thus, a balanced decomposition \((v, T_1, T_2)\) and the sorted lists \(L_1(v)\) and \(L_2(v)\) can be computed in \(O(\log n)\) time with \(O(n)\) processors. Then
we apply the algorithm recursively on $T_1$ and $T_2$. Since the depth of the recursive calls is at most $\log_{3/2} n$, the total time needed for building the data structure $R(T)$ is $O(\log^2 n)$ with $O(n)$ processors.

We next turn to the techniques for searching $R(T)$, which is a set of sorted cartesian matrices. For simplicity we assume that these matrices are padded out so that they are square in shape and with the dimension being a power of 2. This can be done by concatenating large values onto the end of the lists that define the matrices. Since the matrices in $R(T)$ are almost square, the padding will increase the resource bounds only by a constant factor.

The algorithm performs a sequence of iterations which includes matrices in $R(T)$ of smaller size into consideration on each iteration. The matrices that are active are divided into smaller sub-matrices called cells. In each iteration, two representative values in $R(T)$ are chosen from active cells. These representative values are tested for feasibility and used to discard certain cells from further consideration.

Initially all matrices are inactive and let the current cell dimension be the dimension of the largest matrix in $R(T)$. At the beginning of each iteration, all matrices whose dimension equals the current cell dimension are changed from inactive to active. Each newly activated cell and each active cell remaining from the last iteration is divided into four square cells of equal size. Consider the set $S_s$ consisting of the smallest elements from these divided cells. Select the median $x_s$ from $S_s$. Consider the set $S_l$ consisting of the largest elements from these divided cells. Select the median $x_l$ from $S_l$. Both $x_s$ and $x_l$ are tested for feasibility. If $x_s$ is feasible, we discard all active cells whose smallest element is larger
than \( z_s \). If \( x_s \) is not feasible, we discard all active cells whose largest element is less than \( x_s \). Perform similar operation with regard to \( x_l \). This completes one iteration. The current cell dimension is cut to half and the algorithm proceeds to next iteration. After a number of iterations, all active cells consist of a single element and all matrices have been made active. When this occurs, the iterations continue as before but without cell division. After \( O(\log n) \) iterations, a single element in \( R \) remains. This element is the \( p \)-radius \( r_p \).

The crucial fact that makes this algorithm efficient is that the number of active cells at any iteration is bounded by \( O(n \log n) \) [FJ].

To implement this algorithm in parallel, we first assume that there are \( O(n \log n) \) processors. Each active cell is assigned a processor. An active cell is represented by two pairs of pointers into the lists of the cartesian matrix defining the cell. Cell division involves the creation of these pointers and can be performed in constant time. The smallest and the largest element in a cell can be computed in constant time by the processor assigned to the cell. The selection for the medians \( x_l \) and \( x_s \) can be done by using the parallel selection algorithm in [AKSS] in \( O(\log \log(n \log n)) = O(\log \log n) \) with \( O(n \log n) \) processors. If there are only \( O(n) \) processors, the same task can be performed in \( O(\log \log n \log n) \) time (since one processor can always simulate \( k \) processors in \( O(k) \) time). The feasibility tests for \( x_l \) and \( x_s \) take \( O(\log \log n \log n) \) time with \( O(n) \) processors by using the \( V/V/\tau \)-dominating set algorithm developed in previous sections. Thus, each iteration can be performed in \( O(\log \log n \log n) \) time with \( O(n) \) processors. Hence, we have:
Theorem 4.4: The $V/V/p$-center problem can be solved in $O(\log \log n \log^2 n)$ time with $O(n)$ processors on a PRAM. □

Remark: The $A/V/p$ and $V/A/p$-center problems can be solved by similar parallel algorithm within the same time and processor bounds. For the $A/A/p$-center problem, since the set $R$ of candidates for $r_p$ has size $O(pn^2)$, the parallel algorithm for this problem takes $O(p \log \log n \log^2 n)$ time with $O(n)$ processors on a CREW PRAM.
CHAPTER V
RECOGNITION AND DECOMPOSITION OF TWO TERMINAL SERIES PARALLEL GRAPHS

5.1 Introduction

The BTAC methodology can also be used to design efficient parallel algorithms for solving problems on other classes of graphs, provided that the graphs can be represented by a binary tree. We will study the class of Two Terminal Series Parallel (TTSP) graphs in Chapter V and Chapter VI.

The TTSP graphs find important applications in fields such as electrical networks and scheduling problems ([D], [MS], [Si], and [La]). Sequential algorithms for solving TTSP graph problems have been extensively studied. Duffin [D] gave a forbidden subgraph characterization of TTSP graphs. Valdes, Tarjan, and Lawler [VTL] presented a linear time sequential algorithm which recognizes TTSP graphs and constructs their decomposition trees. Bern, Lawler, Wang [BLW] and Takamizawa, Nishizeki, Saito [TNS] developed algorithms for solving various combinatorial problems on TTSP graphs. If the input is given by the decomposition trees, all the above algorithms run in linear time.

In this chapter, we develop a parallel algorithm which determines whether a given acyclic multidigraph $G$ is a TTSP graph. If $G$ is not a TTSP graph, the algorithm answers "no". If $G$ is a TTSP graph, the algorithm constructs a
decomposition tree of $G$. On a graph $G$ with $n$ vertices and $m$ edges, this algorithm runs in $O(\log^2 n + \log m)$ time with $O(n + m)$ processors on an EREW PRAM. Some interesting properties of the TTSP graphs are derived in order to facilitate fast parallel computation.

The TTSP graphs and their decomposition trees are defined in Section 5.2. In Section 5.3, we outline the basic strategy of our recognition and decomposition algorithm. In Section 5.4, the decomposition of some special TTSP graphs is discussed. The decomposition of general TTSP graphs is presented in Section 5.5. The implementation of the algorithm is sketched in Section 5.6.

Using the decomposition tree of TTSP graphs as input, we will apply generalized BTAC methodology to design parallel algorithms for solving TTSP graph problems in the next chapter.

5.2 Definitions

A directed graph (digraph) $G = (V, E)$ consists of a set of vertices $V = V(G)$ and a set of edges $E = E(G)$. An edge is an ordered pair of vertices, denoted by $(u, v)$ or $e(u, v)$, and is called an edge from $u$ to $v$. We also say that $(u, v)$ is an edge "leaving" $u$ and "entering" $v$. If multiple edges are allowed between the same pair of vertices, $G$ is called a multidigraph. $G_1 = (V_1, E_1)$ is called a subgraph of $G$, denoted by $G_1 \subseteq G$, if $G_1$ is a digraph and $V_1 \subseteq V$ and $E_1 \subseteq E$. If $V_1 = V$, $G_1$ is called a spanning subgraph of $G$. If a spanning subgraph $G_1$ of $G$ is a tree, it is called a spanning tree of $G$. The subgraph induced by a subset $V_1 \subseteq V$ is the subgraph $G_1 = (V_1, E_1)$ of $G$ where $E_1 = \{(u, v) \in E \mid u, v \in V_1\}$. 
The **indegree**, \( \text{indeg}(v) \), of a vertex \( v \) is the number of edges "entering" \( v \). The **outdegree**, \( \text{outdeg}(v) \), of \( v \) is the number of edges "leaving" \( v \). If \( \text{indeg}(v) = 0 \), \( v \) is called a **source**. If \( \text{outdeg}(v) = 0 \), \( v \) is called a **sink**. A path \( P \) is a sequence of vertices \( v_1 \cdots v_k \) such that \((v_i,v_{i+1}) \in E\) for \( 1 \leq i < k \). We denote by \( P_{uv} \) a path from \( u \) to \( v \). If \( v_1 = v_k \), \( P \) is a **cycle**. If \( G \) contains no cycle, \( G \) is called an **acyclic multidigraph**, abbreviated as a **multidag**.

**Definition 5.1:** A multidag \( G \) is a **two terminal multidag** if \( G \) has exactly one source and exactly one sink.

**Lemma 5.1:** Let \( G \) be a two terminal multidag with source \( s \) and sink \( t \).

Let \( v \) be a vertex of \( G \) such that \( v \notin \{s,t\} \). Then there exists a path in \( G \) from \( s \) to \( t \) passing through \( v \).

**Proof:** Since \( v \neq t \), there is an \( w \) such that \((v,w)\) is an edge in \( G \). Repeat this process until reach \( t \). This gives a path from \( v \) to \( t \). Similarly, we can find a path from \( s \) to \( v \). □

Let \( G = (V,E) \) be a two terminal multidag with the source \( s \) and the sink \( t \).

For each vertex \( v \neq t \), select an arbitrary edge \( e_v = (v,u) \) for some \( u \) (since \( t \) is the only sink of \( G \) and \( v \neq t \), such an edge exists). We designate \( u \) as the parent of \( v \). Let \( E_1 = \{e_v \mid v \in V, v \neq t\} \) be the set of edges selected. The subgraph \( T = (V,E_1) \) is clearly a spanning tree of \( G \). If an edge \((u,v) \in E\) is in \( T \), it is called a **tree edge**. \( v \) is the **parent** of \( u \) in \( T \). \( u \) is a **child** of \( v \) in \( T \). If there is a path in \( T \) from \( u \) to \( v \), \( u \) is a **descendent** of \( v \) in \( T \) and \( v \) is an **ancestor** of \( u \) in \( T \). We denote by \( T(x) \) the subtree of \( T \) rooted at \( x \). Let \( u \) and \( v \) be two vertices and let \( P_{ut} \) (\( P_{vt} \)) be the path in \( T \) from \( u \) (\( v \)) to the root \( t \). The first common vertex of \( P_{ut} \) and \( P_{vt} \) is called the **nearest common ancestor** of \( u \) and \( v \) in...
denoted by $NCA_T(u,v)$. The path in $T$ from $s$ to $t$ is called the main path of $T$, denoted by $MP(T)$. Let $x$ be a vertex of $T$ and let $y_1, \cdots, y_k$ be children of $x$ in $T$. We say $y_i$ is the heavy child of $x$ if $|T(y_i)| = \max\{|T(y_j)| \mid 1 \leq j \leq k\}$ (break ties arbitrarily). A path $P=v_1 \cdots v_r$ of $T$ is called a heavy path if $v_i$ is the heavy child of $v_{i+1}$ for all $1 \leq i < r$.

The class of two terminal series parallel (TTSP) graphs and their decomposition trees is defined recursively.

**Definition 5.2:**

1. A digraph consisting of two vertices $u$, $v$ and a single edge $(u,v)$ is a primitive TTSP graph with $u$ as the source and $v$ as the sink.

2. If $G_1$ and $G_2$ are two TTSP graphs with sources $s_1, s_2$ and sinks $t_1, t_2$, respectively, so is the graph obtained by either of the following two operations:

   a) **Parallel composition** (denoted by $G_1//G_2$): Identify $s_1$ with $s_2$ and identify $t_1$ with $t_2$. The resulting graph has $s_1(=s_2)$ as the source and $t_1(=t_2)$ as the sink.

   b) **Series composition** (denoted by $G_1*G_2$): Identify $t_1$ with $s_2$. The resulting graph has $s_1$ as the source and $t_2$ as the sink.

By the definition, it is easy to see that any TTSP graph is a two terminal multidag.

**Definition 5.3:**

1. The tree consisting of a single vertex labeled by $e(u,v)$ is a decomposition tree of the primitive TTSP graph $G=\langle\{u,v\},\{e(u,v)\}\rangle$. 
(2) Let $H_1, H_2$ be the decomposition trees of the TTSP graphs $G_1, G_2$, respectively.

(a) Let $H_p$ be the tree with the root $r$ labeled by $p$ and with $H_1$ as the left child of $r$ and with $H_2$ as the right child of $r$. Then $H_p$ is a decomposition tree of the TTSP graph $G_p = G_1 \parallel G_2$.

(b) Let $H_s$ be the tree with the root $r$ labeled by $s$ and with $H_1$ as the left child of $r$ and with $H_2$ as the right child of $r$. Then $H_s$ is a decomposition tree of the TTSP graph $G_s = G_1 \ast G_2$.

In order to avoid confusion, the vertices of $H$ are called nodes.

If $H$ is a decomposition tree of a TTSP graph $G$, the leaf nodes of $H$ are labeled by the edges of $G$. An internal node $x$ of $H$ is labeled either by $p$ or by $s$ indicating which composition is used to construct a new TTSP graph from the TTSP graphs represented by the children of $x$ in $H$. Figure 13 shows a TTSP graph $G$ and a decomposition tree $H$ of $G$.

Since the parallel composition is symmetric and associative and the series composition is associative, the decomposition tree of a TTSP graph is not unique. If $G$ has $m$ edges, any decomposition tree of $G$ consists of $m$ leaf nodes and total $2m - 1$ nodes.

Our parallel algorithm will determine whether a multidag is a TTSP graph. However, since the property of having exactly one source and exactly one sink can be easily checked on an EREW PRAM (details in section 5.6), we will concentrate on two terminal multidags.
Figure 13. A TTSP graph and its decomposition tree.

Figure 14. A TTSP graph is decomposed into two reducing components.
5.3 Reducing Components of Two Terminal Multidags

We outline in this section the basic ideas of our parallel decomposition algorithm. The basic strategy of the algorithm is "divide and conquer". For the time being, suppose \( G \) is a TTSP graph and \( H \) is a decomposition tree of \( G \). Each internal node \( x \) of \( H \) represents a subgraph \( G_x \) of \( G \) which itself is a TTSP graph. \( G_z \) is called a reducing component of \( G \). Let \( x_i \) \((1 \leq i \leq k)\) be \( k \) internal nodes of \( H \) such that for any \( i \neq j \), \( x_i \) and \( x_j \) are not related to each other in \( H \). (Namely, \( x_i \) is not an ancestor of \( x_j \) and \( x_j \) is not an ancestor of \( x_i \)). Let \( G_i \) be the TTSP graph represented by \( x_i \) with the source \( u_i \) and the sink \( v_i \) \((1 \leq i \leq k)\). If we delete all edges and vertices (except \( u_i \) and \( v_i \)) of \( G_i \) from \( G \) and add a new edge \( e(u_i,v_i) \) into \( G \) for each \( i \) \((1 \leq i \leq k)\), the resulting graph \( G_0 \) is also a TTSP graph. \( G_0 \) is called the reduction of \( G_i \) \((1 \leq i \leq k)\). The subtree \( H_i \) of \( H \) rooted at \( x_i \) is a decomposition tree of \( G_i \) \((1 \leq i \leq k)\). Let \( H_0 \) be the tree obtained from \( H \) by deleting the subtrees \( H_i \) (keeping the root \( x_i \)) and relabel \( x_i \) by the newly introduced edge \( e(u_i,v_i) \). Then \( H_0 \) is a decomposition tree of \( G_0 \). In Figure 14, the TTSP graph in Figure 13 is decomposed into two reducing components \( G_1, G_2 \) and their reduction \( G_0 \). The corresponding decomposition trees are also shown.

Our algorithm works as follows: Find a collection \( G_1, \cdots, G_k \) of reducing components of \( G \) and the reduction \( G_0 \) of them so that \( |G_i| \leq c |G| \) \((0 \leq i \leq k)\) for some constant \( c < 1 \). Then run the algorithm recursively on each \( G_i \) in parallel to construct a decomposition tree \( H_i \) for each \( G_i \) \((0 \leq i \leq k)\). Finally construct a decomposition tree \( H \) of \( G \) from \( H_i \) \((0 \leq i \leq k)\).
The problem is that we do not know whether $G$ is a TTSP graph. Even if $G$ is a TTSP graph, $H$ is not known (this is what we want to find). In order to make this strategy work, we have to characterize the reducing components of a two terminal multidag $G$ by using the properties of $G$, not the properties of $H$.

Throughout this chapter, $W$ exclusively denotes the digraph shown in Figure 15. The following forbidden subgraph characterization of TTSP graphs is well known ([D] and [VTL]) and is essential for our algorithm.

**Lemma 5.2:** A two terminal multidag $G$ is a TTSP graph iff $G$ does not contain subgraph homeomorphic to $W$. (A graph is homeomorphic to $W$ if it includes at least four distinct vertices $a,b,c,d$ and pairwise internal vertex disjoint paths $P_{ba}, P_{ca}, P_{bc}, P_{db},$ and $P_{dc}$.)

In order to facilitate the development of our parallel algorithm, we will prove a stronger version of Lemma 5.2.

**Definition 5.4:** A forbidden figure is a multidag includes at least four distinct vertices $a,b,c,d$ and the paths $P_{ha}, P_{ca}, P_{bc}, P_{db},$ and $P_{dc}$. These paths may intersect each other at some internal vertices except that:

1. For any $x,y,z \in \{a,b,c,d\}$, $z$ is not on $P_{yx}$; and
2. The last intersection vertex of $P_{bc}$ and $P_{ba}$ is not on $P_{dc}$.

**Lemma 5.3:** Let $G$ be a two terminal multidag. $G$ is a TTSP graph iff $G$ does not contain forbidden figure as a subgraph.

**Proof:** By Lemma 5.2, it is enough to show that a forbidden figure $U$ contains a subgraph homeomorphic to $W$. Since $U$ is acyclic by definition, Figure 16 (1) shows all possible intersection patterns of the paths of $U$. The following steps will find a subgraph of $U$ homeomorphic to $W$. 
Figure 15. The forbidden subgraph $W$.

Figure 16. The proof of Lemma 5.3.
Step 1: Let \( a' \) be the first intersection vertex of \( P_{ba} \) and \( P_{ca} \). Replace \( a \) by \( a' \). Discard the portion \( P_{a'a} \) from the paths \( P_{ba} \) and \( P_{ca} \). This gives a graph where \( P_{ba'} \) and \( P_{ca'} \) has no internal intersection vertex (Figure 16 (2)).

Step 2: Let \( b' \) be the last intersection vertex of \( P_{ba'} \) and \( P_{bc} \). Replace \( b \) by \( b' \). Replace \( P_{bc} \) by the portion \( P_{b'bc} \); \( P_{ba'} \) by the portion \( P_{b'a'} \). Replace the path \( P_{db} \) by the concatenation of \( P_{db} \) and the portion \( P_{bb'} \). This gives a graph where the paths \( P_{b'a'} \), \( P_{ca'} \) and \( P_{b'c} \) have no internal intersections (Figure 16 (3)).

Step 3: Let \( d' \) be the last intersection vertex of the paths \( P_{db'} \) and \( P_{dc} \). By the condition (2) of definition 5.4, \( d' \) and \( b' \) are distinct. Replace \( d \) by \( d' \). Replace the paths \( P_{db'} \) and \( P_{dc} \) by proper portions. We get a graph where only \( P_{d'c} \) may internally intersect with \( P_{b'a'} \) and \( P_{b'c} \) (Figure 16 (4)).

Step 4: If \( P_{d'c} \) does not intersect with \( P_{b'a'} \), skip this step. Otherwise, let \( b'' \) be the last intersection vertex of \( P_{d'c} \) and \( P_{b'a'} \). We get a graph with vertices \( b', b'', a', c \) and corresponding paths as shown in Figure 16 (5).

Step 5: At this point, we have a graph as shown in Figure 16 (6). (If step 4 is applied, the names of the vertices are different). Let \( c' \) be the first intersection vertex of \( P_{b'c} \) and \( P_{d'c} \). Replace \( c \) by \( c' \). Replace \( P_{d'c}, P_{b'c} \) and \( P_{ca'} \) by proper new paths. We get a graph homeomorphic to \( W \) (Figure 16 (7)). □

We next formally define the concepts of the reducing component and the reduction which are discussed intuitively at the beginning of this section.

**Definition 5.5:** Let \( G = (V,E) \) be a two terminal multidag. A reducing component of \( G \) with the head attachment \( u \) and the tail attachment \( v \) is a subgraph \( G_1 = (V_1,E_1) \) of \( G \) with at least two edges such that:
(1) \( G_1 \) is a two terminal multidag with \( u \) as the source and \( v \) as the sink.

(2) For any \((x,y)\in E\), \( y \in V_1 - \{u,v\} \) implies \( x \in V_1 \).

(3) For any \((x,y)\in E\), \( x \in V_1 - \{u,v\} \) implies \( y \in V_1 \).

Intuitively, no edges of \( G \) can enter or leave a vertex in \( V_1 - \{u,v\} \) from the "outside" of \( V_1 \).

Let \( G = (V,E) \) be a two terminal multidag and let \( G_i = (V_i,E_i) \) (1 \( \leq i \leq k \)) be \( k \) reducing components of \( G \) with attachments \( u_i \) and \( v_i \), respectively. \( G_i \) (1 \( \leq i \leq k \)) are pairwise disjoint if for \( i \neq j \), \( G_i \) and \( G_j \) have no common edges and at most have their attachments as common vertices.

**Definition 5.6:** Let \( G = (V,E) \) be a two terminal multidag and let \( G_i = (V_i,E_i) \) (1 \( \leq i \leq k \)) be \( k \) pairwise disjoint reducing components of \( G \). The reduction of \( G_i \) is the graph \( G_0 = (V_0,E_0) \) where \( V_0 = V - \bigcup_{1 \leq i \leq k} (V_i - \{u_i,v_i\}) \), \( E_0 = (E - \bigcup_{1 \leq i \leq k} E_i) \cup \{e(u_i,v_i) | 1 \leq i \leq k \} \) where \( e(u_i,v_i) \) are new edges.

The following lemma justifies the divide and conquer strategy.

**Lemma 5.4:** Let \( G = (V,E) \) be a two terminal multidag with source \( s \) and sink \( t \). Let \( G_i = (V_i,E_i) \) (1 \( \leq i \leq k \)) be \( k \) pairwise disjoint reducing components of \( G \) with attachments \( u_i \) and \( v_i \), respectively. Let \( G_0 \) be the reduction of \( G_i \). Then:

1. \( G_0 \) is a two terminal multidag with \( s \) as the source and \( t \) as the sink.
2. \( G \) is a TTSP graph iff \( G_0 \) and every \( G_i \) (1 \( \leq i \leq k \)) is a TTSP graph.

**Proof:** We only prove the case \( k = 1 \). For \( k > 1 \), the lemma is proved by induction.
(1) $G_0$ is clearly a multidag. We need to show that $s$ (t) is the only source (sink) of $G_0$. If $s \notin V_1$, $s \in V - (V_1 - \{u_1,v_1\}) = V_0$. If $s \in V_1$, $s$ must be the source $u_1$ of $G_1$. Therefore, $s \in V - (V_1 - \{u_1,v_1\}) = V_0$. Similarly, $t \in V_0$.

Next we show that $G_0$ has no other sources or sinks. $u_1$ is the only vertex in $G_0$ whose outdegree in $G_0$ is different from that in $G$. In $G_0$, $u_1$ is not a sink because of the existence of the edge $e(u_1,v_1)$. Thus no new sink is introduced into $G_0$ and $t$ is the only sink in $G_0$. Similarly $s$ is the only source in $G_0$.

(2) "If": Suppose both $G_0$ and $G_1$ are TTSP graphs. Toward a contradiction, assume $G$ is not. By lemma 5.2, $G$ contains a subgraph $\bar{G}$ homeomorphic to $W$. Since $G_0$ and $G_1$ are TTSP graphs, $\bar{G} \subseteq G_0$ and $\bar{G} \subseteq G_1$. So, $\bar{G}$ contains at least one edge in $E_1$. Since $G_1$ is a reducing component of $G$, edges of $G$ can enter or leave $V_1$ only through $u_1$ or $v_1$, respectively. Thus, $\bar{G}$ contains a path $P$ in $G_1$ from $u_1$ to $v_1$ (Figure 17). Replace $P$ by the edge $e(u_1,v_1)$ in $\bar{G}$. The resulting graph is a subgraph of $G_0$ homeomorphic to $W$. This contradicts to the assumption that $G_0$ is a TTSP graph.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure17.png}
\caption{The proof of Lemma 5.4.}
\end{figure}
"Only if": Suppose \( G \) is a TTSP graph. If \( G_1 \) is not a TTSP graph, it contains a subgraph \( G \) homeomorphic to \( W \). But any subgraph of \( G_1 \) is also a subgraph of \( G \). This contradicts the assumption that \( G \) is a TTSP graph.

If \( G_0 \) is not a TTSP graph, let \( G \) be a subgraph of \( G_0 \) homeomorphic to \( W \). If \( e(u_1,v_1) \notin G \), \( G \) is also a subgraph of \( G \), a contradiction. If \( e(u_1,v_1) \in G \), replace it by a path of \( G_1 \) from \( u_1 \) to \( v_1 \). The resulting graph is a subgraph of \( G \) and homeomorphic to \( W \). Again, a contradiction. \( \Box \)

5.4 Decomposition of Chain Type TTSP Graphs

In this section we discuss how to decompose some special class of TTSP graphs. The general case is considered in the next section.

**Definition 5.7:** A two terminal multidag (TTSP graph) \( G \) is a *chain type* two terminal multidag (*chain type* TTSP graph) if \( G \) contains a path containing all vertices of \( G \).

In this section, we assume \( G = (V,E) \) is a chain type two terminal multidag where \( V = \{1,2,\cdots,n\} \) and \( 1-2-\cdots-n \) is a path. \([i,j] \) denotes the set of vertices \( x \) such that \( i \leq x \leq j \).

In order to decompose \( G \) into reducing components of \( G \), we use a spanning tree \( T \) of \( G \) to guide the searching.

For each \( i \in V \) \((i \neq n)\), designate \( P(i) = \max\{j | (i,j) \in E\} \) as the parent of \( i \). The collection of these parent pointers \( \{(i,P(i)) | i = 1,2,\cdots,n-1\} \) clearly forms a spanning tree of \( G \) rooted at \( n \). It is called the *jump tree* of \( G \). Figure 18 shows a chain type TTSP graph and its jump tree. The solid lines represent tree edges. The dash lines represent nontree edges.
Figure 18. The jump tree of a chain type TTSP graph.

Figure 19. The proof of Lemma 5.5.
**Lemma 5.5**: Let $G$ be a chain type TTSP graph and let $T$ be the jump tree of $G$. For any $i \in V$, define $L(i) = \min \{ j \mid j \in T(i) \}$. Then

1. For any $i$, $V(T(i)) = [L(i), i]$.
2. For any $i > 1$, the vertex set $[(L(i) - 1), i]$ induces a reducing component of $G$ with $(L(i) - 1)$ as the head attachment and $i$ as the tail attachment. (If $L(i) = 1$, let $L(i) - 1 = 1$).
3. Let $i_1, \ldots, i_r$ be all children of $i$ in $T$ such that $i_1 > i_2 > \cdots > i_r$. Then $i_1 = i - 1$ and for any $k$ ($1 \leq k < r$), $L(i_k) - 1 = i_{k+1}$. Moreover, for any $1 \leq k \leq r$, the vertex set $[(L(i_k) - 1), i]$ induces a reducing component of $G$ with $(L(i_k) - 1)$ as the head attachment and $i$ as the tail attachment. (If $L(i_k) = 1$, let $L(i_k) - 1 = 1$).

**Proof**: (1) Suppose there is a $j$ such that $L(i) < j < i$ and $j \notin T(i)$. Let $k = NCA_T(i, j)$ be the nearest common ancestor of $i$ and $j$. The subgraph consisting of the vertices $L(i), j, i, k$ and the paths $L(i) \rightarrow j, j \rightarrow i, i \rightarrow k, L(i) \rightarrow i, j \rightarrow k$ is a forbidden figure. (The first three paths are the portions of the chain $1 \rightarrow \cdots \rightarrow n$. The last two paths are the paths in $T$. See Figure 19 (1)). This contradicts the assumption that $G$ is a TTSP graph. So, no such $j$ exists.

(2) The condition (1) of definition 5.5 is clearly true. We prove (2) and (3) of definition 5.5.

Suppose $(x, y) \in E$ is an edge such that $y \in [L(i), i - 1]$. Let $k = NCA_T((L(i) - 1), i)$. If $x < L(i) - 1$, the subgraph consisting of vertices $x, (L(i) - 1), y, k$ and the paths $x \rightarrow (L(i) - 1), (L(i) - 1) \rightarrow y, x \rightarrow y, y \rightarrow k, (L(i) - 1) \rightarrow k$ is a forbidden figure (Figure 19 (2)). A contradiction. Thus, $x \geq L(i) - 1$. 
Suppose \((x,y)\in E\) is an edge such that \(x\in [L(i),i-1]\). If \(y > i\), \(P(x) = \max\{j | (x,j)\in E\} \geq y > i\) and \(x\) can not be in \(T(i)\). So, we must have \(y \leq i\). (Figure 19 (3)).

(3) From (1) we have: \(V(T(i)) = [L(i),i], V(T(i_k)) = [L(i_k),i_k] \) \((1 \leq k \leq r)\). Since \(V(T(i)) = \bigcup_{1 \leq k \leq r} V(T(i_k)) \cup \{i\}\) and \(i_1 > \cdots > i_r\), these equations imply \(i_1 = i - 1\) and \(L(i_k) - 1 = i_{k+1}\). The proof that \([L(i_k) - 1, i]\) induces a reducing component is similar to the proof of (2). □

Lemma 4.3 in Chapter IV is also needed in this chapter. For completeness, we restate this lemma as follows.

**Lemma 5.6:** Let \(T\) be any rooted tree with \(n\) vertices. Then there exists a vertex \(i_0\) in \(T\) such that either

\[(1)\] \(n/3 \leq |T(i_0)| \leq 2n/3\); or

\[(2)\] \(|T(i_0)| > 2n/3\) and if \(i_1, \cdots, i_r\) are children of \(i_0\) in \(T\), then \(|T(i_k)| < n/3\) for \(1 \leq k \leq r\). □

**Remark:** From the proof of Lemma 4.3, it is easy to see that \(i_0\) can be chosen such that the path from \(i_0\) to the root of \(T\) is a heavy path of \(T\).

We are ready to prove the main theorem on the decomposition of chain type TTSP graphs.

**Theorem 5.7:** Let \(G\) be a chain type TTSP graph with \(n \geq 12\) vertices. Then \(G\) can be decomposed into reducing component \(G_1\) and the reduction \(G_0\) such that \(|G_i| \leq 5n/6\) \((i = 0,1)\).

**Proof:** Applying Lemma 5.6 to the jump tree \(T\) of \(G\), let \(i_0\) be the vertex satisfying Lemma 5.6.
If (1) of Lemma 5.6 is true, let \( G_1 \) be the reducing component induced by the vertex set \([L(i_0)-1,i_0]\) and let \( G_0 \) be the reduction of \( G_1 \). Then:

\[
|G_1| \leq |T(i_0)| + 1 \leq 2n/3 + 1 \leq 5n/6 \quad \text{and} \quad |G_0| \leq n - |T(i_0)| + 2 \leq n - n/3 + 2 \leq 5n/6.
\]

If (2) of Lemma 5.6 is true, let \( i_1, \ldots, i_r \) be the children of \( i_0 \) such that \( i_1 > \cdots > i_r \). Since \(|T(i_k)| < n/3\) for all \( 1 \leq k \leq r \), there is a \( t \) such that \( n/3 \leq \sum_{k=1}^{t} |T(i_k)| \leq 2n/3 \). By Lemma 5.5, the vertex set \([L(i_t)-1,i_0]\) induces a reducing component \( G_1 \). Let \( G_0 \) be the reduction of \( G_1 \). Then

\[
|G_1| \leq 2 + \sum_{k=1}^{t} |T(i_k)| \leq 2 + 2n/3 \leq 5n/6 \quad \text{and} \quad |G_0| \leq (n - \sum_{k=1}^{t} |T(i_k)|) + 1 \leq (n - n/3) + 1 \leq 5n/6. \quad \square
\]

5.5 Decomposition of General TTSP Graphs

In this section, we discuss the decomposition of general TTSP graphs.

Let \( G = (V,E) \) be a two terminal multidag with source \( s \) and sink \( t \). Let \( T \) be any spanning tree of \( G \) with \( t \) as the root. We will use \( T \) to guide the search for the reducing components of \( G \).

Definition 5.8: Let \( G = (V,E) \) be a TTSP graph and let \( T \) be a spanning tree of \( G \) rooted at the sink \( t \) of \( G \). For any \( x \in V \) define the crossing set of \( x \) with respect to \( T \) as:

\[
CROSS_T(x) = \{ z \in V - T(x) \mid \text{there is an edge } (z,y) \in E \text{ such that } y \in T(x) \}.
\]

Lemma 5.8: Let \( G = (V,E) \) be a TTSP graph and let \( T \) be a spanning tree of \( G \). Let \( x \) be a vertex of \( G \).
Figure 20. The proof of Lemma 5.8.
(1) If \( x \) is not on the main path \( MP(T) \) (see Section 5.2 for definition), then 
\[ CROSS_T(x) \] consists of exactly one vertex.

(2) If \( x \) is on \( MP(T) \), then \( CROSS_T(x) = \emptyset \).

**Proof:** (1) Let \( y \) be any vertex in \( T(x) \). By Lemma 5.1, there is a path \( P_{sy} \)
in \( G \) from the source \( s \) to \( y \). Since \( x \) is not on \( MP(T) \), \( s \) is not in \( T(x) \). Let
\((z,y')\) be the last edge on \( P_{sy} \) such that \( z \in T(x) \) and \( y' \in T(x) \). Then
\( z \in CROSS_T(x) \). So, \( CROSS_T(x) \) contains at least one vertex (Figure 20 (1)).

Suppose that \( CROSS_T(x) \) contains more than one vertex. Let \( z_1 \) and \( z_2 \) be
two distinct vertices in \( CROSS_T(x) \) and let \( e_1 = (z_1, y_1) \) and \( e_2 = (z_2, y_2) \) be two
edges in \( G \) such that \( y_i \in T(x) \), \( z_i \in T(x) \) (\( i = 1,2 \)). Let \( v = NCA_T(y_1, y_2) \) and let
\( w \) be the nearest common ancestor of \( z_1,z_2 \), and \( v \) in \( T \). By Lemma 5.1, there
are two paths \( P_1 \) and \( P_2 \) from the source \( s \) to \( z_1 \) and \( z_2 \), respectively. Let \( q \) be
the last common vertex of \( P_1 \) and \( P_2 \).

If \( q \neq z_1 \) and \( q \neq z_2 \) (Figure 20 (2)), the graph consisting of the vertices
\( w,z_1,v,q \) and the paths \( z_1 \rightarrow w \), \( v \rightarrow x \rightarrow w \), \( z_1 \rightarrow y_1 \rightarrow v \), \( q \rightarrow z_1 \), \( q \rightarrow z_2 \rightarrow y_2 \rightarrow v \) is a forbidden
figure.

If \( q = z_2 \) (the case \( q = z_1 \) is similar) (Figure 20 (3)), the subgraph consisting
of the vertices \( w,z_1,z_2,v \) and the paths \( z_1 \rightarrow w \), \( v \rightarrow x \rightarrow w \), \( z_1 \rightarrow y_1 \rightarrow v \), \( z_2 \rightarrow z_1 \), \( z_2 \rightarrow y_2 \rightarrow v \)
is a forbidden figure.

In any case, the assumption that \( CROSS_T(x) \) contains more than one vertex
leads to the existence of a forbidden figure in \( G \). This contradicts the fact
that \( G \) is a TTSP graph. So \( CROSS_T(x) \) consists of exactly one vertex.
Let \( x \in MP(T) \). Toward a contradiction, suppose \( \text{CROSS}_T(x) \neq \emptyset \). Let \( z \in \text{CROSS}_T(x) \) and suppose \((z,y) \in E\) such that \( y \in T(x)\), \( z \notin T(x) \). Let \( w = \text{NCA}_T(x,z) \). Clearly \( w \neq x \) (otherwise \( z \in T(x) \)) and \( w \neq z \) (otherwise \( G \) contains a cycle) (Figure 20 (4)). Let \( P_{sz} \) be a path in \( G \) from \( s \) to \( z \). The subgraph consisting of the vertices \( w,z,x,s \) and the paths \( z-w, z-x, x-y-x, s-x, s-z \) is a forbidden figure. This contradicts the fact that \( G \) is a TTSP graph. \( \square \)

Let \( G = (V,E) \) be a TTSP graph and let \( T \) be a spanning tree of \( G \). By Lemma 5.8, for any \( x \in V - MP(T) \), \( \text{CROSS}_T(x) \) contains exactly one vertex. This vertex is denoted by \( C_T(x) \). For any \((u,v) \in E\), \( u \notin T(x) \) and \( v \in T(x) \) implies \( u = C_T(x) \). Moreover, for any \( y \in T(x) \), there exists a path in \( G \) from \( C_T(x) \) to \( y \) since there exists a path in \( G \) from the source \( s \) to \( y \) and this path must pass through \( C_T(x) \).

One might think that \( T(x) \cup \{C_T(x)\} \) induces a reducing component of \( G \) with attachments \( C_T(x) \) and \( x \). Unfortunately, this is not true: There might exist some edges \((u,v) \in E\) such that \( u \in T(x) - \{x\} \) and \( v \notin T(x) \). However, we will show that \( T(x) \) is a "building block" of some reducing component of \( G \).

**Definition 5.9:** Let \( G = (V,E) \) be a TTSP graph and let \( T \) be a spanning tree of \( G \). Let \( x \) be a vertex of \( G \) and let \( y_1, \ldots, y_k \) be children of \( x \) in \( T \). The *resolution graph*, \( \text{RES}_T(x) \), of \( G \) at \( x \) with respect to \( T \) is the digraph consisting of the vertex set \( \{a_1, \ldots, a_k\} \) and the edge set \( \{(a_j,a_i) \mid C_T(y_j) \in T(y_i)\} \).

**Definition 5.10:** The *cluster* of \( G \) at \( x \) induced by \( y_i \) with respect to \( T \) is defined as: \( \text{CLUS}_T(x,y_i) = \bigcup_j \{T(y_j)\} \mid j = i \) or there is a path from \( a_j \) to \( a_i \) in \( \text{RES}_T(x) \). Define \( V_T(x,y_i) = \text{CLUS}_T(x,y_i) \cup \{x,C_T(y_i)\} \).
Figure 21 (2) shows the resolution graph $RES_T(x)$ of the TTSP graph $G$ shown in Figure 21 (1). The set $CLUS_T(x,y)$ is the union of the vertices in the subtrees $T(y_2), T(y_3), \text{ and } T(y_4)$.

Lemma 5.9: Let $G=(V,E)$ be a TTSP graph and let $T$ be a spanning tree of $G$. Suppose $x$ is a vertex of $G$ on the main path $MP(T)$ and $y_1, \ldots, y_k$ are children of $x$ in $T$ where $y_1$ is also on $MP(T)$.

1. The resolution graph $RES_T(x)$ is a tree rooted at $a_1$;
2. For each $i \neq 1$, the set $V_T(x,y_i)$ induces a reducing component of $G$ with $C_T(y_i)$ as the head attachment and $x$ as the tail attachment.

Proof: (1) For each $i \neq 1$, there is at most one edge in $RES_T(x)$ leaving $a_1$. For $a_1$, since $C_T(y_1)$ does not exist, there is no edge in $RES_T(x)$ leaving $a_1$. Since $G$ is acyclic, it is easy to see that $RES_T(x)$ is acyclic also. Hence, $RES_T(x)$ is a directed forest with $a_1$ as one of its roots.
Suppose there exists a $j \neq 1$ such that $a_j$ is also a root in $RES_T(x)$. Then $C_T(y_j) \notin T(y_i)$ for any $1 \leq i \leq k$. Let $w = NCA_T(x, C_T(y_j))$ (Figure 22 (1)). Clearly $w \neq x$ (otherwise $C_T(y_j) \in T(y_i)$ for some $i$) and $w \neq C_T(y_j)$. Let $P$ be a path in $G$ from the source $s$ to $C_T(y_j)$. The subgraph of $G$ consisting of the vertices $s, C_T(y_j), x, w$ and the paths $s-y_1-x, s-C_T(y_j), x-w, C_T(y_j)-w, C_T(y_j)-y_j-x$ is a forbidden figure. This contradicts the assumption that $G$ is a TTSP graph. Thus, $a_1$ is the only root of $RES_T(x)$ and $RES_T(x)$ is a tree.

(2) The condition (1) of definition 5.5 can be easily checked. We prove conditions (2) and (3) of definition 5.5.

To prove the condition (2) of definition 5.5, suppose $(u, v) \in E$ is an edge such that $v \in CLUS_T(x, y_i)$. We need to show that $u \in V_T(x, y_i)$.

Case 1: $v \in T(y_i)$. By Lemma 5.8, either $u \in T(y_i) \subseteq V_T(x, y_i)$; or $u = C_T(y_i) \in V_T(x, y_i)$.

Case 2: $v \notin T(y_i)$. Then $v \notin T(y_j)$ for some $j$ and there is a path $P$ in $RES_T(x)$ from $a_j$ to $a_i$. Let $(a_j, a_p)$ be the first edge on $P$. Thus $C_T(y_j) \in T(y_j)$. Since $v \notin T(y_j)$, either $u \in T(y_j)$ or $u = C_T(y_j) \in T(y_p) \subseteq CLUS_T(x, y_i)$. In any case, $u \in V_T(x, y_i)$ (Figure 22 (2)).

To prove the condition (3) of definition 5.2, suppose $(u, v) \in E$ is an edge such that $u \in CLUS_T(x, y_i)$. We need to show $v \in V_T(x, y_i)$.

Suppose $u \in T(y_j)$ for some $j$ and there is a path $P$ in $RES_T(x)$ from $a_j$ to $a_i$. Since $a_j$ is not a root in $RES_T(x)$, there exists an $a_r$ such that $(a_i, a_r)$ is an edge in $RES_T(x)$. Thus, $C_T(y_j)$ exists and $C_T(y_j) \in T(y_r)$. Let $w = NCA_T(x, v)$ be the nearest common ancestor of $x$ and $v$ in $T$. 
Figure 22. The proof of Lemma 5.9.
If \( w = x \), then \( v \in T(x) \). So, \( v \in T(y_q) \) for some \( q \) where \( y_q \) is a child of \( x \) in \( T \) (Figure 22 (3)). Since \((u,v) \in E \) and \( v \in T(y_q) \), \( C_T(y_q) = u \in T(y_j) \). Thus \((a_q, a_j)\) is an edge in \( RES_T(x) \). The combination of this edge and the path from \( a_j \) to \( a_i \) is a path in \( RES_T(x) \) from \( a_q \) to \( a_i \). Hence, \( v \in T(y_q) \subseteq CLUS_T(x,y_i) \subseteq V_T(x,y_i) \).

If \( w \neq x \), consider the subgraph of \( G \) consisting of the vertices \( w,u,x \), \( C_T(y_i) \) and the paths \( u-v-w \), \( x-w \), \( C_T(y_i)-C_T(y_j)-u \), \( C_T(y_i)-y_-x \), \( u-y_j-x \). This subgraph is a forbidden figure (Figure 22 (4)). This contradicts the assumption that \( G \) is a TTSP graph. So, this case cannot occur. (Notice that if the assumption that \( a_i \) is not a root of \( REST(x) \) is dropped, the argument fails). □

Remark: We may compare Lemma 5.9 with Lemma 5.5 as follows. Let \( G = (V,T) \) be a chain type TTSP graph where \( V = \{1, \ldots, n\} \) and \( 1-2, \ldots, -n \) is a path of \( G \). Let \( T \) be the jump tree of \( G \) and let \( L(i) = \min\{j \in T(i)\} \).

Then \( C_T(i) = L(i) - 1 \). Let \( i_1, \ldots, i_r \) be the children of \( i \) in \( T \) such that \( i_1 > i_2 > \cdots > i_r \). By Lemma 5.5, \( C_T(i_k) = L(i_k) - 1 = i_{k+1} \in T(i_{k+1}) \) for \( 1 \leq k < r \).

The resolution graph \( REST_T(i) \) consists of the vertices \( a_1, \ldots, a_r \) and the chain \( a_1-a_2-\cdots-a_r \).

So, \( CLUS_T(i,i_k) = \bigcup_{1 \leq j \leq k} T(i_j) = \bigcup_{1 \leq j \leq k} [L(i_j),i_j] = [L(i_k),i] \) and \( V_T(i,i_k) = [L(i_k),i] \cup \{C_T(i_k),i\} = [L(i_k)-1,i] \).

Hence, conclusion (3) of Lemma 5.5 is just a special case of Lemma 5.9. The conclusion (2) of Lemma 5.5 holds for chain type TTSP graphs. For general TTSP graphs, it is not true. (Namely the vertex set \( V_T(x,y_i) \) does not necessarily induce a reducing component).

We need one more technical lemma.
Lemma 5.10: Let $T$ be a rooted tree with vertex set $\{a_1, \ldots, a_k\}$ and the root $a_1$. Suppose each vertex $a_i$ has a positive weight $w(a_i)$. Let $TW(a_i) = \sum_{a_j \in T(a_i)} w(a_j)$ be the total weight of the vertices in $T(a_i)$. Let $n$ be a fixed number. If $w(a_i) \leq n/3$ for all $1 \leq i \leq k$ and $TW(a_1) \geq 2n/3$, then there exists an $a_{i_0} \in \{a_1, \ldots, a_k\}$ such that either:

1. \( n/3 \leq TW(a_{i_0}) \leq n/2 \) and $a_{i_0}$ is not the root of $T$; or

2. If $a_{i_1}, \ldots, a_{i_r}$ are children of $a_{i_0}$ in $T$, then
   
   a. $TW(a_{i_0}) > n/2$; and
   
   b. $TW(a_{i_j}) < n/3$ ($1 \leq j \leq r$); and
   
   c. $\sum_{j=1}^{r} TW(a_{i_j}) > n/6$.

Proof: First observe that if $a_{i_0}$ satisfies (1), $a_{i_0}$ can not be the root $a_1$ since $TW(a_1) \geq 2n/3$ by assumption. The following procedure will find $a_{i_0}$:

(i) $a_{i_0} = a_1$;

(ii) Let $a_{i_1}, \ldots, a_{i_r}$ be children of $a_{i_0}$ in $T$.

If there is an $a_{i_l}$ such that $TW(a_{i_l}) > n/2$ then $a_{i_0} = a_{i_l}$; goto (ii);

Else (* $TW(a_{i_l}) \leq n/2$ for all $1 \leq l \leq r$ *)

If there is an $a_{i_l}$ such that $TW(a_{i_l}) \geq n/3$ then $a_{i_l}$ satisfies (1)

Else we have $TW(a_{i_l}) > n/2$, $TW(a_{i_l}) < n/3$ ($1 \leq l \leq r$) and $\sum_{l=1}^{r} TW(a_{i_l}) = TW(a_{i_0}) - w(a_{i_0}) > n/2 - n/3 = n/6$

so $a_{i_0}$ satisfies (2). □
Although Lemma 5.9 is valid for any spanning tree \( T \) of a TTSP graph \( G \), in order to decompose \( G \) into reducing components of small size, we have to use a special spanning tree \( T \) of \( G \) defined as follows.

**Definition 5.11:** Let \( G \) be a two terminal multidag. A spanning tree \( T \) of \( G \) is a *normal spanning tree* if the main path \( MP(T) \) is a heavy path.

We are now ready to prove our main theorem on the decomposition of general TTSP graphs.

**Theorem 5.11:** Let \( G = (V,E) \) be a TTSP graph with \( n \geq 6 \) vertices and let \( T \) be a normal spanning tree of \( G \). Then \( G \) can be decomposed into reducing components \( G_j \) \((1 \leq j \leq q)\) for some \( q \) and the reduction \( G_0 \) such that either:

1. \( |G_j| \leq 5n/6 \) for \( 1 \leq j \leq q \) and \( G_0 \) consists of exactly the vertices on the main path \( MP(T) \); or

2. \( |G_j| \leq 5n/6 \) for \( 0 \leq j \leq q \).

**Proof:** Applying Lemma 5.6 to \( T \), we can find a vertex \( x_0 \) satisfying either (1) or (2) of Lemma 5.6. Since \( T \) is normal, \( x_0 \) can be chosen from the vertices on the main path of \( T \).

Case 1: (1) of Lemma 5.6 is true: \( n/3 \leq |T(x_0)| \leq 2n/3 \). Consider any vertex \( x \) on \( MP(T) \). Let \( y_1, \ldots, y_k \) be the children of \( x \) in \( T \) and suppose \( y_1 \) is on \( MP(T) \). In the resolution graph \( RES_T(x) \), \( a_1 \) is the root. Suppose \( a_{i_1}, \ldots, a_{i_r} \) are children of \( a_1 \) in \( RES_T(x) \). By Lemma 5.9, each \( V_T(x,y_{i_j}) \) \((1 \leq j \leq r)\) induces a reducing component \( G_j \) of \( G \).

If \( x \in T(x_0) \), then \( V_T(x,y_{i_j}) \subseteq T(x_0) \). Hence, \( |G_j| = |V_T(x,y_{i_j})| \leq |T(x_0)| \leq 2n/3 < 5n/6 \).
If \( x \notin T(x_0) \), then \( CLUST_T(x, y_j) \cup \{x\} \subseteq V - T(x_0) \). Hence
\[
|G_j| = |CLUST_T(x, y_j) \cup \{x\}| + 1 \leq (|V| - |T(x_0)|) + 1 \leq n - n/3 + 1 \leq 5n/6.
\]

Perform this decomposition for all \( x \in MP(T) \). Let \( G_1, \cdots, G_q \) be all reducing components obtained this way. Let \( G_0 \) be the reduction of \( G_j \) \((1 \leq j \leq q)\). \( G_0 \) consists of exactly the vertices on \( MP(T) \). So, this decomposition satisfies (1).

Case 2: (2) of Lemma 5.6 is true: \( |T(x_0)| > 2n/3 \) and for all children \( y_1, \cdots, y_k \) of \( x_0 \) \( |T(y_j)| < n/3 \). Suppose \( y_1 \) is on the main path. Consider the resolution graph \( RES_T(x_0) \). The vertex set of \( RES_T(x_0) \) is \( \{a_1, \cdots, a_k\} \) with \( a_1 \) as the root. Give each \( a_i \) a weight \( w(a_i) = |T(y_i)| \). Then \( w(a_i) = |T(y_i)| < n/3 \) for \( 1 \leq i \leq k \) and \( TW(a_i) = \sum_{i=1}^{k} |T(y_i)| = |T(x_0)| - 1 > 2n/3 - 1 \). So, \( TW(a_1) \geq 2n/3 \). Apply Lemma 5.10 to the tree \( RES_T(x_0) \), one of the following two cases holds:

Case 2.1: There is a non-root \( a_{i_0} \) such that \( n/3 > TW(a_{i_0}) \geq n/2 \). By Lemma 5.9, \( V_T(x_0, y_{i_0}) \) induces a reducing component \( G_1 \). Let \( G_0 \) be the reduction of \( G_1 \). Then
\[
|G_1| = |CLUST_T(x_0, y_{i_0})| + 2 = TW(a_{i_0}) + 2 \leq n/2 + 2 \leq 5n/6
\]
and
\[
|G_0| = |G| - |CLUST_T(x_0, y_{i_0})| = n - TW(a_{i_0}) \leq n - n/3 < 5n/6.
\]

Case 2.2: There is an \( a_{i_0} \) such that if \( a_{i_1}, \cdots, a_{i_r} \) are children of \( a_{i_0} \) in \( RES_T(x_0) \), then \( \sum_{j=1}^{r} TW(a_{i_j}) > n/6 \) and \( TW(a_{i_j}) < n/3 \) for \( 1 \leq j \leq r \). By Lemma 5.9, each \( V_T(x_0, y_{i_j}) \) \((1 \leq j \leq r)\) induces a reducing component \( G_j \) of \( G \). Let \( G_0 \) be the reduction of them. Then for each \( 1 \leq j \leq r \):
\[
|G_j| = |CLUST_T(x_0, y_{i_j})| + 2 = TW(a_{i_j}) + 2 < n/3 + 2 \leq 5n/6; \text{ and}
\]
In either case the decomposition satisfies (2). □

Remark: The different reducing components of $G$ constructed in Theorem 5.11 may have common attachments $u, v$. We can arbitrarily assign the edges $(u, v) \in E$ (if any) to any reducing component so that the reducing components obtained are pairwise disjoint.

5.6 Implementation

In this section, we sketch the implementation and analyze the resource bounds of our parallel recognition and decomposition algorithm.

The algorithm is as follows:

Algorithm 5.1 DECOM:

Input: A multidag $G = (V, E)$ given by edge list form.

(1) Test if $G$ is a two terminal multidag. If not, $G$ is not a TTSP graph, stop;

(2) If $|G| < 12$, test and decompose $G$ directly (discussed later), stop;

(3) Construct a normal spanning tree $T$ of $G$;

(4.1) If $T$ is a chain, $G$ is a chain type two terminal multidag. Try to decompose $G$ into $G_0$ and $G_1$ as in Theorem 5.7;

(4.2) If $T$ is not a chain, try to decompose $G$ into $G_0, \cdots, G_q$ as in Theorem 5.11;

(5) If either step (4.1) or (4.2) fails, $G$ is not a TTSP graph, stop;

(6) Recursively call DECOM on $G_0, \cdots, G_q$ in parallel;
(7) If one of \( G_i \) \((0 \leq i \leq q)\) is not a TTSP graph, \( G \) is not either, stop;

(8) If every \( G_i \) \((0 \leq i \leq q)\) is a TTSP graph, so is \( G \). The decomposition tree \( H_i \)
for each \( G_i \) has been constructed in step (6). Construct a decomposition tree \( H \) of \( G \) from \( H_i \) \((0 \leq i \leq q)\).

End DECOM.

The correctness of the algorithm is proved by induction on the size of \( G \).

If \( G \) is not a two terminal multidag, it is not a TTSP graph. The DECOM algorithm rejects \( G \) at step (1). So, we assume \( G \) is a two terminal multidag. If \( |G| < 12 \), the algorithm is correct at step (2). Suppose \( |G| \geq 12 \). If \( G \) is a TTSP graph, by Theorem 5.7 or Theorem 5.11, \( G \) will be successfully decomposed into smaller TTSP graphs \( G_i \) \((0 \leq i \leq q)\) at step (4). By the induction hypothesis, \( G_i \)
\((0 \leq i \leq q)\) will be correctly recognized and decomposed at step (6). So, the algo­rithm will recognize \( G \) as a TTSP graph and construct a decomposition tree for \( G \) at step (8).

If \( G \) is not a TTSP graph, on the other hand, then either step (4) (4.1 or 4.2) fails, or at least one of \( G_i \) \((0 \leq i \leq q)\) found in step (4) is not a TTSP graph. In the first case, the algorithm rejects \( G \) at step (5). In the second case, one of \( G_i \) \((0 \leq j \leq r)\) will be rejected at step (6) and the algorithm will reject \( G \) at step (7). Hence, DECOM is correct.

We next sketch some implementation details.

The input to the algorithm is a multidag \( G = (V, E) \) given by edge list form where \( |V| = n \) and \( |E| = m \). The algorithm uses \( O(n + m) \) processors. Each vertex \( i \in V \) is assigned a processor \( P_i \). Each edge \( e \in E \) is assigned a processor \( P_e \).
In order to simplify the analysis of the DECOM algorithm, we assume the algorithm performs the following operation as initialization.

In $O(\log m)$ time, the processors $P_e \ (e \in E)$ count the number of edges between each pair of vertices and replace them by a single edge (see Appendix B). The resulting graph $G_0$ is a simple dag (no multiple edges). Clearly, $G$ is a TTSP graph iff $G_0$ is a TTSP graph. Moreover, the decomposition tree of $G$ can be easily constructed from the decomposition tree of $G_0$. The conversion from $G$ to $G_0$ takes $O(\log m)$ time with $O(m)$ processors. This step contributes the $O(\log m)$ factor in the time bound of our algorithm. Thus without loss of generality, we assume that $G$ is a simple dag. In particular, $G$ contains $m \leq O(n^2)$ edges. Hence, $O(\log m) = O(\log n)$.

Step (1) of the DECOM algorithm can be implemented as follows: the processors $P_e \ (e \in E)$ sort the edge set of $G$ according to the second vertex of each edge. This can be done by the algorithm in [Col] in $O(\log m) = O(\log n)$ time with $O(m)$ processors. Then the number of children of each vertex can be computed in $O(\log m) = O(\log n)$ time with $O(m)$ processors. The vertices with no children are labeled as "sources". The processors $P_i \ (i \in V)$ compute the number of sources of $G$ in $O(\log n)$ time with $O(n)$ processors. The number of sinks of $G$ is computed similarly. Thus, step (1) takes $O(\log n)$ time with $O(n + m)$ processors.

For step (2), since the number of vertices of $G$ is at most 11, we can apply the linear time sequential algorithm of Valdes, Tarjan and Lawer [VTL] to test and decompose $G$. Thus, this step can be implemented in constant time with only one processor.
Step (3) needs more discussion. In order to construct a normal spanning tree of $G$, we first construct an arbitrary spanning tree $T_0$ of $G - \{s\}$ ($s$ is the source of $G$), and then modify $T_0$ to be a normal spanning tree.

For each $i \in V$ ($i \neq s$ and $i \neq t$), we select an arbitrary vertex $P(i)$ such that $(i, P(i)) \in E$. This selection can be done in $O(\log m) = O(\log n)$ time with $O(m)$ processors. The pointers $P(i)$ represent a spanning tree $T_0$ of $G - \{s\}$. By using the algorithm of Tarjan and Vishkin [TV1], $|T_0(i)| = |i \in V - s|$ can be computed in $O(\log n)$ time with $O(n)$ processors. Then the heavy child of each vertex in $T_0$ can be determined. Let $x_0$ be a leaf of $T_0$ such that the path in $T_0$ from $x_0$ to the root $t$ is a heavy path. Since $x_0$ is not the source $s$, we can find an edge $(y_1, x_0) \in E$ for some $y_1$. If $y_1 = s$, we are done. If $y_1 \neq s$, we start from $y_1$ and travel down $T_0$ along a heavy path until reaching another leaf $x_1$ of $T_0$. Repeat this procedure on $x_1$. Since $G$ is acyclic, we eventually get a path $P_1 = s - x_k - y_k - x_{k-1} - y_{k-1} - \cdots - x_1 - y_1 - x_0$ where $x_j$ ($0 \leq j \leq k$) is a leaf of $T_0$, $y_j - x_{j-1}$ ($1 \leq j \leq k$) is not an edge in $T_0$ and $x_j - y_j$ ($1 \leq j \leq k$) is a heavy path of $T_0$ (Figure 23 shows an example with $k = 2$). We modify $T_0$ as follows: Let $P(s) = x_k$ and $P(y_j) = x_{j-1}$ for $1 \leq j \leq k$. Other pointers $P(i)$ are not changed. These new pointers represent a spanning tree $T$ of $G$.

**Lemma 5.12:** The tree $T$ constructed above is a normal spanning tree of $G$.

**Proof:** Let $P = P_1 P_0$ be the main path of $T$ where $P_0$ is the path in the original tree $T_0$ from $x_0$ to $t$ and $P_1$ is the path from $s$ to $x_0$ constructed as above. Let $x$ and $y$ be any two consecutive vertices on $P$ where $x$ is the parent of $y$. We must show that $y$ is a heavy child of $x$. 
Case 1: \( x \in P_0 \) and \( x \neq x_0 \). For any child \( z \neq y \) of \( x \), clearly \( |T(z)| \leq |T_0(z)| \) and \( |T(y)| \geq |T_0(y)| \). Since \( P_0 \) is a heavy path in \( T_0 \), \( |T_0(y)| \geq |T_0(z)| \). Thus, \( |T(z)| \leq |T(y)| \).

Figure 23. Constructing a normal spanning tree.

Case 2: \( x = x_i \) and \( y = y_{i+1} \) for some \( i \). Since \( x \) has only one child \( y \) in \( T \), \( y \) is the heavy child of \( x \).

Case 3: \( x \) is a vertex on the path \( x_i - y_i \) for some \( i \) and \( x \neq x_i \). Using similar arguments as in case 1, we can show that \( y \) is a heavy child of \( x \). \( \square \)

In order to implement this procedure in parallel, we define:

\[ E_1 = \{(i, P(i)) \mid i \text{ is the heavy child of } P(i) \text{ in } T_0\}; \text{ and} \]

For each leaf \( i \) in \( T_0 \), we select an arbitrary edge \((j, i)\). Let \( E_2 \) be the set of edges selected.

Let \( E = E_1 \cup E_2 \) and let \( T \) be the subgraph induced by \( E \). Since \( E_1 \) is a collection of vertex disjoint paths in \( G \), the indegree of each vertex in \( T \) is 1 (the
indegree of the source $s$ is 0). Hence, $T$ represents a "downward" spanning tree of $G$ where the source $s$ is the "root" and every edge is "toward" $s$ in $T$. Both $E_1$ and $E_2$ can be identified in $O(\log n)$ time with $O(n + m)$ processors. Thus, $T$ can be identified within the same time and processor bounds. As described above, $T$ contains the path $P_1$ needed to construct the normal spanning tree $T$ of $G$ (Lemma 5.12). By using the well known "pointer jump" technique ([TV1] for example) on the tree $T$, $P_1$ can be identified in $O(\log n)$ time. After $P_1$ is known, the conversion from $T_0$ to the normal spanning tree $T$ can be done in constant time. So step (3) takes $O(\log n)$ time with $O(n + m)$ processors.

Step (4) is the central part of the algorithm. In order to decide which of step (4.1) or step (4.2) should be executed, the algorithm tests whether $T$ is a chain by checking if any vertex has indegree greater than 1 in $T$. This can be done in $O(\log n)$ time.

If $T$ is a chain, step (4.1) is executed. We construct the jump tree of $G$ as follows. At this point, we can assume that $G$ is a chain type two terminal multi-dag with vertex set $\{1, \ldots, n\}$ and a path $1 \rightarrow 2 \rightarrow \cdots \rightarrow n$. To construct the jump tree of $G$, it is enough to compute $P(i) = \max\{j \mid (i, j) \in E\}$ for each vertex $i$. This can be done in $O(\log n)$ time by the processors $P_e (e \in E)$.

By the remark after Lemma 5.9, the decomposition of $G$ by using Theorem 5.7 is just a special case of the decomposition by using Theorem 5.11. So, we only discuss step (4.2) in more detail.

First, $\{|T(i)|\}_{i \in V}$ are computed in $O(\log n)$ time with $O(n)$ processors [TV1]. Then a vertex $x_0$ satisfying Lemma 5.6 is selected. This selection can be done in $O(\log n)$ time.
Suppose (2) of Lemma 5.6 is true. The children \( y_1, \ldots, y_k \) of \( x_0 \) in \( T \) are identified. In \( O(\log n) \) time with \( O(n) \) processors, we can build a data structure for each \( y_i \) such that the predicate "\( z \in T(y_i) \)" can be checked by a single processor in constant time. Then the processors \( P_e \) where \( e = (z, y) \) such that \( z \notin T(y_i) \) and \( y \in T(y_i) \) will record "\( z \in \text{CROSS}_T(y_i) \)". If \( |\text{CROSS}_T(y_i)| \leq 2 \) for any \( y_i \), \( G \) is not a TTSP graph and the algorithm terminates. Otherwise the unique vertex \( C_T(y_i) \) in \( \text{CROSS}_T(y_i) \) (1 \( \leq i \leq k \)) is known and the resolution graph \( \text{RES}_T(x_0) \) can be constructed in constant time.

Using similar techniques, we can find a vertex \( a_i \) in \( \text{RES}_T(x_0) \) satisfying Lemma 5.10. A reducing component of \( G \) is induced by a vertex set \( V_T(x_0, y_i) = \bigcup_j \{ T(y_j) \} \mid j = i \) or there is a path from \( a_j \) to \( a_i \) in \( \text{RES}_T(x_0) \) \( \bigcup \{ x_0, C_T(y_i) \} \). The vertices \( a_j \) such that there is a path from \( a_j \) to \( a_i \) in \( \text{RES}_T(x_0) \) can be identified in \( O(\log n) \) time. Since the vertices in \( T(y_j) \) can be identified in constant time, the set \( V_T(x_0, y_i) \) can be determined in constant time also. Then the processors \( P_e \) check if \( V_T(x_0, y_i) \) is indeed a reducing component. If not, \( G \) is not a TTSP graph. Otherwise, \( V_T(x_0, y_i) \) induces a reducing component \( G_i \). The construction of the reduction \( G_0 \) can be done in constant time.

If \( x_0 \) satisfies (1) of Lemma 5.6, the operations described above are performed on all vertices \( x \in MP(T) \) in parallel (as in Theorem 5.11). This will give the required decomposition. Thus, the total running time for step (4.2) (or step 4.1) is \( O(\log n) \) with \( O(n + m) \) processors.

Step (5) and step (7) clearly takes constant time. Step (6) is the recursive call.
For step (8), the algorithm records which edge of $G_0$ is introduced for each $G_i$. Then the decomposition tree $H$ of $G$ can be constructed from $H_i$ ($0 \leq i \leq q$) in constant time.

We next analyze the time and the processor bounds of the DECOM algorithm.

If $G$ has $n < 12$ vertices, step (2) takes constant time. Suppose $n \geq 12$. By Theorem 5.11, one application of DECOM on $G$ will decompose $G$ into $G_0, \ldots, G_q$ so that either $|G_i| \leq 5n/6$ for $0 \leq i \leq q$; or $|G_i| \leq 5n/6$ for $1 \leq i \leq q$ and $G_0$ is a chain type two terminal multidag. In the second case, the second application of DECOM will decompose $G_0$ into components of size at most $\frac{5}{6}|G_0| \leq \frac{5n}{6}$ (Theorem 5.7). Thus, the depth of the recursive calls is at most $2 \log_{5/6} n$. Since all steps of DECOM can be executed in $O(\log n)$ time, the total running time is $O(\log^2 n)$. Taking the initialization step into account (converting a multidag to a simple dag), the DECOM algorithm can be executed in $O(\log^2 n + \log m)$ time.

On a graph $G$ with $n$ vertices and $m$ edges, the algorithm needs $n + m$ processors. After $G$ is decomposed into $G_0, G_1, \ldots, G_q$, one new edge and two new vertices are introduced for each $G_i$ ($1 \leq i \leq q$). So, the total number of vertices (edges) in $\{G_i\}_{0 \leq i \leq q}$ is $n + 2q (m + q)$. Since $G$ can be decomposed into at most $n$ components, the total number of processors needed for the DECOM algorithm is at most $(n + 2n) + (m + n) = O(m + n)$.

In summary we have:
Theorem 5.13: The DECOM algorithm correctly recognizes and decomposes TTSP graphs in $O(\log^2 n + \log m)$ time with $O(n + m)$ processors on an EREW PRAM. □
CHAPTER VI
PARALLEL ALGORITHMS FOR SOLVING
TTSP GRAPH PROBLEMS

In this chapter, we apply the binary tree algebraic computation methodology to design efficient parallel algorithms for solving various problems on two terminal series parallel graphs. If the input TTSP graph $G$ is given by the decomposition tree, these algorithms run in $O(m \log m)$ time with $O(m)$ processors on an EREW PRAM where $m$ is the number of edges in the input TTSP graph. If the input TTSP graph $G$ is given by edge list, we first run the parallel recognition and decomposition algorithm (see Chapter V) on $G$ to construct a decomposition tree for $G$ and then apply the algorithms presented in this chapter. In this case the total running time will be $O(n \log^2 n + m \log m)$ with $O(n + m)$ processors on an EREW PRAM (since this is the resource bounds of the recognition and decomposition algorithm). Throughout this chapter, we assume the input is given by the decomposition tree form.

In Section 6.1, we define a generalized version of the BTAC problem and present a parallel algorithm for solving this problem. In Section 6.2, we apply the generalized BTAC algorithm to solve the weighted maximum matching set problem on TTSP graphs. The methodology presented in Section 6.2 can be used to design efficient parallel algorithms for many other combinatorial
problems on TTSP graphs. In Section 6.3 we utilize the generalized BTAC algorithm, in a different way, to solve two other problems, the maximum cycle and the maximum path problems, on TTSP graphs.

6.1 Generalized BTAC Problem

In this section we define a generalized version of the BTAC problem on a finite set $S$ and present a parallel algorithm for solving this problem. Just as the BTAC algorithm provided a powerful tool for solving tree problems, the generalized BTAC algorithm will be extensively used for solving various TTSP graph problems.

Let $S$ be a set with $k$ elements. In order to avoid double subscripts, we assume $S = \{1, \cdots, k\}$ in this section. Let $R^+$ denote the set of real numbers plus a symbol $-\infty$. We define $-\infty + x = -\infty$ and $-\infty < x$ for any real number $x$. We denote by $\Pi = \{(a_1, \cdots, a_k) | a_i \in R^+ \text{ for } 1 \leq i \leq k\}$ the set of all $k$-element vectors over $R^+$.

Let $\Psi_S = \{f : S \times S \rightarrow S\}$ be the set of binary functions on $S$. For any $f \in \Psi_S$, $f$ induces a binary function $f : \Pi \times \Pi \rightarrow \Pi$ as follows: On two given vectors $A = (a_1, \cdots, a_k)$ and $B = (b_1, \cdots, b_k)$, define $f(A, B) = (c_1, \cdots, c_k)$ where $c_{ij} = \max_{i,j} \{a_i + b_j \mid f(i, j) = l\}$. (If there is no $i, j$ such that $f(i, j) = l$, define $c_{ij} = -\infty$). Denote by $\Psi_S = \{f \mid f \in \Psi_S\}$ the set of these induced binary functions.

A generalized binary computation tree is a binary tree $T$ such that each leaf $v$ of $T$ is labeled by a vector $L(v) \in \Pi$ and each internal vertex $u$ is labeled by an arbitrary function $F(u) = f \in \Psi_S$. The generalized BTAC problem on $S$ is to compute a final label $L(u) \in \Pi$ for all vertices of $T$ in the obvious way.
The generalized BTAC problem can be solved in parallel efficiently.

**Theorem 6.1:** For any set $S$ with $k$ elements, the generalized BTAC problem on $S$ can be solved in $O(k^3 \log n)$ time with $O(n)$ processors on an EREW PRAM.

**Proof:** We modify the BTAC algorithm (Algorithm 2.1) to solve the generalized BTAC problem.

Algorithm 2.1 consists of two loops. Each loop is executed at most $O(\log n)$ iterations. We will show that, when Algorithm 2.1 is performed on a generalized binary computation tree $T$, both loop bodies can be implemented in at most $O(k^3)$ time with $O(n)$ processors on an EREW PRAM. Then, our theorem follows.

We only discuss the implementation of the first loop. The implementation for the second loop is similar. For each vertex $v$ of $T$, we assign one processor $P_v$ to $v$.

First, consider the RAKE operation. Let $u$ be an internal vertex of $T$ with a function label $F(u) = f$ for some $f \in \Psi_S$. Let $v_1$ and $v_2$ be two children of $u$ in $T$. Suppose that both $v_1$ and $v_2$ are leaves with final labels $L(v_1) = (a_1, \ldots, a_k)$ and $L(v_2) = (b_1, \ldots, b_k)$, respectively. When the RAKE operation is performed, $v_1$ and $v_2$ are deleted and the final label $L(u) = F(u)(L(v_1), L(v_2))$ is computed immediately. The final label $L(u) = (c_1, \ldots, c_k)$ is evaluated by the processor $P_u$ as follows: Each variable $c_l$ ($1 \leq l \leq k$) is initialized to $-\infty$. Then, $P_u$ looks up the multiplication table of $f$. Suppose $f(i, j) = l$ is an entry in the multiplication table. $P_u$ reads the elements $a_i$ and $b_j$ from the vectors $L(v_1)$ and $L(v_2)$, computes $a_i + b_j$ and then updates the variable $c_l$. Since there are $k^2$ entries in
the multiplication table, the whole operation can be performed in \( O(k^2) \) time by the processor \( P_u \).

Suppose that \( u \) is an internal vertex of \( T \) and the left child \( v_1 \) of \( u \) is a leaf and the right child \( v_2 \) is not a leaf. (The opposite case is similar). The RAKE operation deletes \( v_1 \). Knowing one of its argument \( L(v_1) = (a_1, \cdots, a_k) \), \( u \) will be associated with an unary function \( F^1(u) \) defined as:

\[
F^1(u)(X) = F(u)(L(v_1), X)
\]

where the argument \( X = (x_1, \cdots, x_k) \) is the unknown final vector label \( L(v_2) \). We must find a suitable representation for this unary function \( F^1(u) \).

Suppose \( F^1(u)(X) = Y = (y_1, \cdots, y_k) \). Then we have:

\[
y_i = \max_{j \in \{a_i + x_j | f(i, j) = l\}} = \max_j \{\max_i \{a_i | f(i, j) = l\} + x_j\} = \max_j \{c^l_j + x_j\}
\]

where \( c^l_j = \max_i \{a_i | f(i, j) = l\} \). Hence, \( F^1(u) \) can be represented by \( k^2 \) parameters \( \{c^l_j | 1 \leq l, j \leq k\} \). These parameters can be computed by the processor \( P_u \) in \( O(k^2) \) time.

Suppose that \( v \) is a leaf of \( T \) and is the only child of its parent \( u \). In this case the final label \( L(v) = (a_1, \cdots, a_k) \) is known and \( u \) is associated with an unary function \( F^1(u) \) which is represented by \( k^2 \) parameters \( \{c^l_j | 1 \leq j, l \leq k\} \). The RAKE operation deletes \( v \) and evaluates the final label \( L(u) = F^1(u)(L(v)) \). \( L(u) \) can be evaluated by the processor \( P_u \) from the parameters \( \{c^l_j\} \) and the vector \( (a_1, \cdots, a_k) \) in \( O(k^2) \) time.

We next consider the COMPRESS operation. Let \( u \) and \( v \) be two vertices on a maximal chain of \( T \) where \( u \) is the parent of \( v \). Suppose \( u \) and \( v \) are associated with unary functions \( F^1(u) \) and \( F^1(v) \), respectively. Suppose \( F^1(v) \) has parameters \( \{c^l_j | 1 \leq j, l \leq k\} \) and \( F^1(u) \) has parameters \( \{d^l_i | 1 \leq l, i \leq k\} \). That is:
$F^1(v)(x_1, \cdots, x_k) = (y_1, \cdots, y_k)$ where $y_i = \max_j\{c_j + x_j\}$, and

$F^1(u)(y_1, \cdots, y_k) = (z_1, \cdots, z_k)$ where $z_i = \max_i\{d_i^t + y_i\}$.

Hence, $z_i = \max_{i}\{d_i^t + y_i\} = \max_{j}\{d_j^t + \max_j\{c_j^t + x_j\}\}$

$= \max_{j}\{\max_{i}\{d_i^t + c_j^t\} + x_j\} = \max_{j}\{e_j^t + x_j\}$, where $e_j^t = \max_{i}\{d_i^t + c_j^t\}$.

Thus, the composite function $F^1(u) \circ F^1(v)$ can be represented by the parameters

$\{e_j^t | 1 \leq i, j \leq k\}$. These parameters can be computed by the processor $P_u$ in $O(k^3)$ time.

Other parts of Algorithm 2.1 need not be changed. Since each operation can be performed in at most $O(k^3)$ time and the loop bodies of Algorithm 2.1 are executed at most $O(\log n)$ iterations, the entire algorithm takes $O(k^3 \log n)$ time with $O(n)$ processors as claimed. □

In the next two sections, we will apply the generalized BTAC algorithm to solve some maximum subgraph problems on TTSP graphs.

Remark: If we replace the term "max" by "min", replace the symbol $-\infty$ by $+\infty$ and define $+\infty + x = +\infty$ and $+\infty > x$ for any real number $x$, the generalized BTAC problem on $S$ can be defined similarly. All the discussions and the results presented above remain true. The corresponding algorithm can be used to solve minimum subgraph problems on TTSP graphs.

6.2 Weighted Maximum Matching Set Problem

Bern, Lawler, and Wong [BLW] developed a powerful methodology for designing sequential algorithms for solving many combinatorial problems on TTSP graphs. These problems include: maximum matching set, maximum independent set, minimum dominating set problems etc. We will show that the
sequential algorithms for these problems can be converted into instances of the
generalized BTAC problem and hence be solved by the algorithm presented in
Section 6.1.

In this section, we design an efficient parallel algorithm for the weighted
maximum matching set problem. Parallel algorithms for other problems listed
above can be developed by using similar techniques.

Let \( G = (V,E) \) be a TTSP graph. Suppose that each edge \( e \) of \( G \) is given a
weight \( w(e) \). Let \( M \subseteq E \) be a subset of edges of \( G \). We denote by
\( w(M) = \sum_{e \in M} w(e) \) the total weight of the edges in \( M \). If \( M = \emptyset \), define
\( w(M) = 0 \). The weighted MMS problem is to find a matching set \( M \) in \( G \) with
maximal weight \( w(M) \).

By this definition, an ordinary maximum matching set of \( G \) is just a max­
imum weight matching set of \( G \) where each edge of \( G \) has weight 1.

In order to solve the weighted MMS problem on TTSP graphs, we consider
the pairs of the form \((G,M)\) where \( G = (V,E) \) is a TTSP graph with source \( s \) and
sink \( t \), and \( M \subseteq E \) is a subset of the edges of \( G \). According to whether \( M \) is
a matching set of \( G \) or not, and whether the sink \( t \) and the source \( s \) is matched
in \( M \) or not, the pair \((G,M)\) can be in one of the five states \( \{s_1, \ldots, s_5\} \) defined
as follows: The state \( s_1 \) consists of the pairs \((G,M)\) where \( M \) is a matching set
of \( G \), both \( s \) and \( t \) are matched in \( M \). The state \( s_2 \) consists of the pairs where
\( M \) is a matching set, \( s \) is matched, \( t \) is not. The state \( s_3 \) consists of the pairs where
\( M \) is a matching set, \( t \) is matched, \( s \) is not. The state \( s_4 \) consists of the pairs where
\( M \) is a matching set, neither \( s \) nor \( t \) is matched. The state \( s_5 \) con­sists of the pairs where \( M \) is not a matching set. Define \( S = \{s_1, \ldots, s_5\} \).
Figure 24. The multiplication table for the weighted MMS problem.
Let \((G_1, M_1)\) and \((G_2, M_2)\) be two pairs in the states \(s_i\) and \(s_j\), respectively. Consider the series composition \(G_1 \ast G_2\) of \(G_1\) and \(G_2\) and the pair \((G_1 \ast G_2, M_1 \cup M_2)\). This pair is in a new state \(s_i\) which is completely determined from \(s_i\) and \(s_j\). According to this correspondence, we can induce a binary function \(f_s : S \times S \to S\) as follows: If the pair \((G_1, M_1)\) is in the state \(s_i\) and the pair \((G_2, M_2)\) is in the state \(s_j\), then \(f_s(s_i, s_j)\) is the state of the pair \((G_1 \ast G_2, M_1 \cup M_2)\). Similarly, we can introduce a binary function \(f_p : S \times S \to S\) for the parallel composition rule.

The multiplication table of these two functions are shown in Figure 24. In this table, each state \(s_i\) \((1 \leq i \leq 5)\) is represented by an intuitive figure. \(\otimes\) denotes a matched vertex and \(\bigcirc\) denotes an unmatched vertex. Wiggly lines denote the edges in the set \(M\). The row index of the table is the first argument. The column index of the table is the second argument. Each entry in the table has two figures. The first figure is the value for \(f_s\), the second figure is the value for \(f_p\). It is straightforward to check the correctness of this table.

Let \(T\) be a decomposition tree of a TTSP graph \(G = (V, E)\). Each leaf \(v\) of \(T\) represents a primitive TTSP graph. Let \(G_v\) denote this primitive TTSP graph and let \(e_v\) denote the unique edge in \(G_v\). Each internal vertex \(u\) of \(T\) represents a TTSP graph, denoted by \(G_u\). The root \(r\) of \(T\) represents the original TTSP graph \(G\).

In order to compute the weight of a maximum weight matching set of \(G\), we will compute a vector label \(L(u) = (a_1, \cdots, a_5)\) for each vertex \(u\) of \(T\) such that for each \(1 \leq i \leq 5\), \(a_i = \max\{w(M) \mid M \subseteq E(G_u)\text{ and } (G_u, M)\text{ is in the state } s_i\}\). (If no \(M \subseteq E(G_u)\) such that the pair \((G_u, M)\) is in the state \(s_i\), let \(a_i = -\infty\).)
For each leaf $v$ of $T$, according to whether the edge $e_v$ of $G_v$ is included in $M$ or not, the pair $(G_v,M)$ can be either in the state $s_1$ or $s_4$. These are the only states that the pair $(G_v,M)$ might be in. Thus, we label $v$ by the vector $L(v) = (w(e_v), -\infty, -\infty, 0, -\infty)$.

Let $u$ be an internal vertex of $T$ with label $s$. Suppose $v_1$ and $v_2$ are two children of $u$ with labels $L(v_1) = (a_1, \ldots, a_5)$ and $L(v_2) = (b_1, \ldots, b_5)$, respectively. Then the label $L(u) = (c_1, \ldots, c_5)$ can be computed by the following formulas: $c_i = \max_{i,j} \{a_i + b_j \mid f_s(s_i, s_j) = s_l\}$ $(1 \leq i \leq 5)$. If $u$ has a label $p$, $L(u) = (c_1, \ldots, c_5)$ can be computed by the following formulas: $c_i = \max_{i,j} \{a_i + b_j \mid f_p(s_i, s_j) = s_l\}$ $(1 \leq i \leq 5)$.

It is easy to see that the computation of these final labels is exactly an instance of the generalized BTAC problem performed on $T$. This observation suggests the following parallel algorithm for computing the weight of a maximum weight matching set of $G$: First, label each leaf $v$ of $T$ by the vector $L(v) = (w(e_v), -\infty, -\infty, 0, -\infty)$. Label each internal vertex $u$ either by the function $f_s$ or by the function $f_p$ depending on which composition rule is used at $u$. Then apply the generalized BTAC algorithm on $T$. Upon termination, the root $r$ of $T$ gets a label $L(r) = (a_1, \ldots, a_5)$. The weight of a maximum weight matching set of $G$ will be $\max\{a_1, a_2, a_3, a_4\}$. (Since the states $s_1, \ldots, s_4$ represent that the edge set $M$ in the pair $(G,M)$ is a matching set in $G$).

If $G$ has $m$ vertices, the decomposition tree $T$ of $G$ has $2m - 1$ nodes. Thus by Theorem 6.1, this algorithm runs in $O(\log m)$ time with $O(m)$ processors on an EREW PRAM.
The algorithm described above only computes the weight of a maximum weight matching set of $G$. In order to actually construct such a maximum weight matching set, we can modify the algorithm as follows: At each internal vertex $u$ of $T$, when computing the label $L(u) = (c_1, \cdots, c_5)$, the algorithm also records which pair of $(a_i, b_j)$ (with the maximal $a_i + b_j$ value) is used to obtain $c_l$ for each $1 \leq l \leq 5$. This information can be used to determine which edge should be included in the maximum weight matching set of $G$. The time and the processor bounds of the modified algorithm are not changed. In summary we have:

**Theorem 6.2:** If the input is given by the decomposition trees, the weighted MMS problem on TTSP graphs can be solved in $O(\log m)$ time with $O(m)$ processors on an EREW PRAM where $m$ is the number of edges in the input TTSP graph. □

**Remark:** This method can also be applied to obtain parallel algorithms for other TTSP graph problems listed at the beginning of this section. The only task we have to do is the construction of the set $S$ and the multiplication tables of the functions $f_s$ and $f_p$ defined on $S$. Of course, for each different problem, the multiplication table is different. Readers can find a thorough discussion in [BLW] on how these tables can be constructed.

### 6.3 Maximum Path and Maximum Cycle Problems

In this section we apply the generalized BTAC algorithm to solve two other problems, the *maximum path* and the *maximum cycle* problems, on TTSP graphs. The method used in this section is slightly different from the approach in Section 6.2.
Let \( G = (V, E) \) be a TTSP graph with source \( s \) and sink \( t \). Suppose that each edge \( e \) of \( G \) is given a weight \( w(e) \). For any subset \( M \subseteq E \), let \( w(M) \) denote the total weight of the edges in \( M \). The \textit{maximum cycle (MC)} problem is to find a simple cycle (ignore the direction of edges of \( G \)) \( C \) in \( G \) with maximum total weight \( w(C) \). The \textit{maximum path (MP)} problem is to find a simple path \( P \) in \( G \) from the source \( s \) to the sink \( t \) with maximum total weight \( w(P) \).

Let \( C(G) \) denote the maximum of the total weight of any simple cycle in \( G \). Let \( P(G) \) denote the maximum of total weight of any path in \( G \) from \( s \) to \( t \). We only discuss the parallel algorithm for computing the values of \( C(G) \) and \( P(G) \). The maximum weight cycle and the maximum weight path can be actually constructed by using the method mentioned in Section 6.2.

**Lemma 6.3**: Let \( G_1 \) and \( G_2 \) be two given TTSP graphs. Let \( G_s = G_1 \ast G_2 \) be the series composition of \( G_1 \) and \( G_2 \) and let \( G_p = G_1 \parallel G_2 \) be the parallel composition of \( G_1 \) and \( G_2 \). Then:

1. \( C(G_s) = \max(C(G_1), C(G_2)) \);
2. \( P(G_s) = P(G_1) + P(G_2) \);
3. \( C(G_p) = \max(C(G_1), C(G_2), P(G_1) + P(G_2)) \);
4. \( P(G_p) = \max(P(G_1), P(G_2)) \);

**Proof**: The lemma follows from the following observations.

1. Any simple cycle in \( G_s \) is either a simple cycle in \( G_1 \) or in \( G_2 \).
2. Any path in \( G_s \) from the source to the sink is a concatenation of one path in \( G_1 \) and one path in \( G_2 \).
(3) Any simple cycle in \( G_p \) is either a simple cycle in \( G_1 \), or a simple cycle in \( G_2 \), or a concatenation of one path in \( G_1 \) and one path in \( G_2 \).

(4) Any path in \( G_p \) from the source to the sink is either a path in \( G_1 \), or a path in \( G_2 \). □

In order to solve the MC and the MP problems on TTSP graphs, we define 
\[ S = \{ s_1, s_2, s_3 \} \]
and introduce two partial binary functions \( f_s \) and \( f_p \) on \( S \) as in Figure 25. \( (s_1, s_2, s_3) \) are the symbols used to define the functions \( f_s \) and \( f_p \). These two functions describe the equations in Lemma 6.3).

Let \( T \) be a decomposition tree of a TTSP graph \( G = (V, E) \). Each leaf \( v \) of \( T \) represents a primitive TTSP graph with an edge \( e_v \). We label each leaf \( v \) by \((0, w(e_v), 0)\) and label each internal vertex \( u \) of \( T \) by \( f_s \) or \( f_p \) depending which composition rule is used at \( u \). Then, we perform the generalized BTAC algorithm on \( T \). (Although \( f_s \) and \( f_p \) are partial functions, the generalized BTAC algorithm can still be applied on \( T \). The only modification needed is as follows: when a processor looks up the multiplication table of \( f_s \) or \( f_p \) and sees an undefined entry, the processor ignores the entry without updating any information stored in the common memory).

Let \( u \) be a vertex of \( T \) and suppose that \( L(u) = (a_1, a_2, a_3) \) is the final label of \( u \) after the algorithm terminates. From Lemma 6.3, the definition of \( f_s \) and \( f_p \), and by using induction on the structure of \( G \), one can prove:

(a) \( a_1 = C(G_u) \);
(b) \( a_2 = P(G_u) \);
(c) \( a_3 = 0 \).
Figure 25. The multiplication table for the MC and MP problems.
In particular, if the root \( r \) of \( T \) gets a label \( L(r) = (c_1, c_2, c_3) \), we have \( C(G) = C(G_r) = c_1 \) and \( P(G) = P(G_r) = c_2 \). Thus, we conclude:

**Theorem 6.4:** If the input is given by the decomposition tree, the MC and the MP problems on TTSP graphs can be solved in \( O(\log m) \) time with \( O(m) \) processors on an EREW PRAM. □

**Remark 1:** We have presented a general methodology for designing efficient parallel algorithms for solving various problems on TTSP graphs, by assuming that the input is decomposition tree. If the input is given by edge list, we have to run the recognition and decomposition algorithm (Chapter V) in order to construct a decomposition tree. In this case the resource bounds of the algorithms will be increased to \( O(\log^2 n + \log m) \) time with \( O(n + m) \) processors on an EREW PRAM. Thus, the decomposition process is the bottle-neck in solving TTSP graph problems. Any improvement over the decomposition algorithm presented in Chapter V will uniformly improve all the parallel algorithms developed in this chapter.

**Remark 2:** The only property of TTSP graphs that is essential in our methodology is that a TTSP graph can be represented by a binary tree. Bern, Lawler, and Wong [BLW] mentioned that some other classes of graphs also have this property. (For example, outerplanar graphs, Halin networks, bandwidth-limited networks and so on). Therefore, the methodology presented in this chapter can also be applied to design efficient parallel algorithms for these graphs.
CHAPTER VII
DEPTH FIRST SPANNING TREE PROBLEM
IN PLANAR GRAPHS

In this chapter we present a parallel algorithm for constructing depth first spanning trees in planar graphs. The algorithm runs in $O(\log^2 n)$ time with $O(n)$ processors on an ARBITRARY CRCW PRAM (this model will be simply referred to as CRCW PRAM in this chapter). Our algorithm is within an $O(\log^2 n)$ factor of optimal and greatly improves the best previously known algorithm for the problem which runs in $O(\log^3 n)$ time with $O(n^4)$ processors on a CREW PRAM [Sm].

In Section 7.1, we review the parallel algorithm for this problem developed by Smith [Sm]. In Sections 7.2 and 7.3, we develop the key subroutine for our new algorithm. Section 7.4 discusses the implementation of the algorithm.

7.1 Background

The depth first spanning tree (dfst) of graphs have been used as the basis for many graph algorithms [Ta]. Linear time sequential algorithms (the greedy method) for constructing a dfst is well known [Ta]. It has been conjectured, however, that the construction of a dfst in a general graph is inherently sequential. Actually, Reif shows that the problem of constructing the dfst produced by
the greedy method is *P*-complete (see [Re], [An], and [AA]). In other words, this problem is in *NC* iff *P* = *NC*, which is considered extremely unlikely. (However, there may exist some other method, completely different from the greedy method, that constructs a dfst in polylogarithmic time with polynomially many processors). In spite of many attempts, the dfst problem for general graphs is still not known to be in *NC*.

Recently, Smith discovered a nice parallel algorithm which finds a dfst in a planar graph in \(O(\log^3 n)\) time with \(O(n^4)\) processors [Sm]. This work is important since it is the first such work showing that the dfst problem for planar graphs is in *NC*. However, due to the importance of the dfst in designing graph algorithms, it is very desirable to have a more efficient parallel algorithm.

In this chapter we develop such an algorithm. Our algorithm takes only \(O(\log^2 n)\) time with \(O(n)\) processors on a CRCW PRAM. It is within an \(O(\log^2 n)\) factor of optimal and is a substantial improvement over Smith's algorithm. The model we use is stronger than the CREW PRAM model used by Smith. However, one step of our model can be simulated by the weaker model in \(O(\log n)\) steps. In particular, [Vi] shows that this simulation can be performed in \(O(\log^2 n)\) steps by using parallel sorting. Later, [Col] shows that the sorting can be done in \(O(\log n)\) time with \(O(n)\) processors on a CREW PRAM. Hence, by combining the algorithms in [Vi] and the sorting algorithm in [Col], this simulation can be performed in \(O(\log n)\) time with \(O(n)\) processors on a CREW PRAM. Thus, our algorithm can be implemented on a CREW PRAM in \(O(\log^3 n)\) time with \(O(n)\) processors, still much better than Smith's algorithm.
Throughout this chapter, \( G = (V,E) \) denotes a connected planar graph, \( n \) denotes the number of vertices of \( G \).

A depth first spanning tree \( T \) of \( G \) rooted at a specified vertex \( r \in V \) can be constructed as follows: Starting from \( r \), successively add edges to \( T \) until a vertex is reached all of whose exit edges (i.e. incident edges other than the one used to arrive at the vertex) are incident upon \( T \). At this point, backtrack a minimum distance until a vertex is reached that has exit edges not incident upon \( T \) and continue. Repeat this process until all vertices of \( G \) are exhausted \([Ta]\). The DFST problem considered in this chapter is the problem of constructing a dfst for a given planar graph \( G \) with a specified root \( r \).

Suppose that \( G \) is embedded in the plane. \( G \) partitions the plane into a number of connected regions. Each connected region is called a *face*. The unbounded face is called the *outer face*.

The combinatorial representation of an embedded plane graph \( G = (V,E) \) is defined as follows: Each edge \( e = (x,y) \in E \) is represented by two directed *darts*: \( (x,y) \) and \( (y,x) \). The combinatorial representation of \( G \) consists of two lists of these darts. The first list \( L_1 \) is ordered so that for each vertex \( v \in V \), the darts "leaving \( v \)" are consecutive in \( L_1 \) and are in the order they appear in the embedding in counterclockwise direction. The second list \( L_2 \) is ordered so that for each face \( f \) of \( G \), the darts on the boundary of \( f \) are consecutive in \( L_2 \) and are in the order they appear in the embedding in clockwise direction. Given a planar graph \( G \), the combinatorial representation of a plane embedding of \( G \) can be found in \( O(\log^2 n) \) time with \( O(n) \) processors on a CRCW PRAM (Klein and Reif [KR]). We will use this representation in our algorithm.
Let $C$ be a (simple) cycle of $G$. The vertex set of $G$ is partitioned into three subsets: the vertices on $C$, the vertices in the interior of $C$ and the vertices in the exterior of $C$. We use $\text{int}(C)$ ($\text{ext}(C)$) to denote the interior (exterior) of $C$. If $|\text{int}(C)| \leq cn$ and $|\text{ext}(C)| \leq cn$ for some constant $c < 1$, $C$ is called a $cn$ separating cycle of $G$.

Let $T$ be any spanning tree of $G$ rooted at a vertex $t \in V$. $T$ introduces a partial order on $V$: $v_1 < v_2$ if $v_2$ is an ancestor of $v_1$ in $T$. In general, if $v_1$ and $v_2$ are two arbitrary vertices they may be not comparable in the partial order induced by $T$. A dfst $T$ of $G$ can be characterized by the following well known fact:

**Lemma 7.1 [Sm]:** A spanning tree $T$ of $G = (V, E)$ is a dfst iff for any edge $e \in E$ the end vertices of $e$ are comparable in the partial order induced by $T$. □

Let $P_r$ be a (simple) path in $G$ with $r$ as one of its end vertices. Suppose that $G - P_r$ (which is the graph obtained from $G$ by deleting all vertices in $P_r$ and all edges incident to a vertex in $P_r$) is the union of the connected components $\{G_i\}_{1 \leq i \leq k}$. Let $e$ be an edge of $G$. Define $e$ to be:

(a) A touching edge of $G_i$ if one end vertex of $e$ is in $G_i$ and another end vertex of $e$ is on $P_r$ --- call this vertex the point where $e$ touches $P_r$.

(b) An inessential touching edge of $G_i$ if it is a touching edge of $G_i$ and there exists another touching edge $e'$ of $G_i$ which touches $P_r$ at a point further (in $P_r$) from $r$ than the point at which $e$ touches $P_r$.

(c) An essential touching edge of $G_i$ if it is a touching edge of $G_i$ and the conditions in statement (b) are not satisfied.
Let $e_i$ be an essential touching edge of $G_i$ ($1 \leq i \leq k$) and let $x_i$ be the end vertex of $e_i$ in $G_i$. Suppose $T_i$ is a dfst of $G_i$ rooted at $x_i$ ($1 \leq i \leq k$). From lemma 7.1, it is easy to show:

**Lemma 7.2 [Sm]:** Let $T$ be the union of $P_r$, $\{e_i\}$ ($1 \leq i \leq k$) and $\{T_i\}$ ($1 \leq i \leq k$). Then $T$ is a dfst of $G$ rooted at $r$. □

Figure 26 shows an example of lemma 7.2. The planar graph in Figure 26 is divided by the path $P_r$ into two connected components $G_1$ and $G_2$. The edges $e_1$ and $e_2$ are two touching edges of $G_1$ where $e_2$ is the essential touching edge of $G_1$. The edges $e_3$ and $e_4$ are two touching edges of $G_2$ where $e_4$ is the essential touching edge of $G_2$. $T_1$ and $T_2$ are the dfst of $G_1$ and $G_2$, respectively. The union of $P_r$, $e_2$, $e_4$, $T_1$, and $T_2$ is a dfst of $G$ rooted at $r$.

![Figure 26. An illustration of Lemma 7.2.](image)

Smith's algorithm works as follows: Find a path $P_r$ in $G$ with $r$ as one end vertex such that $G - P_r$ is the union of connected components $\{G_i|1 \leq i \leq k\}$ and
$|G_i| \leq cn$ for some constant $c < 1$. Then, find an essential touching edge $e_i$ for each $G_i$ ($1 \leq i \leq k$) and let $x_i$ be the end vertex of $e_i$ in $G_i$. Next, recursively call the algorithm on each $G_i$ ($1 \leq i \leq k$) in parallel to construct a dfst $T_i$ of $G_i$ rooted at $x_i$. Finally, construct a dfst $T$ for $G$ rooted at $r$ as in lemma 7.2.

In order to find the separating path $P_r$, Smith uses the following method: First find a $2n/3$ separating cycle $C$ in $G$. Then, find a path $P_1$ from the root $r$ to an arbitrary vertex $x$ on $C$. Let $e$ be an edge on $C$ incident to $x$. Define $P_r = P_1 \cup C - \{e\}$. Since each connected component of $G - P_r$ is either in the interior or in the exterior of $C$, they are bounded by $2n/3$ in size. Thus, $P_r$ is the required separating path.

The problem now is reduced to the construction of a $2n/3$ separating cycle $C$ in $G$. This subroutine is the most resource consuming part of Smith's algorithm. It uses $O(n^4)$ processors and $O(\log^2 n)$ time. In Sections 7.2 and 7.3, we will present a parallel algorithm for finding a $7n/8$ separating cycle in $G$ that uses only $O(\log n)$ time with $O(n)$ processors on a CRCW PRAM.

Independently, Miller has developed a parallel algorithm for finding a separating cycle in a planar graph [Mi] that takes $O(\log n)$ time with $O(n)$ processors on a CRCW PRAM. Our techniques are quite different from Miller's and may find applications in solving other planar graph problems.

Our method consists of two phases. The first phase finds a path $P$ in $G$ such that $P$ separates the vertices of $G$ into two subsets $A$, $B$ and neither $A$ nor $B$ contains more than $2n/3$ vertices. (The designated root $r$ is not necessarily an end vertex of $P$, so $P$ is not the separating path $P_r$ described above). This phase is discussed in Section 7.2. The second phase modifies $P$ to be a $7n/8$
separating cycle $C$ of $G$. This phase is discussed in Section 7.3. The whole process takes only $O(\log n)$ time with $O(n)$ processors on a CRCW PRAM. The complete dfst algorithm is presented in Section 7.4.

7.2 Finding a Separating Path

Our depth first spanning tree algorithm will work for any planar graph $G$. However, in Sections 7.2 and 7.3 we assume that $G$ is a biconnected planar graph given by the combinatorial representation form. When the subroutine developed in these two sections is called from the main algorithm (Section 7.4), we will prepare $G$ such that these conditions are satisfied.

The goal of the first phase is to find a path $P$ in $G$ so that $P$ divides the vertices of $G$ into two subsets $A, B$ and both $A$ and $B$ contain $\leq 2n/3$ vertices. To do so, we first construct an auxiliary graph $G_t$ and find a $2n/3$ separating cycle $C_e$ in $G_t$. Then, we modify $C_e$ to be a separating path $P$ of $G$.

Let $T$ be any spanning tree of $G=(V,E)$. Make each face (including the outerface) of $G$ a triangle by adding a suitable number of additional edges such that the added edges do not intersect with each other. This graph, denoted by $G_t$, is called a triangulation of $G$. Since the vertex set of $G_t$ is $V$, $T$ is also a spanning tree of $G_t$. $G_t$ has a natural plane embedding inherited from the embedding of $G$. Since $G_t$ is a planar graph with $|V|=n$ vertices, the total number of edges and faces in $G_t$ is $O(n)$ [BM]. The edge set of $G_t$ can be partitioned into three parts: the tree edges in $T$, the edges in $G$ but not in $T$, and the added edges. We denote by $NTE$ the nontree edges in $G_t$ (namely the union of the second and the third parts). Each nontree edge $e \in NTE$ introduces a
cycle, denoted by $C_e$, with some tree edges in $T$. Let $P_e = C_e - \{e\}$ denote the corresponding path in $T$. The following lemma is proved by Lipton and Tarjan [LT1] and serves as the basis for our algorithm.

**Lemma 7.3 [LT1]:** For any spanning tree $T$ and triangulation $G_t$ of $G$, there exists an edge $e \in NTE$ such that neither $int(C_e)$ nor $ext(C_e)$ contains more than $2n/3$ vertices (i.e. $C_e$ is a $2n/3$ separating cycle of $G_t$). \(\Box\)

Notice that the cycle $C_e$ in this lemma is a cycle in $G_t$, not necessarily a cycle in $G$, since the edge $e$ satisfying this lemma may be an added edge. However, the corresponding path $P_e$ is a path in $G$ since it only consists of edges in $T$. Moreover, $P_e$ separates the vertices of $G$ into two subsets $int(C_e)$ and $ext(C_e)$. Thus, if the cycle $C_e$ in $G_t$ satisfying lemma 7.3 is found, the goal of the first phase is fulfilled: The path $P_e$ in $G$ is the required path. Hence, the problem is reduced to find the $2n/3$ separating cycle in $G_t$.

Lipton and Tarjan give a linear time sequential algorithm for finding $C_e$ [LT1]. Their algorithm, however, does not seem to be parallelizable. We develop new methods so that $C_e$ can be constructed efficiently in parallel.

Although lemma 7.3 is true for arbitrary spanning trees $T$ and triangulations $G_t$ of $G$, we will use a special spanning tree in order to make efficient parallel computation possible.

Let $f_0$ be the outerface of $G$ and let $e_0$ be an edge on the boundary of $f_0$. A spanning tree $T$ of $G$ is an *enclosing spanning tree* with *window* $e_0$ if it contains all edges on the boundary of $f_0$ except for $e_0$. For an enclosing spanning tree $T$, we can triangulate the outerface $f_0$ in such a way that if $e$ is an edge added in order to triangulate the face $f_0$, then $ext(C_e) = \emptyset$. Let $C_0$ be the cycle
induced by the window $e_0$. Then $C_0$ is exactly the boundary of $f_0$. Clearly $ext(C_0)=\emptyset$. For such a triangulation $G_t$ and an enclosing spanning tree $T$, if $e$ is an edge added in order to triangulate $f_0$, then $int(C_e) \supseteq int(C_0)$. Thus, if $C_e$ satisfies lemma 7.3, $C_0$ also satisfies lemma 7.3. Therefore, in order to find the edge satisfying lemma 7.3, we do not need to consider the edges added in order to triangulate the outerface $f_0$. Hence, we do not need to triangulate $f_0$. From now on, $T$ will denote a fixed enclosing spanning tree of $G$ with window $e_0$ and $G_t$ will denote a triangulation of $G$ where the outerface of $G$ is not triangulated. We will find an edge $e$ satisfying lemma 7.3 for this particular $T$ and $G_t$.

We will compute, for each $e \in NTE$, the number of vertices in $int(C_e)$ and $ext(C_e)$ and then determine the cycle $C_e$ satisfying lemma 7.3. In order to do this, we have to utilize another graph $G=(V,E)$ and a spanning tree $T^*$ of $G$ defined later. To avoid confusion, the members of $V$ are called nodes, the members of $E$ are called arcs. The node set $V$ is the set $NTE$. For each $e \in NTE$, we denote by $\bar{e}$ the corresponding node in $V$. The arc set $E$ is defined as: $E=\{(\bar{e}_1,\bar{e}_2) \mid e_1$ and $e_2$ are on the same face of $G_t\}$.

Given a plane embedding of $G_t$, there is a natural plane embedding for $G$. A node $\bar{e} \in V$ is represented by a point on the edge $e$. Let $f$ be a face of $G_t$ and let $e_1,e_2,e_3$ be three boundary edges of $f$. If two edges (say $e_1,e_2$) are tree edges in $T$, $\bar{e}_3$ is the only node in $V$ and no arcs in $E$ are contained in $f$. If only one edge (say $e_1$) is a tree edge in $T$, $\bar{e}_2$ and $\bar{e}_3$ are nodes in $V$ and the arc $(\bar{e}_2,\bar{e}_3)$ is in $E$ which can be drawn in $f$. If no edges are tree edges, $\bar{e}_1,\bar{e}_2,\bar{e}_3$ are nodes in $V$. Three arcs $(\bar{e}_1,\bar{e}_2), (\bar{e}_2,\bar{e}_3), (\bar{e}_3,\bar{e}_1)$ are in $E$ which can be drawn in $f$. 
Figure 27. The enclosing spanning tree and the dual tree.

Figure 28. The proof that $T^*$ is connected.
Figure 27 (1) shows a graph $G$, an enclosing spanning tree $T$, a triangulation $G_t$ and the corresponding graph $\overline{G}$.

**Lemma 7.4:** Let $T$ be the collection of arcs in $E$ constructed as follows:
For each face $f$ of $G_t$, if $f$ contains only one arc in $E$, include this arc in $T$; if $f$ contains three arcs in $E$, arbitrarily include two of them in $T$. Then $T$ is a spanning tree of $\overline{G}$.

**Proof:** First we show that $T$ is connected. It is enough to show that each node $\bar{e} \in V$ is connected to the node $\bar{e}_0$ by the arcs in $T$ where $e_0$ is the window of the enclosing spanning tree $T$. We prove this by induction on the number of faces in $G_t$.

If $G_t$ has only one face, $\overline{G}$ has only one node $\bar{e}_0$ and the claim is true.

Assume that the claim is true if $G_t$ has at most $k$ faces. Now suppose $G_t$ has $k + 1$ faces. Let $f$ be the face of $G_t$ containing $e_0$ on its boundary. Let $e_1$ and $e_2$ be two other edges on the boundary of $f$. We consider two cases.

Case 1: One of $e_1$ and $e_2$ (say $e_1$) is a tree edge. Consider the graph $G' = G_t - \{e_0\}$. $G'$ is a triangulated graph with $k$ faces, $T$ is an enclosing spanning tree of $G'$ with window $e_2$ and $G'$ contains at most $k$ faces. By the induction hypothesis, the node $\bar{e}$ in question is connected to the node $\bar{e}_2$ by the arcs in $T$. Since the arc $(\bar{e}_0, \bar{e}_2)$ is also in $T$, $\bar{e}$ is connected to $\bar{e}_0$ by the arcs in $T$ (Figure 28 (1)).

Case 2: $e_1$ and $e_2$ are not tree edges. Let $C_1$ ($C_2$) be the cycle induced by $e_1$ ($e_2$) in $T$. The nontree edge $e$ in question is either in $\text{int}(C_1)$ or in $\text{int}(C_2)$. Suppose it is in $\text{int}(C_1)$. (If it is in $\text{int}(C_2)$, the proof is similar). Let $G_1$ be the graph induced by the vertices in $C_1$ and $\text{int}(C_1)$. $G_1$ is a triangulated graph, the
restriction of $T$ to $G_1$ is an enclosing spanning tree of $G_1$ with window $e_1$. The number of faces of $G_1$ is less than that of $G_t$. By the induction hypothesis, $\bar{e}$ is connected to $\bar{e}_1$ by the arcs in $T$. Since the nodes $\bar{e}_0, \bar{e}_1, \bar{e}_2$ are connected by two arcs in $T$, $\bar{e}$ is connected to $\bar{e}_0$ by the arcs in $T$ (Figure 28 (2)). This completes the induction.

Next we show that $T$ contains no cycles. Suppose it does. Let $C = \bar{e}_1, \bar{e}_2, \ldots, \bar{e}_k$ be a cycle in $T$. Let $V_1$ be the set of vertices of $G_t$ which are enclosed in $C$ in the embedding. Any edge of $G_t$ connecting a vertex in $V_1$ and a vertex in $V - V_1$ must be one of $e_i$ for some $1 \leq i \leq k$. Since none of $e_i$ is a tree edge, no tree edges connect the vertices in $V_1$ and the vertices in $V - V_1$. This contradicts to the fact that $T$ is a spanning tree of $G_t$ (Figure 29). □
We designate $\bar{e}_0$ as the root of $T$. This gives all arcs in $T$ a direction: the direction is toward the root $\bar{e}_0$. Let $f$ be a face of $G_t$ whose three boundary edges $e_1, e_2, e_3$ are all nontree edges. If the arcs $(\bar{e}_2, \bar{e}_3)$ and $(\bar{e}_3, \bar{e}_1)$ are included in $T$, where $\bar{e}_1$ is the parent of $\bar{e}_2$ and $\bar{e}_2$ is the parent of $\bar{e}_3$ in $T$, we make a "short cut" in $T$: Replace the arc $(\bar{e}_3, \bar{e}_2)$ by the arc $(\bar{e}_3, \bar{e}_1)$ in $T$. Perform this modification on $T$ for all faces of $G_t$ wherever possible. The resulting graph, denoted by $T^*$, is still a spanning tree of $G$. $T^*$ is called the dual tree of $T$ in $G_t$, or simply a dual tree. Figure 27 (2) shows the dual tree $T^*$ of the graph $G$ shown in Figure 27 (1).

For the enclosing spanning tree $T$ of $G_t$, we arbitrarily select a vertex $q$ on the outerface of $G_t$ as the root. This gives the edges in $T$ a direction: the direction toward the root $q$. For each vertex $v \in V$, let $L_T(v)$ be the distance from $v$ to $q$ in $T$.

Before we turn to the discussion of how the dual tree $T^*$ can help our computation, we first present the algorithm for constructing the enclosing spanning tree $T$, the triangulation $G_t$, the dual tree $T^*$ and computing the values $L_T(v)$ ($v \in V$). This algorithm takes $O(\log n)$ time with $O(n)$ processors on a CRCW PRAM.

**Algorithm 7.1. Construct $G_t$ and $T^*$:**

Input: A planar graph $G = (V, E)$ given by combinatorial representation;

1. Construct an enclosing spanning tree $T$ of $G$: Let $v_1, \ldots, v_k$ be the vertices on the outerface $f_0$ of $G$. Arbitrarily select a vertex from $\{v_i\}_{1 \leq i \leq k}$ as the root $q$. Arbitrarily select an edge $e_0$ on $f_0$ as window. This can be done in $O(1)$ time with $O(n)$ processors. Next construct an arbitrary spanning
tree \( T_1 \) for the subgraph \( G - \{ v_1, \ldots, v_k \} \). This can be done in \( O(\log n) \) time with \( O(n) \) processors (Shiloach and Vishkin [SV2]). Finally, arbitrarily select an edge \( e \) connecting a vertex in \( \{ v_1, \ldots, v_k \} \) to a vertex in \( V - \{ v_1, \ldots, v_k \} \). Then \( T = T_1 \cup \{ e \} \cup \{ \text{edges on the boundary of } f_0 \text{ except } e_0 \} \) is an enclosing spanning tree of \( G \). By using the Euler tour technique (Tarjan and Vishkin [TV1]), \( T \) can be converted to the rooted version and \( L_T(v) \) can be computed in \( O(\log n) \) time with \( O(n) \) processors.

(2) Construct a triangulation \( G_t \): For each face \( f \) of \( G \), let \( v_1, \ldots, v_k \) be the vertices on \( f \) in that order. Add new edges \( (v_1, v_3), (v_1, v_4), \ldots, (v_1, v_{k-1}) \). These edges are added into the two lists of the combinatorial representation of \( G \) in a way that the resulting lists constitute a combinatorial representation for \( G_t \). This step takes \( O(1) \) time with \( O(n) \) processors.

(3) For each face \( f \) of \( G_t \), according to the condition on its boundary edges, choose proper arcs for \( T \) as in lemma 7.4. This step can be done in \( O(1) \) time with \( O(n) \) processors.

(4) By using the Euler tour technique [TV1], \( T \) can be converted to the rooted version with \( e_0 \) as the root in \( O(\log n) \) time with \( O(n) \) processors. Then, \( T \) is modified to form the dual tree \( T^* \) in \( O(1) \) time with \( O(n) \) processors.

End Algorithm 7.1.

We next turn to the construction of the cycle \( C_e \) in \( G_t \) satisfying lemma 7.3. For each nontree edge \( e \in NTE \), \( e \) introduces a cycle \( C_e \) in \( T \) and corresponds to a node \( \bar{e} \) in \( T^* \). We define four values for each \( \bar{e} \in T^* \):

\[
M(\bar{e}) = \min \{ L_T(v) \mid v \text{ is on the cycle } C_e \};
\]
\(O(\bar{e}) = |C_\bar{e}|\) = the number of vertices on the cycle \(C_\bar{e}\);
\(N(\bar{e}) = |\text{int}(C_\bar{e})|\) = the number of vertices in the interior of \(C_\bar{e}\);
\(P(\bar{e}) = |\text{ext}(C_\bar{e})|\) = the number of vertices in the exterior of \(C_\bar{e}\).

After these values have been computed, the edge \(e\) satisfying lemma 7.3 can be selected in \(O(1)\) time with \(O(n)\) processors on a CRCW PRAM by a concurrent write. Hence, the problem is reduced to the evaluation of these values. Since \(P(\bar{e}) = n - O(\bar{e}) - N(\bar{e})\), we only need to compute the first three values. (The value \(M(\bar{e})\) is not needed for this purpose. However, it will simplify the computation).

The computation is carried out in two passes. The first pass computes the values \(M(\bar{e})\) for all \(\bar{e} \in T^*\). The second pass computes the values \(O(\bar{e})\) and \(N(\bar{e})\) for all \(\bar{e} \in T^*\).

Both passes are performed on the dual tree \(T^*\). According to the conditions stated below, we distinguish four kinds of nodes in \(T^*\).

Type 1: \(\bar{e}\) is a leaf node in \(T^*\).

Type 2: \(\bar{e}\) has only one child \(\bar{e}_1\) in \(T^*\). Let \(f\) be the face of \(G_t\) containing the edges \(e\) and \(e_1\) on its boundary. The third boundary edge \(e_2\) of \(f\) is a tree edge. The direction of \(e_2\) in \(T\) is "opposite" to the direction of the arc \((\bar{e}_1, \bar{e})\) in \(T^*\) (Figure 30 (1), the dotted arrow is the arc in \(T^*\)).

Type 3: Similar to type 2, except that the direction of \(e_2\) in \(T\) is the "same" as the direction of \((\bar{e}_1, \bar{e})\) in \(T^*\) (Figure 30 (2)).

Type 4: \(\bar{e}\) has two children in \(T^*\).
Figure 30. The type 2 and type 3 nodes.

Figure 31. The computation of the first pass.
We consider the first pass. If $\bar{e}$ is a type 1 node, let $f$ be the face of $G_t$ such that $e$ is on the boundary of $f$ and two other boundary edges of $f$ are tree edges. Let $a, b, c$ be the three vertices on the boundary of $f$. Then $M(\bar{e}) = \min\{L_T(a), L_T(b), L_T(c)\}$ (Figure 31 (1)).

If $\bar{e}$ is a type 2 or 3 node with $\bar{e}_1$ as its child in $T^*$, let $f$ be the face of $G_t$ containing edges $e$ and $e_1$ on its boundary. Let $a$ be the vertex on $f$ which is an end vertex of $e$ but not an end vertex of $e_1$. Then $M(\bar{e}) = \min\{L_T(a), M(\bar{e}_1)\}$. (See Figure 31 (2) or Figure 31 (3)).

If $\bar{e}$ is a type 4 node with children $\bar{e}_1$ and $\bar{e}_2$ in $T^*$, let $f$ be the face of $G_t$ containing edges $e, e_1, e_2$ on its boundary. Then, we have $M(\bar{e}) = \min\{M(\bar{e}_1), M(\bar{e}_2)\}$ (Figure 31 (4)).

In order to compute $M(\bar{e})$ for all $\bar{e} \in T^*$, we first label each leaf node $\bar{e}$ of $T^*$ by the value $M(\bar{e})$ as described above. This value is completely determined from the local information $L_T(a), L_T(b), L_T(c)$ which has been computed in Algorithm 7.1. For an internal node $\bar{e}$ with children $\bar{e}_1$ and $\bar{e}_2$ (if $\bar{e}$ is a type 2 or 3 node, it has only one child), the value $M(\bar{e})$ is completely determined from the values of $M(\bar{e}_1), M(\bar{e}_2)$ and the local information $L_T(a)$ which is known already. It is easy to see that this computation is just a slight variation of the BTAC problem discussed in Chapter II. Therefore, it can be carried out by using the BTAC algorithm (Algorithm 2.1) in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM.

The computation of values of $O(\bar{e})$ and $N(\bar{e})$ (the second pass) is similar to the first pass.
If $\bar{e}$ is a type 1 node, we clearly have $O(\bar{e})=3$ and $N(\bar{e})=0$.

If $\bar{e}$ is a type 2 node with $\bar{e}_1$ as its child in $T^*$, we have (Figure 32 (1)):

\[
O(\bar{e}) = O(\bar{e}_1) + 1 \text{ and } N(\bar{e}) = N(\bar{e}_1).
\]

If $\bar{e}$ is a type 3 node, we distinguish two cases.

If $M(\bar{e}) < M(\bar{e}_1)$ (Figure 32 (2)), we have: $O(\bar{e}) = O(\bar{e}_1) + 1$ and $N(\bar{e}) = N(\bar{e}_1)$.

If $M(\bar{e}) = M(\bar{e}_1)$ (Figure 32 (3)), we have: $O(\bar{e}) = O(\bar{e}_1) - 1$ and $N(\bar{e}) = N(\bar{e}_1) + 1$.

---

(1) \hspace{1cm} (2)

(3) \hspace{1cm} (4)

Figure 32. The computation of the second pass.
If $\bar{e}$ is a type 4 node with children $\bar{e}_1$ and $\bar{e}_2$ in $T^*$, let $f$ be the face of $G_i$ containing $e, e_1$ and $e_2$ on its boundary. Let $a$ be the vertex on $f$ common to $e_1$ and $e_2$. Let $P$ be the path common to the two cycles induced by $e_1$ and $e_2$ (Figure 32 (4)). Then, $a$ is an end vertex of $P$. Let $b$ be another end vertex of $P$. It is easy to show that $L_T(b) = \max\{M(\bar{e}_1), M(\bar{e}_2)\}$. All vertices on $P$, except $b$, are in the interior of $C_e$. Since the length of $P$ is $L_T(a) - L_T(b) + 1$, we have:

(a) $O(\bar{e}) = O(\bar{e}_1) + O(\bar{e}_2) - 2(L_T(a) - L_T(b)) - 1$

(b) $N(\bar{e}) = N(\bar{e}_1) + N(\bar{e}_2) + (L_T(a) - L_T(b)) = N(\bar{e}_1) + N(\bar{e}_2) + L_T(a) - \max\{M(\bar{e}_1), M(\bar{e}_2)\}$.

In order to compute $O(\bar{e})$ and $N(\bar{e})$, we first label each leaf node $\bar{e}$ of $T^*$ by $N(\bar{e}) = 0$ and $O(\bar{e}) = 3$. For an internal node $\bar{e}$ of $T^*$ with children $\bar{e}_1$ and $\bar{e}_2$ in $T^*$ (if $\bar{e}$ is a type 2 or type 3 node, it has only one child), the value $O(\bar{e})$ is completely determined from the values $O(\bar{e}_1), O(\bar{e}_2)$ and the local information $L_T(a), M(\bar{e}_1), M(\bar{e}_2)$ which has been computed already. Similarly, $N(\bar{e})$ is completely determined from the values $N(\bar{e}_1), N(\bar{e}_2)$ and the local information $L_T(a), M(\bar{e}_1), M(\bar{e}_2)$. Thus, as in the first pass, these values can be computed by using the BTAC algorithm in $O(\log n)$ time with $O(n)$ processors. This completes the discussion of the second pass. The entire operation can be summarized as:

**Algorithm 7.2. Find a Separating Path:**

(1) Call Algorithm 7.1 to construct an enclosing spanning tree $T$, the triangulation $G_i$ and the dual tree $T^*$ of $G$;
(2) Perform the first pass to compute \( M(\bar{e}) \) for all \( \bar{e} \in T^* \);

(3) Perform the second pass to compute \( O(\bar{e}) \) and \( N(\bar{e}) \) for all \( \bar{e} \in T^* \) and let \( P(\bar{e}) = n - O(\bar{e}) - N(\bar{e}) \) for each \( \bar{e} \in T^* \);

(4) Select an edge \( e \) such that \( O(\bar{e}) \leq 2n/3 \) and \( P(\bar{e}) \leq 2n/3 \). This edge satisfies lemma 7.3.

End Algorithm 7.2.

From Algorithm 7.2 we have:

**Theorem 7.5:** The cycle \( C_e \) of \( G_t \) satisfying Lemma 7.3 can be found in \( O(\log n) \) time with \( O(n) \) processors. □

### 7.3 Finding a Separating Cycle

The cycle \( C_e \) in theorem 7.5 is a cycle in \( G_t \), not necessarily a cycle in \( G \) since the edge \( e \) may be an added edge. However, it is crucial for our algorithm to find a separating cycle in \( G \). In this section we discuss how to find a \( 7n/8 \) separating cycle \( C \) in \( G \).

Let \( C_e \) be the cycle in theorem 7.5 and let \( P = C_e - \{e\} \). Define \( A = \text{int}(C_e) \) and \( B = \text{ext}(C_e) \). Thus, \( |A| + |B| + |P| = n \). By lemma 7.3, \( |A| \leq 2n/3 \) and \( |B| \leq 2n/3 \). Without loss of generality, we assume \( |A| \geq |B| \).

If the edge \( e \) is indeed an edge in \( G \), we are done: the cycle \( C_e \) is a \( 2n/3 \) separating cycle in \( G \). So, we assume that \( e \) is not an edge of \( G \).
Figure 33. The path $P$ divides the boundary of the face $f$.

Figure 34. The construction of the separating cycle.
The path $P$ is a path in $G$. Moreover, two end vertices of $P$ are on the same face $f$ of $G$ (by the construction of $C_e$). Let $Q$ be the portion of the boundary of $f$ in $\text{int}(C_e)$ and let $R$ be the portion of the boundary of $f$ in $\text{ext}(C_e)$ (Figure 33).

If either $Q$ or $R$ (say $R$) does not intersect with $P$, let $C = P \cup R$. Then $C$ is a cycle in $G$. Since $|\text{int}(C)| = |\text{int}(C_e)| \leq 2n/3$ and $|\text{ext}(C)| = |\text{ext}(C_e)| \leq 2n/3$, $C$ is a $2n/3$ separating cycle in $G$ and we are done. So, we assume that both $Q$ and $R$ intersect with $P$.

Let $a_0, a_1, \ldots, a_s$ be the intersection vertices of $Q$ with $P$ in that order where $a_0$ and $a_s$ are two end vertices of $P$. Let $b_0, \ldots, b_t$ be the intersection vertices of $R$ with $P$ in that order where $b_0 = a_0$ and $b_t = a_s$. Since $G$ is assumed to be biconnected, the boundary of any face of $G$ is a simple cycle [Mi]. In particular, the vertices $\{a_0, a_1, \ldots, a_s, b_1, \ldots, b_{t-1}\}$ are distinct since they are on the boundary of the face $f$.

Let $C_i$ (1 ≤ $i$ ≤ $s$) be the cycle formed by the portion of $Q$ from $a_{i-1}$ to $a_i$ and the proper portion of $P$. Let $A_i$ be the interior of $C_i$. Let $D_j$ (1 ≤ $j$ ≤ $t$) be the cycle formed by the portion of $R$ from $b_{j-1}$ to $b_j$ and the proper portion of $P$. Let $B_j$ be the interior of $D_j$. (Figure 34 (1) shows an example where $s = t = 4$). Note that the cycles $C_i$ and $C_{i+1}$ (1 ≤ $i$ ≤ $s$) have one common vertex $a_i$.

If there exists an $i$ such that $|C_i| + |A_i| \geq n/8$, then $|\text{int}(C_i)| = |A_i| \leq |\text{int}(C_e)| \leq 2n/3$ and $|\text{ext}(C_i)| = n - (|C_i| + |A_i|) \leq n - n/8 = 7n/8$. Hence, $C_i$ is a $7n/8$ separating cycle in $G$ and we are done. So, we assume $|C_i| + |A_i| < n/8$ for all 1 ≤ $i$ ≤ $s$. 
Let \( i \) be the least index such that \( |\bigcup_{j=1}^{i}(C_i \cup A_i)| \geq |\bigcup_{j=i+1}^{s}(C_i \cup A_i)| \).

Define \( X = |\bigcup_{j=1}^{i}(C_i \cup A_i)| \), \( Y = |\bigcup_{j=i+1}^{s}(C_i \cup A_i)| \). Thus, \( X \geq Y \) and
\[
X - (|C_i| + |A_i|) + 1 = |\bigcup_{j=1}^{i-1}(A_i \cup C_j)| < |\bigcup_{j=i}^{s}(A_i \cup C_j)| = Y + (|C_i| + |A_i|) - 1.
\]
Hence we have: \( 0 \leq X - Y \leq 2(|C_i| + |A_i|) - 2 < n/4 - 2 \).

Since \( X + Y = |P \cup A| + 1 = |P| + |A| + 1 \), we get: \( 2X < n/4 + (|P| + |A|) - 1 \) and \( X < n/8 + (|P| + |A| - 1)/2 \).

Since \( 2X \geq X + Y = |P| + |A| + 1 \), we have: \( X = (|P| + |A| + 1)/2 \).

Let \( j \) be the least index such that \( b_j \) is further (on \( P \)) from \( a_0 \) than \( a_i \) is.

Let \( P' \) be the portion of \( P \) from \( a_i \) to \( b_j \). Let \( Q' \) be the portion of \( Q \) from \( a_0 \) to \( a_1 \cdots \) to \( a_i \). Let \( R' \) be the portion of \( R \) from \( b_0 \) to \( b_1 \cdots \) to \( b_j \). Finally define \( C = P' \cup Q' \cup R' \). Then, \( C \) is a cycle in \( G \). Figure 34 (1) is an example where \( i = j = 2 \). The thick lines in Figure 34 (1) are the boundary of the cycle \( C \). Since \( \text{int}(C) \subseteq (\bigcup_{i=1}^{i}(A_i \cup C_j)) \cup (\bigcup_{j=1}^{i}B_j) \), we have: \( |\text{int}(C)| \leq X + \sum_{j=1}^{i}|B_j| \leq X + |B| < n/8 + (|P| + |A| - 1)/2 + (n - (|P| + |A|)) = 9n/8 - (|P| + |A|)/2 - 1/2 \leq 9n/8 - n/4 - 1/2 < 7n/8 \).

Since \( \text{int}(C) \cup C \supseteq \bigcup_{i=1}^{i}(C_i \cup A_i) \), we get: \( |\text{int}(C)| + |C| \geq |\bigcup_{i=1}^{i}(C_i \cup A_i)| \) \( = X \geq (|P| + |A|)/2 + 1/2 \geq n/4 + 1/2 \). So, \( |\text{ext}(C)| = n - (|\text{int}(C)| + |C|) \leq 3n/4 - 1/2 < 7n/8 \). Thus, \( C \) is a \( 7n/8 \) separating cycle in \( G \) and we are done. (Figure 34 (2) shows another possible situation where \( i = 3 \) and \( j = 4 \). The analysis is almost identical).

The cycle \( C \) can be constructed as follows: First, we determine the intersection vertices \( a_0, \ldots, a_s \) of \( Q \) with \( P \). Let \( x_i \) \( (1 \leq i \leq s) \) be the number of vertices on the path \( Q \) from \( a_{i-1} \) to \( a_i \) (not including \( a_{i-1} \) and \( a_i \)). Then, we compute
the values \(|C_i|\) and \(|A_i|\). To do this, we perform Algorithm 7.2 once again on a special triangulation \(G_t'\) of \(G\). We triangulate the face \(f\) (whose boundary is the paths \(Q\) and \(R\)) as follows: add new edges \(e_i = (a_{i-1}, a_i)\) into \(f\) and add other new edges as needed. Other faces of \(G\) can be triangulated arbitrarily. Then we form the dual tree \(T^*\) for this special triangulation \(G_t'\) and perform Algorithm 7.2. Upon completion, we have \(|C_i| = O(\bar{e}_i) + x_i\) and \(|A_i| = N(\bar{e}_i) - x_i\) for each \(1 \leq i \leq s\). The whole process takes \(O(\log n)\) time with \(O(n)\) processors.

(Actually, \(|C_i|\) and \(|A_i|\) can be computed more efficiently by using the information obtained in the first phase. We choose this approach to simplify the presentation). Other parts of the construction of \(C\) are straightforward and take only \(O(1)\) time. In summary we have:

**Theorem 7.6:** Let \(G\) be a biconnected planar graph given by the combinatorial representation form. There is a parallel algorithm which finds a cycle \(C\) in \(G\) in \(O(\log n)\) time with \(O(n)\) processors on a CREW PRAM such that neither \(\text{int}(C)\) nor \(\text{ext}(C)\) contains more than \(7n/8\) vertices. □

### 7.4 DFST Algorithm

Our DFST algorithm is as follows. It is different from Smith's algorithm in that the key subroutine developed in the last two sections is implemented and the resources are allocated more carefully in order to achieve the \(O(\log^2 n)\) time and \(O(n)\) processor bound.

**Algorithm 7.3. DFST:**

**Input:** A connected planar graph \(G = (V,E)\) given by the edge list with a specified root \(r\);
Initialization: Find a plane embedding and a combinatorial representation for $G$;

(1) Find the biconnected components $\{G_j\} (1 \leq j \leq q$ for some $q)$ of $G$. Let $x_j$ be the vertex in $G_j$ which is closest to the root $r$. Perform the following steps on each $G_j (1 \leq j \leq q)$ simultaneously to construct a dfst $T_j$ rooted at $x_j$ for $G_j$. The union of $T_j (1 \leq j \leq q)$ will be a dfst of $G$ rooted at $r$ [Sm]. (To avoid double subscripts, we will use symbols $G$ and $r$ for $G_j$ and $x_j$ in the following statements).

(2) Find a $7n/8$ separating cycle $C$ in $G$ as in theorem 7.6;

(3) Find a path $P_1$ from the root $r$ to any vertex $x$ on $C$. Let $e$ be an edge in $C$ incident to $x$. Define $P_r = P_1 \cup C - \{e\}$;

(4) Find the connected components $G_i (1 \leq i \leq k$ for some $k)$ of $G - P_r$;

(5) For each $G_i (1 \leq i \leq k)$ find an essential touching edge $e_i$. Let $x_i$ be the end vertex of $e_i$ in $G_i$;

(6) Construct the combinatorial representation of the subgraphs $G_i$ from the combinatorial representation of $G$;

(7) Recursively call the algorithm on each $G_i (1 \leq i \leq k)$ in parallel to construct a dfst $T_i$ for $G_i$ rooted at $x_i$;

(8) Let $T = \bigcup_i (T_i \cup e_i) \cup P_r$. $T$ is a dfst of $G$ rooted at $r$.

End DFST.

To see that the algorithm takes $O(log^2 n)$ time with $O(n)$ processors on a CRCW PRAM, we make the following observations.
The initialization step can be done in $O(\log^2 n)$ time with $O(n)$ processors [KR]. This step is performed only once. When the algorithm is recursively called on a subgraph $G_i$ of $G$, the combinatorial representation of $G_i$ is constructed in a more efficient way.

Step (1) can be implemented by using the algorithm of Tarjan and Vishkin [TV1] in $O(\log n)$ time with $O(n)$ processors.

Step (2) can be performed by using the algorithm developed in the last two sections. It takes $O(\log n)$ time with $O(n)$ processors.

Step (3) can be implemented as follows: By using the algorithm of Shiloach and Vishkin [SV2], we first construct an arbitrary spanning tree $T$ of $G$ rooted at $r$. This can be done in $O(\log n)$ time with $O(n)$ processors. Next arbitrarily pick an vertex $x_0$ on $C$ and identify the path $P_0$ in $T$ from $x_0$ to $r$. This can be performed by using the standard pointer jump technique in $O(\log n)$ time with $O(n)$ processors. Let $x$ be the vertex on $P_0$ which is on $C$ and is closest to $r$. Let $P_1$ be the portion of $P_0$ from $x$ to $r$. Then $x$ and $P_1$ is what we want to find in step (3). So, step (3) totally takes $O(\log n)$ time with $O(n)$ processors.

Step (4) can be performed by using the algorithm in [SV2] in $O(\log n)$ time with $O(n)$ processors.

In order to implement step (5), we first compute the index number of the vertices on the path $P_r$ (i.e. $r$ has index number 1, the vertex on $P_r$ adjacent to $r$ has index number 2 and so on). The set $E_i$ ($1 \leq i \leq k$) of touching edges for each connected component $G_i$ can be identified in constant time. Next we sort each $E_i$ ($1 \leq i \leq k$) according to the index number of the end vertex of each $e \in E_i$ that is on $P_r$. Sorting can be done in $O(\log n)$ time with $O(n)$ processors [Col].
The essential touching edge of $G_i$ is the last edge in the sorted list of $E_i$. Thus, step (5) takes $O(\log n)$ time with $O(n)$ processors.

Step (6) is implemented as follows: Each subgraph $G_i$ has a natural plane embedding inherited from the embedding for $G$. The combinatorial representation for $G_i$ is computed from the lists $(L_1, L_2)$ of the combinatorial representation of $G$ as follows: In the list $L_1$, delete all darts that have one end vertex on $P_r$. Then perform a parallel sort on the remaining darts in $L_1$ such that the darts belonging to the same connected component $G_i$ are consecutive in the resulting list. Perform similar operations on $L_2$. This gives the combinatorial representation for each $G_i$. Since sorting can be done in $O(\log n)$ time with $O(n)$ processors [Col], step (6) can be performed within the same time and processor bounds.

Step (7) is the recursive call. Step (8) clearly takes constant time.

Since one application of the DFST algorithm on $G$ reduces the problem to the subgraphs of size at most $7n/8$, the depth of recursion is at most $\log_{8/7} n$. Since each step takes at most $O(\log n)$ time with $O(n)$ processors, the DFST algorithm takes $O(\log^2 n)$ time with $O(n)$ processors as claimed. In summary we have:

**Theorem 7.7:** Let $G = (V, E)$ be a planar graph given by edge list form and let $r$ be a specified vertex in $G$. A depth first spanning tree $T$ of $G$ rooted at $r$ can be constructed in $O(\log^2 n)$ time with $O(n)$ processors on an ARBITRARY CRCW PRAM. $\square$
CONCLUSIONS

This research has concentrated on the design of efficient parallel algorithms for solving various combinatorial problems on graphs and the development of new methodology for designing parallel algorithms.

The binary tree algebraic computation problem is defined and an efficient parallel algorithm for the BTAC problem is presented. Although the BTAC problem is defined algebraically, it is interesting to observe that many combinatorial problems on graphs are closely related to this problem. Thus, the BTAC algorithm provides a powerful and general tool for designing parallel algorithms for solving these types of problems. Most of the parallel algorithms presented in this dissertation utilize the BTAC methodology.

The BTAC algorithm is first applied to solve tree problems including: the minimum covering set, maximum independent set, maximum matching set, $r$-dominating set, and $p$-center problems. It is conceivable that efficient parallel algorithms for solving other combinatorial problems on trees may be obtained in a similar way.

We then apply the BTAC methodology to design parallel algorithms for solving TTSP graph problems. An efficient recognition and decomposition algorithm for TTSP graphs is developed. Using the decomposition tree produced by
this algorithm as input and a generalized BTAC algorithm as a basic tool, efficient parallel algorithms for many combinatorial problems on TTSP graphs are obtained systematically.

The method used in designing TTSP graph algorithms can be generalized to solve problems on other classes of graphs, provided that the graphs can be represented, somehow, by a tree structure. These classes of graphs include: Halin networks, outerplanar graphs, bandwidth-limited networks and so on (see [BLW]). Further investigation on parallel algorithms for solving problems on these graphs seems to be a good research area.

An efficient parallel algorithm for constructing depth first spanning trees in planar graphs was presented. This algorithm greatly improves the best previously known algorithm. It is possible that the "enclosing spanning tree --- dual tree" technique used in this algorithm may find applications in solving other planar graph problems.
APPENDICES

A. Computing the Number of Children of Vertices

Let \( T = (V, E) \) be a tree with vertex set \( V = \{1, 2, \ldots, n\} \). Suppose \( T \) is given by the parent pointer representation. Namely, the edge set \( E \) is given as a list \( \{(P(i), i) \mid P(i) \text{ is the parent of } i\} \) in arbitrary order. For every \( i \in V \), we need to compute the number of children of \( i \), and give each child of \( i \) an index number. In particular, if \( T \) is a binary tree, then the child with the index number 1 will be the left child of \( i \) and the child with index number 2 will be the right child of \( i \). We will show that this computation can be done in \( O(\log n) \) time using \( O(n) \) processors on an EREW PRAM.

We assume that the edges are given in an array \( EDGE[1..m] \) in the common memory (where \( m = n - 1 \) is the number of edges in \( T \)). We also need following arrays for the computation: \( INDEX[1..m]; LOW[1..n]; HIGH[1..n] \), and \( NUMBER[1..n] \). The meaning of these arrays will be discussed later.

By using the algorithm in [Col], we sort the edge set \( E \) according to the first vertex of each edge. This can be done in \( O(\log n) \) time using \( O(n) \) processors on an EREW PRAM [Col]. In Figure 35, the sorted edge list of the given tree is shown. (Notice that, in the sorted list, the edges with the same first vertex appear consecutively). Next, compute the "index number" of the edges in
the sorted list. Namely, the edge stored in $E D G E \ [1]$ has index number 1, the edge stored in $E D G E \ [2]$ has index number 2, and so on. In order to do this, we assign one processor $P_i$ to each memory cell $I N D E X \ [i]$. All $P_i \ (1 \leq i \leq m)$ write 1 into $I N D E X \ [i]$. Then, perform the "partial sum" algorithm [Re2] on array $I N D E X$. The index number of the edge stored in $E D G E \ [i]$ is now left in $I N D E X \ [i]$ (Figure 35). The "partial sum" computation can be done in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM [Re2].

Next, assign one processor $P_i$ to each vertex $i \ (1 \leq i \leq n)$ and perform the following program:

1. In parallel, each processor $P_i \ (1 \leq i < n)$ reads $I N D E X \ [i]$ and $E D G E \ [i]$. Suppose that $x_i$ is the value stored in $I N D E X \ [i]$, and $(a_i, b_i)$ is the edge stored in $E D G E \ [i]$.

2. In parallel, each processor $P_i \ (1 < i \leq n)$ reads $E D G E \ [i-1]$. Suppose $(c_i, d_i)$ is the edge stored in $E D G E \ [i-1]$.

3. In parallel, each processor $P_i \ (1 < i < n)$ do:

   If $a_i \neq c_i$, then write $x_i$ into $L O W \ [a_i]$ and write $x_i$ into $H I G H \ [c_i]$;

4. $P_1$ write 1 into $L O W \ [a_1]$; and $P_n$ write $n$ into $H I G H \ [c_{n-1}]$.

5. In parallel, each processor $P_i \ (1 \leq i \leq n)$ reads $L O W \ [i]$, $H I G H \ [i]$, compute the difference of the values stored in $H I G H \ [i]$ and $L O W \ [i]$, then write the result into $N U M B E R \ [i]$.

At this point, the value stored in $N U M B E R \ [i]$ is the number of children of $i$ in $T$ (see Figure 35).
In order to give each child of $i$ an index number, we partition array $EDGE$ into portions. Each portion consists of the edges with the same first vertex. Perform the "partial sum" computation on each portion and compute an index number for each edge within the portion. Now, if an edge $(i, j)$ gets an index number $k$, then $j$ is the $k$th child of $i$.

B. Computing the Number of Edges Between Vertices

Let $G=(V,E)$ be a multidag with vertex set $V=\{1,\ldots,n\}$ and $|E|=m$ edges. We need to compute the number of edges between each pair of vertices. This computation is very similar to the computation discussed in Appendix A.

Let $EDGE[1..m]$, $INDEX[1..m]$, $LOW[1..n,1..n]$, $HIGH[1..n,1..n]$, and $NUMBER[1..n,1..n]$ be the arrays stored in the common memory. The meaning of them are the same as in Appendix A.

We first sort the edge set $E$ of $G$ in array $EDGE$, using the first vertex of each edge as the primary key and the second vertex of each edge as the secondary key. In the sorted list, the edges between the same pair of vertices appear consecutively. Then, compute the index number of each edge in the sorted list (using array $INDEX$). Finally, perform similar program as in Appendix A. The number of edges between the vertex pair $i,j$ will be in $NUMBER[i,j]$. The entire procedure uses $O(m+n)$ processors and $O(\log m)$ time on an EREW PRAM.
Figure 35. Computing the number of children in a tree.
BIBLIOGRAPHY


[Be]: C. Berge, Graphs and Hypergraphs, North-Holland, Amsterdam.


Harold N. Gabow, Parallel Algorithms Using the Binarized Compressed Tree, private communication, 1986.


A. G. Goldman, Minimax Location of a Facility in an Undirected Tree Graph, Transportation Science, 6, 1972, pp. 407-418.


G. Y. Handler, Finding Two Centers of a Tree, the Continuous Case, Transportation Science, 12, 1978, pp. 93-106.

F. Harary, Graph Theory, Addison-Wesley, Reading, MA, 1969.


Xin He and Yaacov Yesha, Parallel Recognition and Decomposition of Two Terminal Series Parallel Graphs, Technical Report, OSU-CISRC-86TR1XH, 1986, accepted for publication in Information and Control.


[LY3]: M. Li and Y. Yesha, The Probabilistic and Deterministic Parallel Complexity of Symmetric Functions, 14th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, 267, pp. 326-335.


