Date: Fri, 16 Dec 88 15:28:26 EST
From: M R Anderson <USERLB7F@UMICHUM>
Subject: tree matching
To: PENGLI@BNR.CA
Message-Id: <3780127@UMICHUM.BITNET>

I am working on more sofisticated methods of combining syntax and semantics for automatic compiler generation, for which I think your paper may be useful. Anyway, I'd like to look into it.
I will look forward to receiving the hardcopy in the near future. As a possible alternative, if you have a TeX file for the paper, we could ftp it and I could generate the hardcopy myself right here. It would be up to you.
Thanks again.
Mike Anderson, EE&CS Dept., EE&CS Bldg.
Univ. of Michigan, Ann Arbor, MI 48109-2122

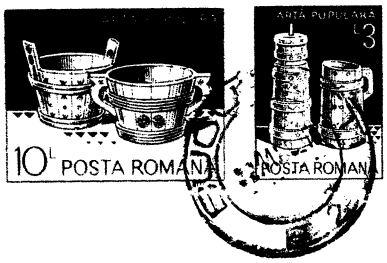
Sure can you send

TR CS-88-22

to Lin, thanks,

Dan

sent Jan. 4/89



PRIN AVION
PAR AVION AIR MAIL

Dr. Cornel Constantinescu

Polytehnic Institute
Department of Computers
Splaiul Independenței nr. 313
77206 BUCUREȘTI
ROMANIA

Dr. Peng Li

Department of Computer Science

University of Waterloo

Waterloo, Ontario N2L 3G1

Canada

Dear Sir,

I would highly appreciate receiving 1 reprint(s) of the following article (s) :

"Pattern matching in trees" 88-23

and of any other papers on related subjects.

I should be greatly indebted to you to be included on your mailing list.

sent with compliments
& added to m. list
Dec. 1/88

My current address:
P.O.Box 66-32,
Bucuresti 6,
ROMANIA

yours faithfully,

Constantinescu.

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP

Professor of Computation:
C.A.R. Hoare, FRS

Professor of Numerical Analysis:
K.W. Morton

8–11 Keble Road
Oxford OX1 3QD
Tel: Oxford (0865) 273837 (direct)

October 26, 1988

Research Reports Secretary
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
CANADA

Dear Sir/Madam

We would like to order one copy of each of the following reports:

CS–88–23 *Pattern matching in trees* by Peng Li. Price $5.00

CS–88–33 *On efficient entreeings* by Paul S. Amerins, Ricardo A. Baeza-Yates, Derick Wood. Price $2.00.

I enclose a cheque for $7.00.

Also enclosed is a list of our current publications for your interest. We are happy to implement exchange arrangements.

Yours sincerely

David Brown
Librarian

encl: Cheque for $7.00
List of PRG Technical Monographs to October 1988

Sent Nov. 17/88

NOV - 7 1988

DEPARTMENT OF HEALTH & HUMAN SERVICES

Public Health Service

Alcohol, Drug Abuse, and
Mental Health Administration
National Institute of Mental Health
Intramural Research Program
Bethesda MD 20892

10/31/88

Dept. of Computer Science
Univ. of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sirs:

Enclosed please find a check for $10 (ten dollars) for CS-88-31 and CS-88-23.

Thank-you

Tom Nosdahl, M.D.

13007 Magellan Ave.

Rockville, Md 20853

rec'd payment cheque # 422 + sent reports

NOV 7 1988

Printing Requisition / Graphic Services

19929

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION: Pattern Matching in Trees CS-88-23

DATE REQUISITIONED: May 20/88

DATE REQUIRED: ASAP

ACCOUNT NO.: 1 2 6 6 1 7 6 4 1

REQUISITIONER – PRINT: D. Wood

PHONE: 4456

SIGNING AUTHORITY: Sue DeAngelis / D. Wood

MAILING INFO – NAME: Sue DeAngelis

DEPT.: C.S.

BLDG. & ROOM NO.: MC 2314

[X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 90

NUMBER OF COPIES: 150

TYPE OF PAPER STOCK: [X] BOND [ ] NCR ____ PT. [X] COVER [ ] BRISTOL [X] SUPPLIED [ ] ____

PAPER SIZE: [X] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] ____

PAPER COLOUR: [X] WHITE [ ] ____

INK: [X] BLACK [ ] ____

PRINTING: [ ] 1 SIDE ____ PGS. [ ] 2 SIDES ____ PGS. * see note below

NUMBERING: FROM ____ TO ____

BINDING/FINISHING: 3 down left side

[X] COLLATING [X] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING:

CUTTING SIZE:

Special Instructions

*Please do the first pages 1-11 (in Roman numerals) single-sided and the rest of the report double-sided.

Math fronts and backs enclosed.

Thanks.

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

				D 0 1
				D 0 1
				D 0 1

TYPESETTING — QUANTITY | OPER. NO. | TIME | LABOUR CODE

P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1

PROOF

P R F				
P R F				
P R F				

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1

PMT

P M T				C 0 1
P M T				C 0 1
P M T				C 0 1

PLATES

P L T				P 0 1
P L T				P 0 1
P L T				P 0 1

STOCK

				0 0 1
				0 0 1
				0 0 1
				0 0 1

BINDERY

R N G				B 0 1
R N G				B 0 1
R N G				B 0 1
M I S 0 0 0 0 0				B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

GRAPHIC SERV. OCT. 85 482-2

To

Sue

From

Zoe

Date

May 19, 1988

memo

University of Waterloo

Would you please have the attached converted into the Math Faculty (fronts and backs) Technical Reports.

Derick would like 150 copies run off please. Also he would like the first pages - 1-11 (ones in Roman Numbers) single-sided the rest to be double-sided.

Thanks Sue.

p.s.

Use Derick's A/C : 126-6176-41

Pattern Matching in Trees

by

Peng Li

Data Structuring Group
Research Report CS-88-23
May 1988

Faculty of Mathematics

University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Pattern Matching in Trees

by

Peng Li

Data Structuring Group
Research Report CS-88-23
May 1988

Pattern Matching in Trees

by

Peng Li

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1988

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis examines a tree automata approach to tree matching problems. This approach is motivated by the finite automata approach which has been very successful in designing string matching algorithms. In particular, we show how the KMP algorithm can be generalized to give tree matching algorithms which preprocess the pattern tree. We also define structures for trees which are analogous to suffix tries and DAWGs for strings and show how they can be used for tree pattern matching. Additionally, we explore some other approaches to tree matching problems.

Acknowledgement

I would like to express my deepest gratitude to my thesis supervisor, Prof. Derick Wood, for his invaluable guidance and encouragement throughout the preparation of this thesis. I would also like to thank him for his friendly advice which have helped me to cope with the challenges both inside and outside the university.

I would also like to thank the readers of my thesis, Prof. Howard Johnson and Prof. Gordon Cormack, for reading this thesis under severe time constraint and suggesting modifications to the thesis.

Contents

List of Tables

List of Figures

Chapter 1

The Tree Matching Problem

1.1 Motivation

During the execution of equational and logic programs, subexpressions are replaced according to a set of replacement rules. The first step in such a replacement system is to find an appropriate subexpression of an expression which matches the left hand side of some replacement rule. Since expressions, or terms, are usually represented as term trees, this reduces to finding the occurrences of a given left hand side, or *pattern tree*, in the given expression, or *text tree*. In other words, this is a *tree pattern matching problem*. Furthermore, this is the specific problem we tackle in this thesis. Hoffmann and O'Donnell show in [13] how tree replacements may be used in automatically generated interpreters for nonprocedural programming languages. The elimination of redundant operations in code optimization in compiler design and the recognition of common subexpressions in simplifying algebraic expressions both involve tree pattern matching; see [3,20]. Tree pattern matching can also be used in context searching in tree structures, abstract data type specification and implementation [12], automatic theorem proving [17], and hierarchically structured databases applications.

In some applications, text trees have their subtrees replaced, so these trees change as

a result of the replacement; in other applications, text trees are essentially static. In this thesis, tree pattern matching is carried out on static text trees.

1.2 Overview of Previous Work

Karp, Miller and Rosenberg pioneered the investigation of the tree pattern matching problem in [15]. They designed a number of algorithms for the problem, and many of their ideas have been used in more recent algorithms.

Overmars and van Leeuwen presented in [22] a top-down algorithm (their best algorithm) for pattern matching in lexicographic trees. The idea underlying the algorithm is that finding a pattern in a text is equivalent to finding all its root-to-leaf paths as some substrings of root-to-leaf paths of the text tree such that all these substrings start at the same node of the text tree. The algorithm finds all occurrences of the root-to-leaf paths of the pattern in the text using a strategy similar to that of the KMP [18] and Aho-Corasick [2] algorithms; it records these occurrences in counters associated with the nodes of the text where these paths occur.

Hoffmann and O'Donnell thoroughly investigated the problem of matching term trees in [14]. They developed algorithms for both bottom-up and top-down approaches. Their top-down algorithm is essentially the same as the one in [22]. The basic operation of their bottom-up algorithm is that of a frontier-to-root tree automaton, which is described in Chapter 2. A number of people have modified their bottom-up algorithm to reduce the time and space complexity of its preprocessing phase; among them, Chase in [8] described a way of generating compressed tables.

Lang, Schimmler and Schmeck in [21] presented an algorithm for matching tree patterns sublinear on the average. The pattern and text trees in their algorithm are represented as ordered lists of *left paths*, which are the longest suffixes of root-to-leaf paths, where every node, except for the leaves, occurs in the same left path as its leftmost son. For example,

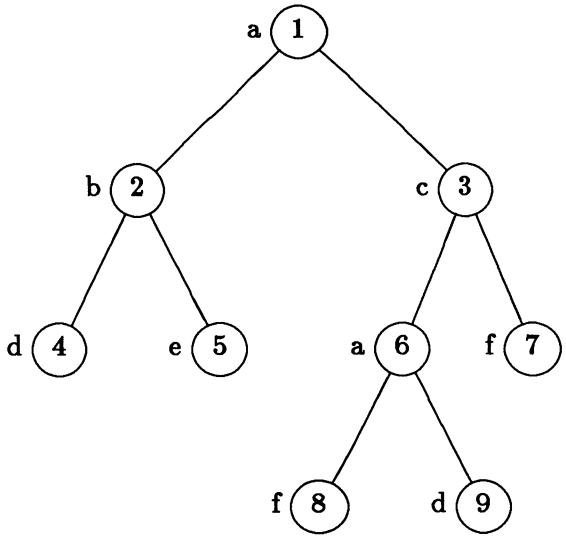


Figure 1.1: Example of a binary tree

the set of left paths of the tree in Figure 1.1 is {abd,e,caf,d,f}[1]. They used a combined approach of Boyer-Moore [6,18] and Aho-Corasick [2] for matching left paths and used the same counting technique found in the top-down algorithms of [22] and [14] to decide if the pattern occurs at a node in the text.

Steyaert and Flajolet did a mathematical analysis of the naive algorithm for tree pattern matching in [26]. They have shown that the expected time complexity of the naive algorithm is *linear* in the total size of the pattern tree and the text tree, although its worst time complexity is quadratic.

Kojima described in [19] a linear algorithm for matching a complete binary pattern tree in a binary text tree. The algorithm starts by checking if the pattern occurs at the root of the text, and when a mismatch is found, it splits the text tree into maximal subtrees such that it is possible for the pattern to match each subtree at its root. The algorithm proceeds recursively as if each new subtree is another text tree. This idea is analogous to sliding the pattern string along the text string in the KMP algorithm of [18].

Ramesh and Ramakrishnan in [23] and [24] explored a new approach to the problem.

[1]The trees in this thesis are drawn using the macros described in [7].

The text and pattern trees are represented as *euler strings*, which preserve the structure of the trees, and tree matching is then reduced to string matching with multiple patterns, for which the Aho-Corasick algorithm [2] is used. The euler string of a tree is obtained by traversing the tree in the order of "root → 1st subtree → root → ··· → root → last subtree → root" and listing the labels of the nodes as they are encountered. For example, the euler string of the tree in Figure 1.1 is "abdbebacafadacfca". The basis of their algorithm is that two term trees are equivalent if and only if their euler strings are equal.

1.3 Overview of the Thesis

The aim of this thesis is two-fold.

First, we use an automata-theoretic approach to provide a coherent and consistent framework for the study of tree pattern matching algorithms. This approach was successful for designing some well-known string matching algorithms such as the Knuth-Morris-Pratt algorithm [18]. It not only suggests a method of obtaining efficient algorithms, but it also provides a common framework within which to describe such algorithms. The automata-theoretic approach is implicit in much of the work on tree pattern matching algorithms, but it has never been the driving force. In chapter 3, we show that some well-known tree matching algorithms follow naturally from this approach.

Second, we explore "analogy" as an approach to give tree pattern matching algorithms from string matching algorithms. For example, in string matching, we can preprocess either the pattern or the text. This leads to the KMP and Boyer-Moore algorithms [18,6] in the former case, and to the DAWG and suffix trie approach in the latter case. All tree matching algorithms that have been considered up until now preprocess the pattern tree. We consider original methods for preprocessing text trees in Chapter 4, and these lead to the definitions of structures for trees analogous to suffix tries and DAWGs for strings. We also give new tree matching algorithms which make use of these structures.

Another analogy is based on the directions in which the texts and the patterns are scanned. For strings, the KMP and the Boyer-Moore algorithms scan the pattern in different directions, the former from left to right and the latter from right to left, while both algorithms move the pattern across the text from left to right. In the first section of Chapter 5, we will discuss how analogous tree matching algorithms may be designed.

There are other approaches to tree matching problems than those mentioned above. The second section of Chapter 5 suggests an approach based on transforming the original tree matching problem to a similar problem involving trees of smaller sizes by partitioning the pattern tree and the text tree.

Chapter 2 introduces the definitions and conventions used in subsequent chapters. The concluding chapter, Chapter 6, examines ways of solving tree matching problems involving patterns that are more general than the ones considered in earlier chapters, and we end the thesis by stating some conclusions and posing some open problems.

Chapter 2

Definitions and Conventions

2.1 Trees and Tree Matching Problems

2.1.1 Term Trees

Let Σ be a set of symbols; Σ is called an *alphabet* if each $\sigma \in \Sigma$ has a unique *degree* $\nu(\sigma)$ which is a nonnegative integer. Informally, a *term tree* over an alphabet Σ is a rooted, ordered tree whose nodes are labeled with elements in Σ; a node with label σ in a term tree has exactly $\nu(\sigma)$ children.

Given an alphabet Σ, define $\nu(\Sigma)=\max\{\nu(\sigma) \mid \sigma \in \Sigma\}$, $\Sigma_i=\{\sigma \in \Sigma \mid \nu(\sigma) = i\}$ and $\Sigma_+=\{\sigma \in \Sigma \mid \nu(\sigma) > 0\}$. A term tree over the alphabet $(\Sigma_2 \cup \Sigma_0)$ is called a *binary tree*, and a term tree over the alphabet $(\Sigma_1 \cup \Sigma_0)$ is called a *unary tree*. The set of all term trees over Σ is denoted by $\mathcal{T}_\Sigma$.

There is a natural one-to-one correspondence between the elements in the set of term trees over Σ and the elements in the set of Σ-*terms*. Σ-terms and the corresponding term trees are formally defined as follows.

Definition 2.1 *Let Σ be an alphabet and ν be the corresponding degree function.*

1. *If $\sigma \in \Sigma$ and $\nu(\sigma) = 0$, then σ is a Σ-term; the corresponding term tree consists of a single node labeled σ which is the root of the tree.*

2. *If $\sigma \in \Sigma$ and $\nu(\sigma) = q > 0$, then $\sigma(t_1, t_2, \ldots, t_q)$ is a Σ-term provided each of the t_is is a Σ-term; the root of the corresponding term tree is labeled σ and the i-th child of the root is the root of the term tree corresponding to t_i.*

3. *Nothing else is a Σ-term or a term tree over Σ.*

For example, let Σ be $\{a, b, c, d, e, f\}$, where $\nu(a) = \nu(b) = \nu(c) = 2$ and $\nu(d) = \nu(e) = \nu(f) = 0$; then, the tree of Figure 1.1 is a term tree over Σ, and the corresponding Σ-term is $a(b(d, e), c(a(f, d), f))$. An *edge* joins every internal node in a tree to each of its children.

Let T be a term tree. The set of all nodes in T is denoted by $N(T)$. $\mid N(T) \mid$ is called the *size* of T, which can also be written as $\mid T \mid$. The root of T is denoted by root(T).

Let v be a node in T. The label of v is denoted by $l(v)$. v is an *external* node (or a *leaf*) if $\nu(l(v)) = 0$; otherwise, it is an *internal* node. The set of all leaves of T is called the *frontier* of T and denoted by $F(T)$; the set of all internal nodes of T is denoted by $I(T)$.

If u is the i-th child of v, written as u=child(v, i), then v is called the *parent* of u, written as v=parent(u). Define $a^0(v) = v$ and $a^i(v)$=parent($a^{i-1}(v)$) for $i > 0$; if $u = a^i(v)$, then u is called the i-th ancestor of v and v is called the i-th descendant of u. Note that a node is both an ancestor and a descendant of itself. A node v is called a *proper descendant* of a node u if $u = a^i(v)$, for some $i > 0$. Two nodes are *cousins* of each other if neither of them is an ancestor of another. The set of all descendants of a node u, $\{v \mid u = a^i(v)$ for some nonnegative integer $i\}$, forms a *subtree* of T rooted at u. Note that every subtree of a term tree is itself a term tree. Define *depth*(v) to be k if root(T) = $a^k(v)$. The *height* of T, denoted by height(T), is defined to be max{depth(v) $\mid$ $v \in T$}. A term tree is *perfect* if all its leaves have the same depth.

To illustrate the above definitions, let T denote the term tree of Figure 1.1. Then, we have $N(T) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $\mid T \mid = 9$, root(T) = 1, $F(T) = \{4, 5, 8, 9, 7\}$, $I(T) =$

$\{1,2,3,6,\}$, $l(1) = l(6) = a$, $2 = \text{child}(1,1)$, $\text{parent}(2) = \text{parent}(3) = 1$, $1 = a^2(5)$, $\text{depth}(5) = \text{depth}(6) = 2$, $\text{height}(T) = 3$, the set of nodes {3,6,7,8,9} forms a subtree rooted at 3, and nodes 2 and 7 are cousins of each other.

Given a term tree T over Σ and a node v in T, an *rn-path* (standing for "root-to-node path") of v consists of all ancestors of v, that is, $\text{rn-path}(v)=\{u \mid u = a^i(v) \text{ for some } i \geq 0\}$. Let λ and $\circ$ denote the empty string and the concatenation operation respectively. The *rn-string* (standing for "root-to-node string") of v is defined as:

$$\text{rn-string}(v) = \begin{cases} \lambda & \text{if } v = \text{root}(T) \\ \text{rn-string}(\text{parent}(v)) \circ (l(\text{parent}(v)), i) & \text{if } v = \text{child}(\text{parent}(v), i) \end{cases}$$

Given the tree of Figure 1.1, rn-path(8)={1,3,6,8} and rn-string(8)=(a,2)(c,1)(a,1). If v is a leaf of P, the root-to-node path (string) of v is also called a root-to-leaf path (string).

In this thesis, defining a term tree is equivalent to defining the label function l, the child function, the father function and the root of the tree. In the drawings of trees, circles are used to represent nodes. Node identifiers are inside the circles, labels are to the left, and states of the nodes assigned by tree automata (described later in this chapter) are to the right.

2.1.2 Patterns and Tree Matching Problems

A *pattern tree* with respect to the trees in $\mathcal{T}_\Sigma$ is a term tree over $\Sigma_+ \cup \{\otimes\}$ such that all internal nodes of the tree are labeled with elements in Σ and all leaves are labeled with the special symbol $\otimes$, called "the don't-care symbol", which is not an element of Σ. We denote the set of all such pattern trees by $\mathcal{P}_\Sigma$. Trees in $\mathcal{T}_\Sigma$ are said to be *text trees* of trees in $\mathcal{P}_\Sigma$. The pattern tree with only one node, which must be labeled by $\otimes$, is called a *trivial pattern*; otherwise, the pattern tree has at least two nodes and is *nontrivial*. An example of a nontrivial pattern is given in Figure 3.2. Note that every subtree of a pattern tree is itself a pattern tree.

Definition 2.2 (Root-Match) *Let P be in $\mathcal{P}_\Sigma$ with root p and T be in $\mathcal{T}_\Sigma$ with root t.*

- *If $l(p) = \otimes$, then P* root-matches *T.*
- *If $l(p) \neq \otimes$, i.e. P consists of more than one node, then P* root-matches *T if $l(p) = l(t)$ and the subtree of P rooted at child(p, i) root-matches the subtree of T rooted at child(t, i) for all i, $1 \leq i \leq \nu(l(p))$.*

Definition 2.3 (Match) *A pattern tree P* matches *(or occurs in) a text tree T at a node v in T if P root-matches the subtree of T rooted at v.*

For example, the pattern tree corresponding to the Σ-term $a(\otimes, \otimes)$ matches the text tree of Figure 1.1 at nodes 1 and 6.

In this thesis, we will consider two kinds of tree matching problems.

Problem 1 *Given a text tree T and a pattern tree P, finds all nodes in T at which P matches T.*

Problem 2 *Given a text tree T and a pattern tree P, determine if there is a node v in T at which P matches T.*

Most of this thesis deals with Problem 1, except for part of Chapter 4 which deals with Problem 2.

Unless otherwise noted, all pattern trees in this thesis are assumed to be nontrivial. We will see in Chapter 6 how to solve pattern matching problems for patterns with non-$\otimes$ leaves.

Analogous to the tree matching problems, we will also encounter two string matching problems in this thesis.

Problem 3 *Given a text string t and a pattern string p, find all occurrences of p in t.*

Problem 4 *Given a text string t and a pattern string p, determine if there is an occurrence of p in t.*

2.2 Tree Automata

To solve the tree pattern matching problems defined in the previous section, a special class of tree automata, called *internal tree automata*, will be used. Internal tree automata differ from general tree automata in that the latter read labels on all nodes of the input tree while the former only reads the labels on the internal nodes of the input tree. Since only internal tree automata will be considered in this thesis, we will omit the word "internal" when we refer to such automata.

2.2.1 Tree Automata as Recognizers

We first define the two basic kinds of tree automata, root-to-frontier automata and frontier-to-root automata, and then introduce some of their variants which are more useful for our purposes.

Both the root-to-frontier and frontier-to-root tree automata operate on an input tree by scanning it synchronously level by level. During this process, an automaton assigns a state to each node in the input tree. (A deterministic finite string automaton acts in a similar fashion where the input may be considered to be a unary tree.) Obviously, a root-to-frontier automaton scans an input tree from the root to the frontier, and a frontier-to-root automaton does the reverse.

The state of a node v is also written as state(v). The tree formed by relabeling the nodes of the input tree with their respective states is called a *state tree*.

Given an alphabet Σ, we use I_Σ to denote the set of term trees whose internal nodes are labeled with elements in Σ. Note that a tree automaton only reads symbols of nonzero degrees.

Root-to-Frontier Automaton

Definition 2.4 *A (deterministic)* root-to-frontier *automaton $\mathcal{A}$ (RFA) with input alphabet*

Σ consists of a finite set S of states, a transition function $M : \Sigma \times S \to \cup_{k=1}^{\nu(\Sigma)} S^k$, an initial state $s_0 \in S$ and a set of final states $F \subseteq S$.

$\mathcal{A}$ takes as its input a term tree in I_Σ and inductively assigns states to the nodes of the input tree as follows:

1. *The state of the root is assigned to be s_0.*
2. *Given that the state of an internal node v is s and that the label of v is σ, the states of the children of v are $M(\sigma, s)$; that is, state(child(v,i)) is the i-th component of $M(\sigma, s)$, for $1 \leq i \leq \nu(\Sigma)$.*

The input tree is accepted *if the states of its leaves are all final states.*

Example 2.1 Suppose that $\mathcal{A}$ is an RFA with start state s_0 and that part of its transition function M is

$$M(a, s_0) = (s_1, s_2), \quad M(a, s_2) = (s_1, s_0), \quad M(b, s_0) = (s_2, s_2),$$
$$M(b, s_1) = (s_2, s_0), \quad M(b, s_2) = (s_1, s_1).$$

Scanning the tree of Figure 2.1 with $\mathcal{A}$ produces the state tree of Figure 2.2.

Frontier-to-Root Automaton

Definition 2.5 *A (deterministic)* frontier-to-root automaton *$\mathcal{B}$ (FRA) with input alphabet Σ consists of S,s_0,M and F, which are as in Definition 2.4 except that the function M is from $\cup_{k=1}^{\nu(\Sigma)}(\Sigma \times S^k)$ to S. The states are assigned to the nodes of an input tree in I_Σ inductively as follows:*

1. *The states of the leaves are all s_0.*
2. *Given that a node v is labeled σ and that s_i is the state of child(v,i), for $1 \leq i \leq \nu(\Sigma)$, the state of v is $M(\sigma, s_1, \cdots, s_{\nu(\Sigma)})$.*

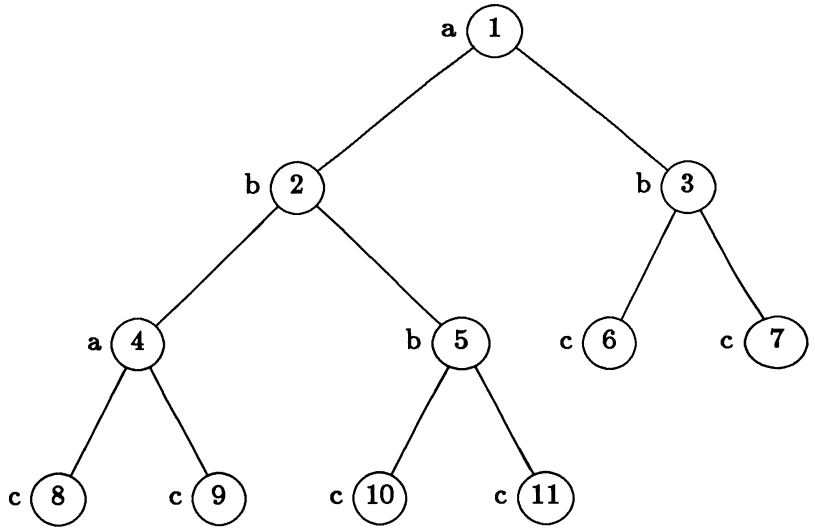


Figure 2.1: An input term tree

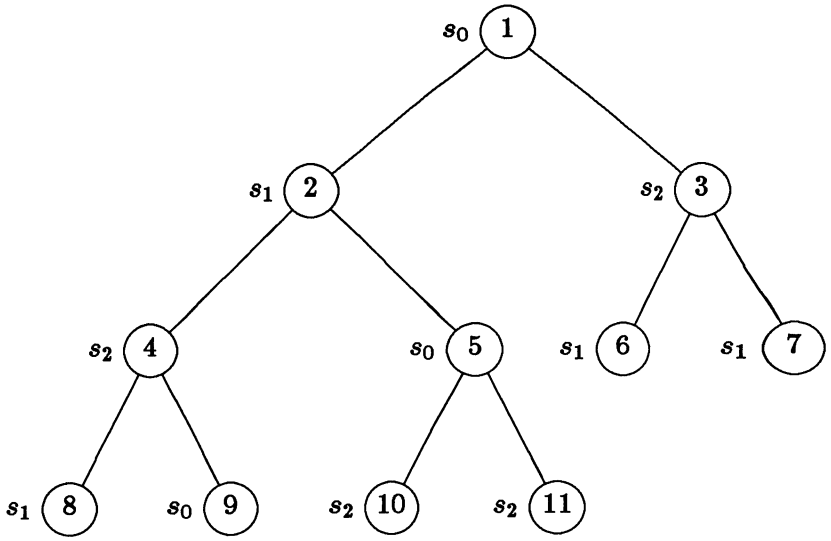


Figure 2.2: A state tree generated by an RFA

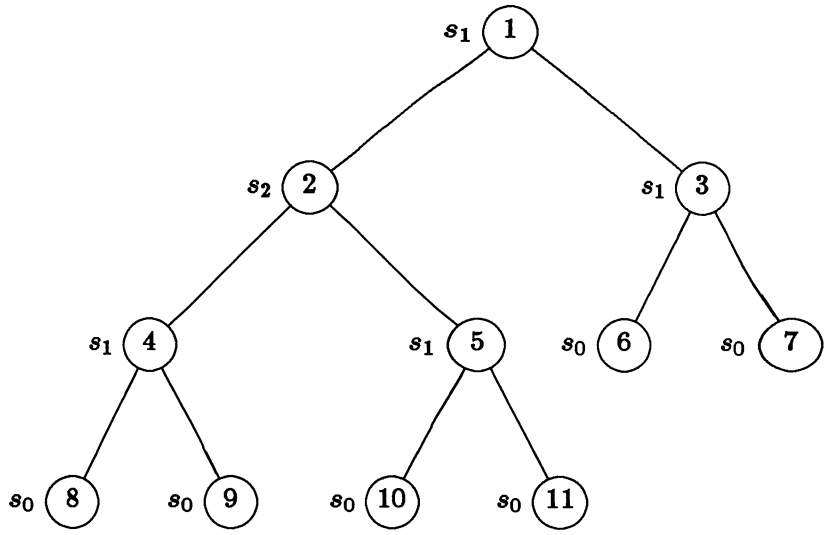


Figure 2.3: A state tree generated by an FRA

An input tree is accepted *if the state of its root is a final state.*

Example 2.2 Suppose that $\mathcal{B}$ is an FRA with start state s_0 and that part of its transition function M is

$$M(a, s_0, s_0) = s_1, \quad M(a, s_2, s_1) = s_1, \quad M(b, s_0, s_0) = s_1,$$
$$M(b, s_1, s_1) = s_2.$$

Scanning the tree of Figure 2.1 with $\mathcal{B}$ produces the state tree of Figure 2.3.

2.2.2 Variations of Tree Automata

The algorithms in the following two chapters will make use of the basic operations of the tree automata just defined. However, in order to solve the tree matching problem efficiently, these algorithms construct tree automata whose operations deviate slightly from the ones defined above. We consider the following two variations of tree automata.

- A *tree automaton with output* is a tree automaton which also produces output when processing[1] the nodes of the input tree, where the output produced depends on the state of the node being processed. For example, a pattern matching tree automaton may report a match whenever a final state is assigned to a node. Most of the tree automata we will see in this thesis are with output.

- A *writing tree automaton* is a tree automaton which also assigns values to the variables associated with the nodes of its input tree. For example, in order to determine all nodes which have i-th descendants, a writing tree automaton marks the i-th ancestor of every node it processes.

2.2.3 Implementing Tree Automata

Definition 2.6 *Given a term tree T, a* fringe *of T is any subset of $N(T)$ such that all nodes in the subset are cousins[2] of each other.*

A tree automaton is a parallel machine; during its operation, the nodes it is processing always form a fringe of its input tree. But given a tree automaton, the state of each node of its input tree is uniquely determined. It is in this sense that we claim that the operation of a tree automaton can be simulated by a sequential process. We now give sequential algorithms which implement the two basic kinds of tree automata.

Algorithm 2.1 *Given a root-to-frontier tree automaton $\mathcal{A}$ consisting of S, s_0, M and F, if T is an input tree of $\mathcal{A}$, this algorithm simulates the operation of $\mathcal{A}$ scanning T.*

1. *The state of root(T) is assigned s_0.*

2. *Invoke the procedure RF(root(T)), which is defined in Figure 2.4.*

3. *If the states of the leaves of T are all final, then T is accepted; otherwise, T is rejected.*

```
procedure RF(v:node);
var i: integer;
begin
    if ((v ∈ I(T)) and (state(v) = s)) then
    begin
        (state(child(v,1)),···,state(child(v,ν(l(v))))) ← M(l(v),s);
        for i from 1 to ν(l(v)) step 1 do
            RF(child(v,i));
    end;
end.
```

Figure 2.4: The procedure RF used in simulating an RFA

Algorithm 2.2 *Given a frontier-to-root tree automaton $\mathcal{B}$ consisting of S, s_0, M and F, and an input tree T, this algorithm simulates the operation of $\mathcal{B}$ scanning T.*

1. *Invoke the procedure FR(root(T)), which is defined Figure 2.5.*
2. *If the state of root(T) is final, then T is accepted; otherwise, T is rejected.*

Note that both Algorithms 2.1 and 2.2 traverse the input tree.

2.3 Model of Computation

The time complexities of the algorithms we will examine are based on the number of operations of the RAM model of computation described in [4].

[1]Processing a node means assigning a state to the node.

[2]Recall that two nodes are cousins if neither is an ancestor of the other.

```
procedure FR(v:node);
var i: integer;
begin
    if (v ∈ F(T)) then
        state(v) ← s_0;
    else
    begin
        for i from 1 to ν(l(v)) step 1 do
            FR(child(v, i));
        state(v) ← M(l(v),state(child(v,1)),···,state(child(v, ν(l(v))))));
    end;
end.
```

Figure 2.5: The procedure *FR* used in simulating an FRA

For a tree matching algorithm, we consider two types of operations: simple assignments and node comparisons. A node comparison involves determining the label of a node or comparing the labels of two nodes. A simple assignment is an assignment statement in which the right hand side is a constant or a variable. In every algorithm we will discuss, there are at least as many simple assignments as node comparisons, so the complexity of the algorithm depends entirely on the number of simple assignments. In some algorithms, the number of simple assignments and the number of node comparisons are not linearly related; in such cases we will also give the complexity based on the number of node comparisons.

The complexity of a tree matching algorithm is usually expressed as a function of the sizes of the text tree and the pattern tree. However, occasionally it is expressed as a function of other attributes of the trees as well as their sizes; for example, the complexity may depend on how balanced the trees are. This is done to indicate that the algorithm is more efficient when these attributes are taken into account; for example, Algorithm 3.1 runs in linear time for perfect pattern trees, although its time complexity for general pattern trees is quadratic in the sizes of the pattern and the text. The sizes of the alphabets over which the term trees are defined are treated as constants.

Tree matching algorithms have been designed for different representations of trees. Some assume that the trees involved cannot be changed, while others allow modification to the trees. Some represent trees using child and parent pointers, others represent them implicitly in arrays, and still others use special tree representations to suit their algorithms. Hence, care must be taken when we compare two tree pattern matching algorithms to ensure a "fair" comparison. In this thesis, all trees are given as doubly linked structures using child and parent pointers.

Chapter 3

Preprocessing Pattern Trees

For string matching, there are basically two classes of algorithms: one class preprocesses the pattern, and the other preprocesses the text. For tree matching, we may also either preprocess the pattern or preprocess the text.[1] When the text tree is unstable or when the pattern tree is to be matched with many text trees, preprocessing the pattern is more appropriate. For example, in some term rewriting systems, the text tree often changes due to the replacements of the left hand sides of the replacement rules by their right hand sides. In this setting, the patterns are used to match many different text trees; hence, it is desirable to preprocess the pattern rather than to preprocess the text. In this chapter, the pattern trees are preprocessed to give appropriate tree automata that process the text trees. Here, the underlying ideas of the KMP algorithm for string pattern matching are applied to tree pattern matching resulting in some well-known tree matching algorithms.

3.1 The KMP Algorithm

Knuth, Morris and Pratt [18] designed the first linear time algorithm that finds all occurrences of a given pattern string within a given text string. Their algorithm constructs a

[1] How to design algorithms which preprocess both the pattern and the text is an open problem.

(deterministic) finite automaton based on the pattern, with conditional λ-transitions, which is used to scan the text. The states of the automaton correspond to the positions in the pattern, from 1 to the length of the pattern[2], and 0, which is the initial state of the automaton. These states keep track of the maximal prefixes of the pattern which have been matched in scanning the text, and the automaton reports a match when it reaches the state standing for the last position of the pattern. The transition function M of the automaton has three kinds of transitions. The first kind corresponds to that when the character at position $i+1$ of the pattern matches the current text character σ, $M(\sigma, i) = i + 1$ is a transition. The second kind contains only transitions of the form $M(\sigma, 0) = 0$, for every symbol σ which is different from the first symbol of the string. The third kind are conditional λ-transitions, that are taken only if transitions of the first two kinds cannot be taken; each such λ-transition is defined as $M(\lambda, i) = next(i)$, where λ denotes the empty string and $next(i)$ is either the greatest j less than i such that pattern$[1, \ldots, j]$ = pattern$[i-j+1, \ldots, i]$ and pattern$[i+1]$ $\neq$ pattern$[j+1]$ or zero if no such j exists. The automaton can be constructed in time proportional to the length of the pattern, and scanning the text takes time proportional to the length of the text. Hence, the string pattern matching problem is solved in time proportional to the total length of the pattern and the text.

Example 3.1 Consider the pattern string $s = ababc$ and the text string $t = babcacababababcc$. The state set of the KMP automaton is {0,1,2,3,4,5}, where 0 is the initial state. The transitions of the automaton are:

$$M(a,0) = 1, \quad M(b,1) = 2, \quad M(a,2) = 3, \quad M(b,3) = 4,$$
$$M(c,4) = 5, \quad M(\neq a, 0) = 0, \quad M(\lambda,1) = 0, \quad M(\lambda,2) = 0,$$
$$M(\lambda,3) = 0, \quad M(\lambda,4) = 2, \quad M(\lambda,5) = 0.$$

Figure 3.1 shows the state diagram of this automaton. The result of scanning the text string t with this automaton is shown below.

[2]The *length* of a string s is the number of symbols in s and is denoted by $| s |$.

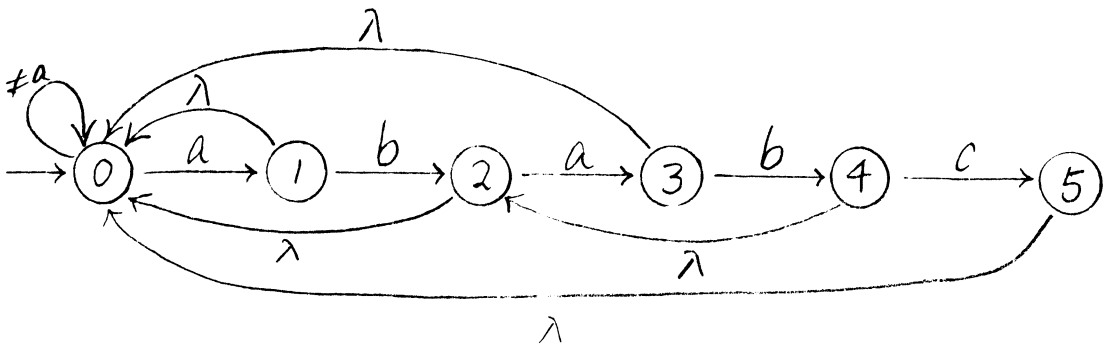


Figure 3.1: The state diagram of Example 3.1

positions:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
text:		*b*	*a*	*b*	*c*	*a*	*c*	*a*	*b*	*a*	*b*	*a*	*b*	*c*	*c*
states:	0	0	1	2	0	1	0	1	2	3	4	3	4	5	···
				0		0					2			0	

Hence, a match is found to start at position 9 and to end at position 13.

3.2 Generalizing the KMP Algorithm for Trees

The KMP algorithm constructs a finite automaton based on the pattern string to scan the text string. We can do something similar for tree pattern matching, but we have two different kinds of tree automata to consider, namely, frontier-to-root automata and root-to-frontier automata. A generalization of the KMP algorithm thus immediately gives us two tree pattern matching algorithms, namely, a bottom-up algorithm and a top-down algorithm, which are also described in [14] and [22].

We now compare these three algorithms and the corresponding automata,

- When a string is scanned by a finite automaton, the state of the automaton after reading the current character depends on the prefix of the input string ending at the current position. In the operation of a frontier-to-root automaton, the state of a node in the input tree depends on its descendents, which form the subtree rooted at the node. In the operation of a root-to-frontier automaton, the state of a node depends on its ancestors, which are in the root-to-node path of the node.

- The states of the finite automaton of the KMP algorithm keep track of the maximal prefixes of the pattern string which have been matched. The state of a node assigned by the frontier-to-root automaton of the bottom-up algorithm keeps track of all subtrees of the pattern tree which root-match the subtree rooted at that node. The state of a node assigned by the root-to-frontier automaton of the top-down algorithm keeps track of the maximum (in terms of its length) of all prefixes of root-to-leaf strings of the pattern tree which are equal to some suffixes of the root-to-node string ending at that node.

Note that two different pattern trees with the same number of nodes may match a text tree at the same node, while two different pattern strings with the same number of characters cannot match a text string at the same position. Furthermore, when several pattern strings match a text string at the same position, all the pattern string must be prefixes of the longest pattern string, but we can not make analogous claim for general term trees. This observation partly explains why the automaton in the bottom-up algorithm may have many more states than the automaton of the top-down algorithm.

The top-down algorithm reports a match at a node v if every root-to-leaf string of the pattern is equal to some substring of a root-to-leaf string of the text starting at v[3]. The

[3]This condition will be formally stated (again) in Chapter 4 as Proposition 4.6

bottom-up algorithm reports a match at a node v if the largest (in terms of its size) subtree of the pattern which root-matches the subtree rooted at v is the pattern itself.

In [22] and [14], the top-down algorithm is describes as one which uses a finite automaton. We will see in the next section that it can also be viewed as an algorithm which uses a root-to-frontier tree automaton.

3.3 Root-to-Frontier Approach

The top-down algorithm described in this section constructs a root-to-frontier writing automaton based on the pattern tree which then scans the text tree to find all matches.

3.3.1 The Automaton

Given a pattern tree $P \in \mathcal{P}_\Sigma$, let $\perp$ be a special symbol denoting a node not in $N(P)$ and, to simplify the description of the algorithm, we define child$(\perp, i)$=root(P), for $1 \leq i \leq \nu(\Sigma)$. Define functions $h : \Sigma_+ \times N(P) \rightarrow N(P) \cup \{\perp\}$ and $L : N(P) \rightarrow N(P) \cup \{\perp\}$ as follows.

$$h(\sigma, v) = \begin{cases} u & \text{if } u \text{ is the node in } P \text{ with the greatest depth}(u) \text{ such that} \\ & \text{rn-string}(u) \text{ is a proper suffix of rn-string}(v) \text{ and } l(u) = \sigma \\ \perp & \text{if no such } u \text{ exists} \end{cases} \tag{3.1}$$

$$L(v) = \begin{cases} u & \text{if } u \text{ is the leaf of } P \text{ with the greatest depth}(u) \\ & \text{such that rn-string}(u) \text{ is a proper suffix of rn-string}(v) \\ \perp & \text{if no such } u \text{ exists} \end{cases} \tag{3.2}$$

The function h acts like the *goto* function of the Aho-Corasick algorithm [2]; the function L is used by the writing operation of the tree automaton defined below.

Definition 3.1 *Given a pattern tree $P \in \mathcal{P}_\Sigma$ with root r, the Path Matching Machine for P, denoted by PMM(P), is a root-to-frontier writing automaton (with output) with input*

alphabet Σ_+, *state set* $N(P)$, *initial state* r, *the set of final states* $F(P)$; *its transition function* M *is defined as* $M(\sigma, v) = (g_1(\sigma, v), \cdots, g_{\nu(\sigma)}(\sigma, v))$ *where*

$$g_i(\sigma, v) = \left\{ \begin{array}{ll} child(v, i) & if\ l(v) = \sigma \\ child(h(\sigma, v), i) & otherwise \end{array} \right\} \text{ for each } i, 1 \leq i \leq \nu(\sigma). \tag{3.3}$$

When a text tree is scanned by $PMM(P)$, *if the state of a node* u *in the text is* s, *the automaton writes on the text tree according to the following procedure, assuming that there is a counter, initialized to 0, associated with every internal node of the text tree:*

if $s \in F(P)$
    increment the counter of the depth(s)-th ancestor of u by 1
endif;
$q \leftarrow s$;
while $(L(q) \neq \perp)$ **do**
    $q \leftarrow L(q)$;
    increment the counter of the depth(q)-th ancestor of u by 1
endwhile.

Note that the state set of PMM(P) is the set of all nodes in P, and that depth(s) for a state s is the depth of the node s in P and can be determined together with the depths of all other nodes in P by a single traversal of P. The counter associated with each internal node of the text tree keeps track of the number of root-to-leaf strings of the pattern which have been found to match root-to-leaf strings of that node in the text. If an occurrence of a root-to-leaf string of the pattern of length i is found to end at node v of the text tree, then the writing operation of the above automaton increments the counter of the i-th ancestor of v.

It is shown in [22] that the functions h and L can be computed in $O(|\ P\ |)$ time. The computation of the functions h and L is essentially a variation of the preprocessing phase

of the Aho-Corasick algorithm[4] [2].

The h-values can be computed level by level by the following equations, for all $\sigma \in \Sigma_+$.

$$h(\sigma, v) = \begin{cases} \perp & \text{if } v = r \\ u & \text{if } v\text{=child}(w,\text{i}) \text{ and } u\text{=child}(h(l(w),w),\text{i}) \text{ and } l(u) = \sigma \\ h(\sigma, u) & \text{if } v\text{=child}(w,\text{i}) \text{ and } u\text{=child}(h(l(w),w),\text{i}) \text{ and } l(u) \neq \sigma \end{cases}$$

When the h-values are known, the L-values can also be computed level by level as follows.

$$L(\sigma, v) = \begin{cases} \perp & \text{if } v = r \\ u & \text{if } v\text{=child}(w,\text{i}) \text{ and } u\text{=child}(h(l(w),w),\text{i}) \text{ and } u \in F(P) \\ L(\sigma, u) & \text{if } v\text{=child}(w,\text{i}) \text{ and } u\text{=child}(h(l(w),w),\text{i}) \text{ and } u \notin F(P) \end{cases}$$

The transition function M can be computed easily once the function h is known. Thus, M can also be computed in $O(\mid P \mid)$ time.

3.3.2 The Matching Procedure

Algorithm 3.1 *Given pattern tree P and text tree T, this algorithm finds all occurrences of P in T.*

1. *Construct the automaton PMM(P).*
2. *Assign a counter to each node of T and initialize them to zero. Scan T with PMM(P) and perform the appropriate writing operations described in Definition 3.1.*
3. *Traverse T, checking the value of the counter at every node; if the counter's value at a node v is $\mid F(P) \mid$, report a match at v.*

Let the *suffix index* of P, denoted by $S(P)$, be the number of levels[5] on which leaves occur. [22] and [14] have shown that Algorithm 3.1 can be implemented to perform subtree

[4]The Aho-Corasick algorithm can be viewed as the KMP algorithm for multiple pattern strings without λ-transitions.

[5]A node v is on level n if depth(v)=n.

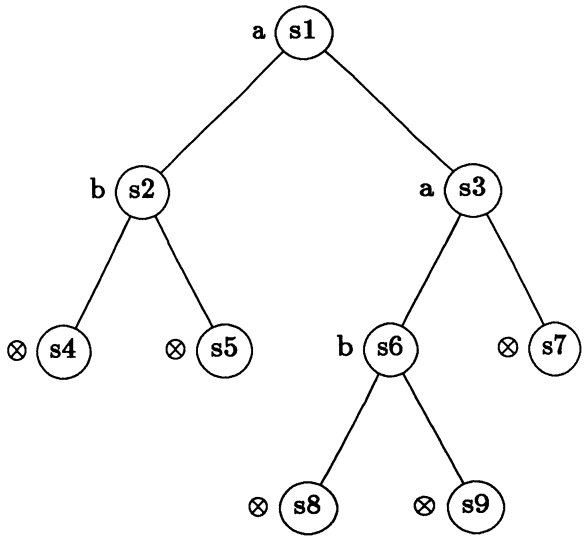


Figure 3.2: A sample pattern tree

matching in $O(\mid T \mid \times S(P) + \mid P \mid)$ time. The complexity of this algorithm based on the number of node comparisons is $O(\mid T \mid + \mid P \mid)$.

3.3.3 An Example

Let P denote the pattern tree of Figure 3.2 with the corresponding functions *child*, *parent* and *depth* defined in Table 3.1. The functions in Table 3.2 are used in defining PMM(P). The result of applying Algorithm 3.1 to a text tree is illustrated in Figure 3.3 and Table 3.3, and the algorithm reports the matches found at nodes 1 and 3.

3.4 Frontier-to-Root Approach

The bottom-up algorithm described in the section constructs a frontier-to-root automaton based on the pattern tree which scans the text tree to find all matches.

child(v,i)	1	2
s1	s2	s3
s2	s4	s5
s3	s6	s7
s6	s8	s9

v:	s2	s3	s4	s5	s6	s7	s8	s9
parent(v):	s1	s1	s2	s2	s3	s3	s6	s6

v:	s1	s2	s3	s4	s5	s6	s7	s8	s9
depth(v):	0	1	1	2	2	2	2	3	3

Table 3.1: Functions corresponding to the pattern in Figure 3.2

$h(\sigma,v)$	s1	s2	s3	s4	s5	s6	s7	s8	s9
a	$\perp$	s1	s1	s1	s1	s1	s3	s1	s1
b	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	s2	$\perp$	$\perp$	$\perp$

v:	s1	s2	s3	s4	s5	s6	s7	s8	s9
$L(v)$:	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	s4	s5

$g_1(\sigma,v)$	s1	s2	s3	s4	s5	s6	s7	s8	s9
a	s2	s2	s6	s2	s2	s2	s6	s2	s2
b	s1	s4	s2	s1	s1	s8	s1	s1	s1

$g_2(\sigma,v)$	s1	s2	s3	s4	s5	s6	s7	s8	s9
a	s3	s3	s7	s3	s3	s3	s7	s3	s3
b	s1	s5	s1	s1	s1	s9	s1	s1	s1

Table 3.2: The functions used by Algorithm 3.1

node:	1	2	3	4	5	6	7	the rest
counter value:	5	0	5	0	2	0	2	all 0

Table 3.3: Counter values corresponding to the tree in Figure 3.3

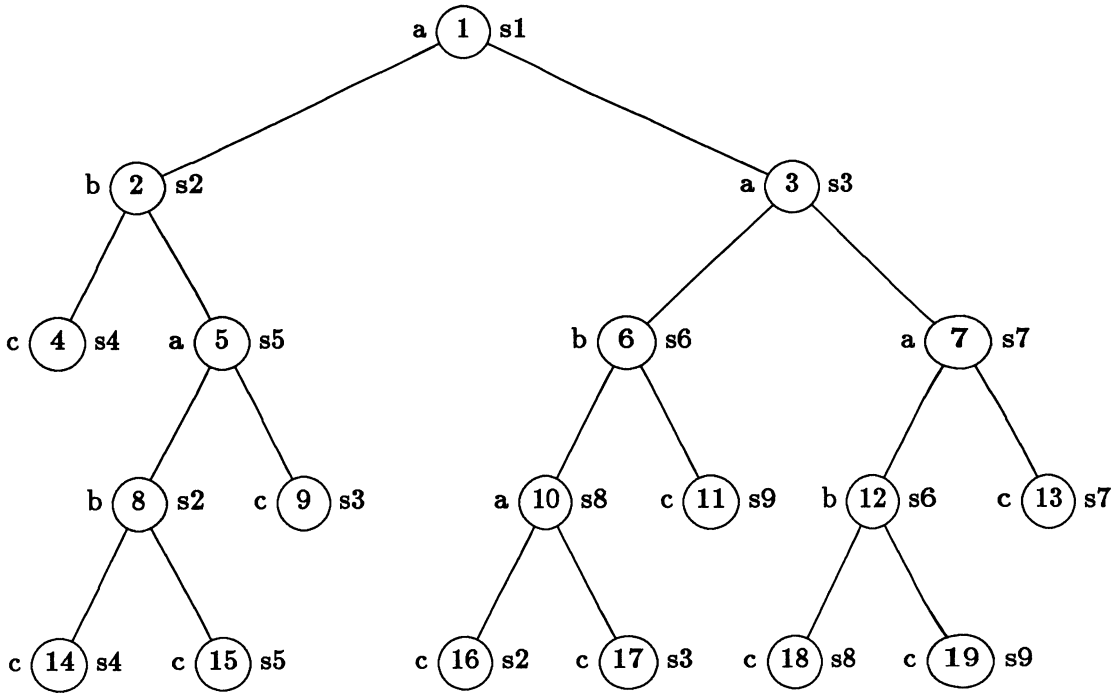


Figure 3.3: State tree resulting from Algorithm 3.1

3.4.1 The Automaton

Definition 3.2 (Match Set) *Given a pattern tree $P \in \mathcal{P}_\Sigma$, let S_P be the set of all subtrees of P. A subset Q of S_P is a* match-set *of P if there exists a tree t in $\mathcal{T}_\Sigma$ such that every pattern in Q root-matches t and every pattern in $S_P - Q$ does not root-match t.*

In other words, a match-set of P is a set of subtrees of P for which there exists a term tree t which is root-matched by all subtrees in the set but by no other subtrees of P. In particular, the set of all subtrees rooted at the leaves of a pattern tree is a match-set since given any text tree consisting of a single node, all subtrees in the set root-match it but no other subtrees of the pattern do. Let each subtree of P be represented by its root. Then, each match-set of P can be represented by a subset of $N(P)$. Let $S(P)$ denote the set of all match-sets of P, represented by subsets of $N(P)$, and let $X(P)$ $(\in S(P))$ denote the match-set consisting of all leaves of P.

Definition 3.3 *Given a pattern tree $P \in \mathcal{P}_\Sigma$ rooted at r, the Subtree Matching Machine P, denoted by SMM(P), is a frontier-to-root automaton with input alphabet Σ_+, state set $S(P)$, initial state $X(P)$, the set of final states $\{Q \mid r \in Q\}$, and the transition function $M : \cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times S(P)^k) \rightarrow S(P)$ defined by:*

$$M(\sigma, Q_1, \cdots, Q_{\nu(\sigma)}) =$$
$$X(P) \cup \{p \in N(P) \mid l(p) = \sigma \text{ and } child(p,i) \in Q_i \text{ for } 1 \leq i \leq \nu(\sigma)\}.$$

In principle, given a pattern tree $P \in \mathcal{P}_\Sigma$, the match-sets of P and the transition function of SMM(P) may be generated by a closure strategy which starts with the match-set $X(P)$ and repeatedly applies the following operations for every symbol σ in Σ_+:

$$Q(\sigma) \leftarrow X(P) \cup \{p \in N(P) \mid l(p) = \sigma \text{ and } child(p,i) \in Q_i \text{ for } 1 \leq i \leq \nu(\sigma)\}.$$

$$M(\sigma, Q_1, \cdots, Q_{\nu(\sigma)}) \leftarrow Q(\sigma)$$

where the Q_i's, $1 \leq i \leq \nu(\sigma)$, have already been generated. This generation algorithm requires $O(\mid S(P) \mid^{\nu(\Sigma)+1} \times \mid P \mid \times \mid \Sigma \mid)$ time, and the table size is $O(\mid S(P) \mid^{\nu(\Sigma)} \times \mid \Sigma \mid)$. The value of $\mid S(P) \mid$ is bounded from above by $2^{|P|}$.

3.4.2 The Matching Procedure

Here is the bottom-up tree matching algorithm.

Algorithm 3.2 *Given a pattern tree P and a text tree T, this algorithm finds all occurrences of P in T.*

1. *Construct the automaton SMM(P).*
2. *Scan T with SMM(P) in such a way that whenever a final state is assigned to a node, report a match found at that node.*

It is clear that the step 2 of Algorithm 3.2 can be accomplished in $O(\mid T \mid)$ time. So, the entire algorithm can be implemented to perform tree pattern matching in $O(\mid T \mid + \mid S(P) \mid^{\nu(\Sigma)+1} \times \mid P \mid)$ time. Its time complexity based on the number of node comparisons is linear in the total size of the text and the pattern.

Algorithm 3.2 first appeared in [14] in a very similar setting. The authors also noted that if a bit string of length height(P) can be associated with every node of the text tree and if intersections and unions of bit strings can be performed in constant time, Algorithm 3.1 and Algorithm 3.2 can be combined to give an algorithm whose time complexity is $O(\mid T \mid + \mid P \mid)$.

3.4.3 An Example

Let P be the pattern tree of Figure 3.2. The states of SMM(P) are encoded as follows:

$$s_0 = X(P) = \{\text{s4,s5,s7,s8,s9}\}$$

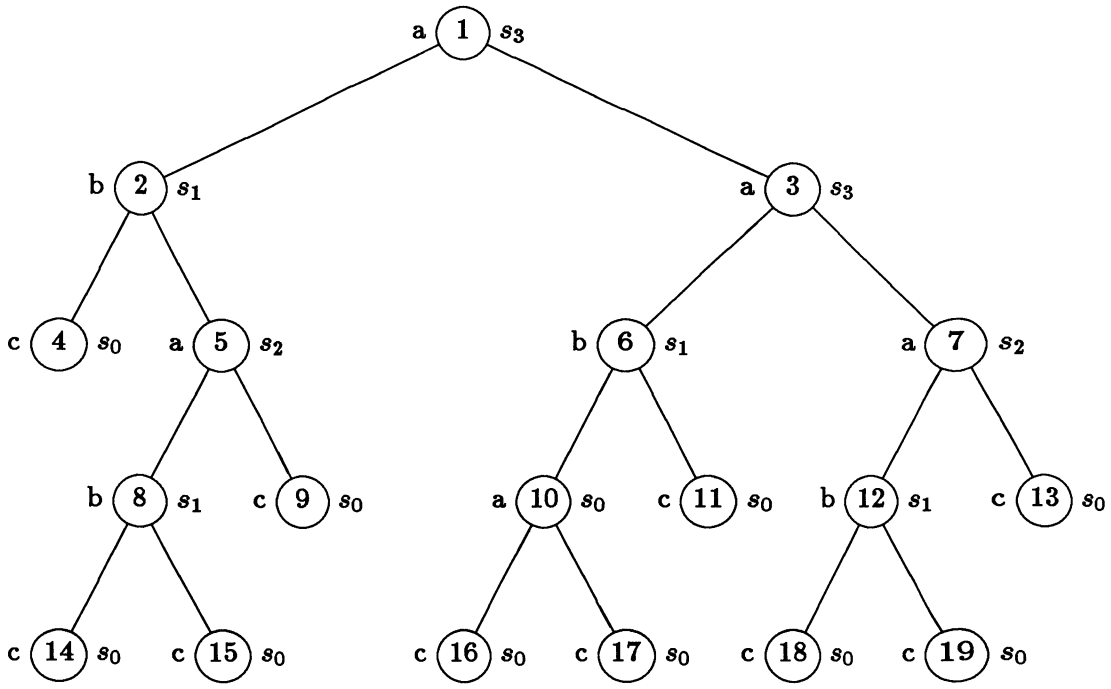


Figure 3.4: State tree resulting from Algorithm 3.2

$$s_1 = \{\text{s2,s6}\} \cup X(P)$$

$$s_2 = \{\text{s3}\} \cup X(P)$$

$$s_3 = \{\text{s1,s3}\} \cup X(P)$$

The set of final states of SMM(P) is $\{s_3\}$. The transition function of SMM(P) is given in Table 3.4. The result of applying Algorithm 3.2 to a text tree is illustrated in Figure 3.4; the algorithm reports the matches found at nodes 1 and 3.

$M(a, Q_1, Q_2)$	s_0	s_1	s_2	s_3
s_0	s_0	s_0	s_0	s_0
s_1	s_2	s_2	s_3	s_3
s_2	s_0	s_0	s_0	s_0
s_3	s_0	s_0	s_0	s_0

$M(b, Q_1, Q_2)$	s_0	s_1	s_2	s_3
s_0	s_1	s_1	s_1	s_1
s_1	s_1	s_1	s_1	s_1
s_2	s_1	s_1	s_1	s_1
s_3	s_1	s_1	s_1	s_1

Table 3.4: The transition function used by Algorithm 3.2

Chapter 4

Preprocessing Text Trees

For environments in which text strings rarely change but there are many pattern matching queries, string matching problems can been solved by preprocessing the text string. Position trees, suffix tries and DAWGs have been used for this purpose; see [4,5]. In this chapter, we generalize these ideas to tree pattern matching. In particular, the concepts of suffix tries, compact suffix tries and DAWGs for strings are generalized to give analogous structures for term trees. Such generalizations have not been examined in the literature.

4.1 Positions in Strings and Trees

We have implicitly used nodes to denote positions in a term tree. We now define positions of term trees formally together with the analogous notion for strings.

Definition 4.1 *A* position *in a string of length n is an integer between 1 and n inclusively. A* position *is a term tree is a node of the term tree.*

Each position in a string corresponds to a unique prefix of the string ending at the position and to a unique suffix starting at the position. Similarly, each node of a term tree

corresponds to the unique root-to-node path of the node and to a unique subtree rooted at the node. For a string, both its prefixes and suffixes are also strings. For a term tree, its subtrees are term trees, but, in general, its root-to-node paths are not.

Every two positions of a string uniquely identify the substring[1] between them. To make analogous claims for term trees, we need to define notions for term trees which are analogous to substring for strings.

Definition 4.2 *Given two nodes u and v in a term tree T such that $u = a^k(v)$ for some $k \geq 0$, the* ad-string *(standing for "ancestor-to-descendant string") from u to v is defined as:*

$$ad\text{-}string(u,v) = \begin{cases} \lambda & \text{if } k = 0 \\ ad\text{-}string(u,w) \circ (l(w), i) & \text{if } k > 0 \text{ and } v = child(w,i) \end{cases}$$

The string ad-string(u,v) is said to start at u and end at v.

In other words, the ad-string from u to v is the rn-string of v less its prefix which is the rn-string of u. Obviously, an rn-string is also an ad-string. In the tree of Figure 4.6, ad-string(2,10)=(b,2)(a,1). Note that nodes u and v uniquely identify the ad-string from u to v provided that u is an ancestor of v. Note that an ad-string is a string in $(\cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times \{1, 2, \cdots, k\}))^*$.

Definition 4.3 *Given a node u in a term tree T, and a nonempty subset S of $N(T)$ containing only descendants of u, S is* connected *at u if $u \in S$ and for every node v in S all nodes between[2] u and v are also in S.*

Definition 4.4 *A* partree *(standing for "part of a tree") rooted at u is defined to be either the empty set or a subset of $N(T)$ connected at u. (In this case, u is called the* root *of the partree.) The empty partree is denoted by μ.*

[1] To be formal, given a string $s = a_1 a_2 \cdots a_n$, a substring of s between i and j is the string $a_i a_{i+1} \cdots a_j$ where $1 \leq i \leq j \leq n$.

[2] A node w is *between* u and v if $u = a^i(w)$ and $w = a^j(v)$, for some $i, j > 0$.

In other words, a partree rooted at a node u is the subtree rooted at u less zero or more subtrees rooted at descendants of u. For example, considering the tree of Figure 4.6, each of the sets {2,4,8}, {2,4,5,11} and the empty set forms a partree rooted at node 2. Note that a partree is not necessarily a term tree.

The *size of a partree* is the number of nodes in it. The *leaves* of a partree are the nodes which are in the partree but none of whose proper descendants is in the partree. A partree is uniquely determined by giving its root and all its leaves.

Given a nonempty partree Q rooted at u and a node v in Q, a *subpartree* of Q rooted at v consists of all descendants of v which are in Q. The only subpartree of the empty partree is the empty partree.

Definition 4.5 *Let T be a term tree and Q_1 and Q_2 be two partrees of T rooted at nodes v_1 and v_2, respectively. Q_1 and Q_2 are said to be* equal *if one of the following conditions holds:*

1. *Q_1 and Q_2 are both empty partrees;*
2. *Q_1 and Q_2 are nonempty, $l(v_1) = l(v_2)$ and the subpartree rooted at child(v_1, i) is equal to the subpartree rooted at child(v_2, i), for $1 \leq i \leq \nu(l(v_1))$.*

Let us compare the notions of substring, ad-string and partree.

- A substring is connected in the sense that for any two symbols in the substring, all symbols between them are also in the substring. In this sense, an ad-string is also connected. According to Definition 4.3, a partree is connected.
- Given a string s, a term tree T and a nonnegative integer n, let i be a position in s and v be a node in T. If s has a substring of length n starting (or ending) at i, the substring is unique; if T has an ad-string of length n ending at v, then it is also unique; but T may have many partrees of size n rooted at v and may have many ad-strings starting at v.

- The maximal substring (in terms of its length) of a string is the string itself. The maximal partree (in terms of its size) of a term tree is the term tree itself.

Given a string s and a set of positions S of s, there is a unique maximal substring (in terms of its length) of s which starts at every position in S. We can make a similar claim for partrees of a term tree, as follows.

Given a term tree T, there is an empty partree rooted at every node of T. Hence, given a subset S of $N(T)$, there exist sets of equal partrees where each set contains a partree rooted at v for every $v \in S$; call these sets *parsets* of S. We say that a parset of S is *maximal* if there is no parset of S containing partrees of larger size.

Proposition 4.1 *Given a term tree T and any nonempty subset S of $N(T)$, there is a unique maximal parset. (We will call any partree equal to the partrees contained in this parset a* maximal partree of S.*)*

Proof: It is obvious that S must have at least one maximal parset. Let R_1 and R_2 be two maximal parsets of S and assume that they are different. Given any two vertices u and v in S, let partrees P_1 and Q_1 be in R_1, and P_2 and Q_2 be in R_2 where P_1 and P_2 are rooted at u and Q_1 and Q_2 are rooted at v. Then, the partree $P_1 \cup P_2$ rooted at u is equal to the partree $Q_1 \cup Q_2$ rooted at v, so they are also contained in some parset of S. Since R_1 and R_2 are different, P_1 and P_2 are not equal, so $\mid P_1 \cup P_2 \mid > \mid P_i \mid$, $i = 1, 2$. This contradicts the assumption that P_1 and P_2 are contained in maximal parsets of S. Therefore, R_1 and R_2 cannot be different, that is, there is only one maximal parset of S. □

In particular, if $S = \{v\}$ for some node v in T, the only maximal partree of S is the subtree of T rooted at v.

There are other notions for trees which correspond to the notion of position for strings. For example, we can also use fringes introduced in Definition 2.6 as positions in term trees. We will say more about this in later chapters.

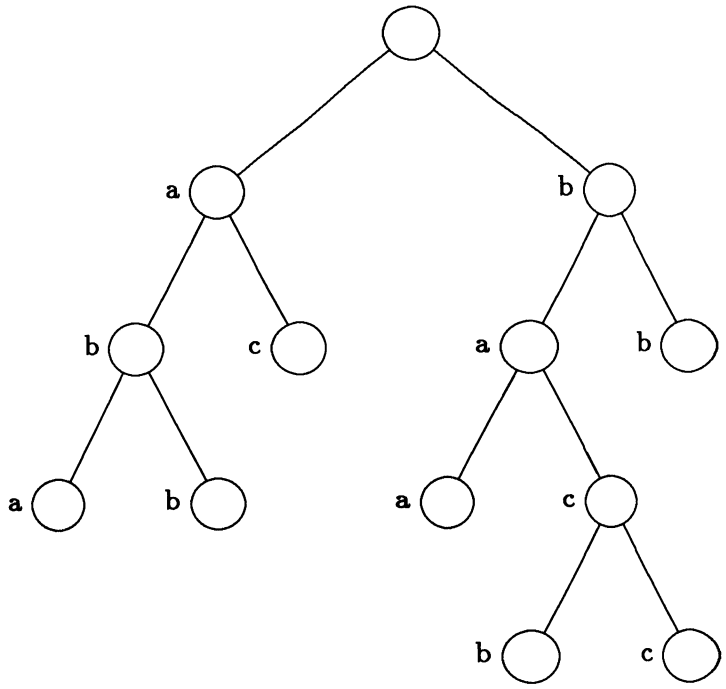


Figure 4.1: An example of a trie

4.2 Suffix Tries and DAWGs

4.2.1 Suffix Tries

Let us first review what suffix tries and compact suffix tries are.

Definition 4.6 *A* trie *is a rooted tree with labeled edges.*

Recall that an edge (u, v) is in a tree if u=child(v). The label on an edge (u, v) of a trie is denoted by label(u, v).

Example 4.1 The trie of the set of strings $\{aba, abb, ac, baa, bacb, bacc, bb\}$ is given in Figure 4.1. Note that in the drawing of the trie, the labels of the edges are put to the left of the vertices under them.

The number of vertices of a trie for strings $s_1, s_2, \cdots, s_m$ is bounded from above by $(\sum_{j=1}^{m} | s_j | + 1)$.

Given a vertex v in a trie I, the *string-label* of v, written as $\mathrm{sl}(v)$, is defined as:

$$\mathrm{sl}(v) = \begin{cases} \lambda & \text{if } v \text{ is the root of } I \\ \mathrm{sl}(u) \circ \mathrm{label}(u, v) & \text{if there is an edge } (u, v) \text{ in } I \end{cases}$$

In other words, $\mathrm{sl}(v)$ is the string formed by concatenating all labels of the edges on the path from the root of I to v.

Definition 4.7 *A node v in a trie of a set of strings* represents *the set of strings which have prefixes equal to sl(v).*

Each leaf v of a trie represents the original strings which are equal to $\mathrm{sl}(v)$; hence a trie of m strings has at most m leaves.

Definition 4.8 *The* suffix trie *of a string s is the trie of all suffixes of s.*[3]

Constructing a suffix trie for a string of length n requires $O(n^2)$ space and time.

Let I be the suffix trie of a string s, and denote each suffix of s by its starting position. According to Definition 4.8, each vertex v of I represents the positions (in s) each of which starts a suffix that has $\mathrm{sl}(v)$ as its prefix. Hence, suffix tries of text strings may be used to solve string matching problems.

A *chain* in a trie of a set of strings is a maximal sequence of edges and vertices of the form

$$< (v_0, v_1), v_1, (v_1, v_2), v_2, \cdots, v_n, (v_n, v_{n+1}) >$$

[3]Note that there is also a slightly different definition which defines the suffix trie of a string s as the trie of all suffixes of $s\$$.

where $n \geq 1$, each of the vertices $v_1, v_2, \cdots, v_n$ has exactly one outgoing edge and all v_i's represent the same set of strings. The *word* of such a chain is the string

$$\text{label}(v_0, v_1) \circ \text{label}(v_1, v_2) \circ \cdots \circ \text{label}(v_n, v_{n+1}).$$

Definition 4.9 *The* compact trie *of a trie I is obtained by replacing every chain in I by an edge labeled with the word of the replaced chain.*

Definition 4.10 *A* compact suffix trie *of a string s is the compact trie of the suffix trie of s.*

Proposition 4.2 *A compact suffix trie C of a string of length n has at most $2n-1$ vertices.*

Sketch of the Proof: C has at most n nonbranching nodes (nodes of degree less than 2), and hence at most $n-1$ branching nodes, giving a total of at most $2n-1$ nodes. □

4.2.2 The DAWG

If we only want to recognize all substrings of a given text string, that is, if we want to solve Problem 2, we can use smaller automata than the suffix trie of the string. The DAWG (Directed Acyclic Word Graph) of a string is one such automaton[4] [5]. The states of a DAWG correspond to a set of substrings of the text such that no two substrings in the set end at the same set of positions in the text and every substring of the text is a suffix of some substring in this set. Each substring in the state set stands for the set of positions in the text where this substring ends. For example, given the text string *abcbc*, the substring *bc* corresponds to a state in the DAWG of *abcbc*, and it stands for the positions 3 and 5. Since the substring *c* also ends at positions 3 and 5, it is said to be *end-equivalent* to *bc*, according to the following definition.

[4]Blumer et al. [5] also describe a class of even smaller automata.

Definition 4.11 *Let string s be $a_1 \ldots a_n (a_1, \ldots, a_n \in \Sigma)$. For any nonempty y in Σ^*, the* end-set *of y in s is defined by end-set$_s(y) = \{i : y = a_{i-|y|+1} \ldots a_i\}$. In particular, the end-set of the empty string λ is defined to be end-set$_s(\lambda) = \{0, 1, 2, \ldots, n\}$. Two strings x and y in Σ^* are* end-equivalent *(in s) if end-set$_s(x)$ = end-set$_s(y)$, and we denote this by $x \equiv_s y$. We denote by $[x]_s$ the equivalence class of x with respect to $\equiv_s$. The* degenerate class *is the equivalence class of the strings that are not substrings of s.*

Given a string s, there is a one-to-one correspondence between the end-sets of s and the equivalence classes of s with respect to $\equiv_s$ such that an end-set E corresponds to an equivalence class $[x]_s$ if and only if $E = \text{end-set}_s(x)$.

Definition 4.12 *The* DAWG *for s is a (partial) deterministic finite automaton with input alphabet Σ, state set $\{[x]_s \mid x$ is a substring of $s\}$, initial state $[\lambda]_s$, and transitions $\{([x]_s, a, [xa]_s) \mid x$ and xa are substrings of $s\}$; all states of the DAWG are final.*

The DAWG defined above recognizes the set of all substrings of s. So, it can be used to scan a given pattern string to check if the pattern occurs in the text string s.

Example 4.2 The transitions of the DAWG for the string $s = abcbc$ are:

$$([\lambda]_s, a, [a]_s), \quad ([a]_s, b, [ab]_s), \quad ([ab]_s, c, [abc]_s),$$
$$([abc]_s, b, [abcb]_s), \quad ([abcb]_s, c, [abcbc]_s), \quad ([\lambda]_s, b, [b]_s),$$
$$([\lambda]_s, c, [bc]_s), \quad ([b]_s, b, [bc]_s), \quad ([bc]_s, b, [abcb]_s).$$

The end-sets corresponding to the states of this DAWG are:

$$\text{end-set}_s(\lambda) = \{0, 1, 2, 3, 4, 5\}, \quad \text{end-set}_s(a) = \{1\},$$
$$\text{end-set}_s(b) = \{2, 4\}, \quad \text{end-set}_s(bc) = \{3, 5\},$$
$$\text{end-set}_s(ab) = \{2\}, \quad \text{end-set}_s(abc) = \{3\},$$
$$\text{end-set}_s(abcb) = \{4\}, \quad \text{end-set}_s(abcbc) = \{5\}.$$

The state diagram of this DAWG is given in Figure 4.2.

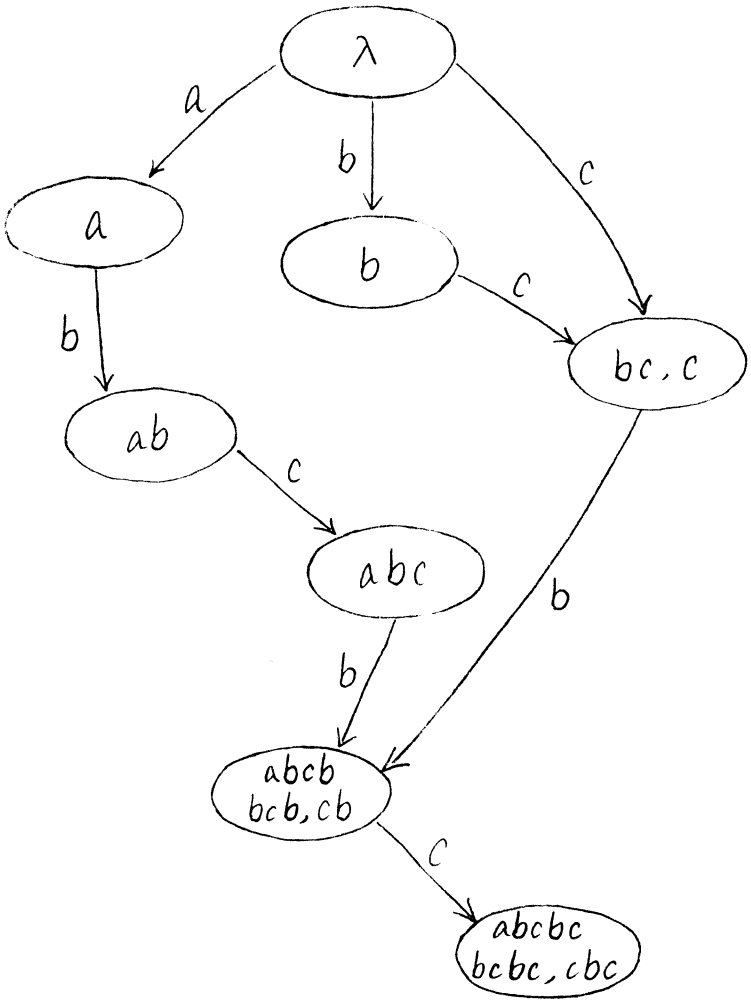


Figure 4.2: The DAWG of *abcbc*

Theorem 4.3 *Given a string s, let C be the compact suffix trie of s. Each vertex in C represents exactly one end-set of s, and each nonempty end-set of s is represented by exactly one vertex of C. Moreover, C has an edge (u, v), where u represents U and v represents V, if and only if $U \supset V$ and there is no end-set W of s such that $U \supset W \supset V$.*

Sketch of the Proof: Let I be the suffix trie of s. Since every vertex in I represents an end-set of s, every vertex in C also represents an end-set of s. Since every substring of s is equal to $\mathrm{sl}(v)$ for some vertex v in I, all nonempty end-sets of s are represented by vertices of I. A set represented by a vertex in I is also represented by a vertex in C, and any two vertices of C represent two different sets of positions, so each nonempty end-set of s is represented by exactly one vertex in C.

The second part of the theorem follows from Definition 4.10. □

Proposition 4.2 and Theorem 4.3 give us the following corollary.

Corollary 4.4 *A string of length n has at most $2n - 1$ nonempty end-sets.*

A string x is a *reverse string* of a string y, written as $x = y^{-1}$, if $| x | = | y |$ and $x[i] = y[| y | - i + 1]$ for $1 \leq i \leq | y |$.

The following theorem illustrates another connection between compact suffix tries and DAWGs.

Theorem 4.5 *Given a string s, let C be the compact suffix trie of s^{-1}, and denote the DAWG of s by D. D can be constructed from C as follows: The states of D are the vertices of C, and D has a transition (u, a, v) if and only if v represents end-set$_{s^{-1}}((sl(u))^{-1} \circ a)$.*

The proof of the above theorem is based on the definitions of end-sets, DAWGs and compact suffix tries, and is omitted here.

Example 4.3 The compact suffix trie of the string $(abcbc)^{-1}$=*cbcba* is given in Figure 4.3 where the end-sets are expressed in terms of the positions of *abcbc* to show the connection with Figure 4.2.

Blumer et al. [5] describe an algorithm which constructs the DAWG of a given text string in time linearly proportional to the length of the text string. This algorithm does not keep track of the end-sets, so it reports only whether a pattern occurs but not where it occurs. Since the time required to read a pattern string with a DAWG is clearly linear in the length of the pattern string, the DAWG can be used to determine whether the pattern occurs in the text in linear time; that is, the DAWG can be used to solve Problem 4 in linear time.

Note that given a text string t and a pattern string p, the suffix trie and the DAWG for t^{-1} can also be used for solving string matching problems involving p and t by scanning p^{-1} instead of p.

4.3 Generalizations Using Ad-Strings

4.3.1 Overview

Given a term tree T over Σ, an ad-string of T from u to v is an *nl-string* (standing for node-to-leaf string) if v is a leaf of T; a reverse string of an rn-string of v is called an *nr-string* (standing for node-to-root string) of v; and a reverse string of an nl-string of v is called an *ln-string* (standing for leaf-to-node string) of v. Listed below are some structures which result from generalizing the concepts introduced in the previous section based on nr-strings, nl-strings, rn-strings and ln-strings of T.

1. The trie for all nr-strings of T is called the *nr-trie* of T; it can be used to recognize all substrings of nr-strings of T.

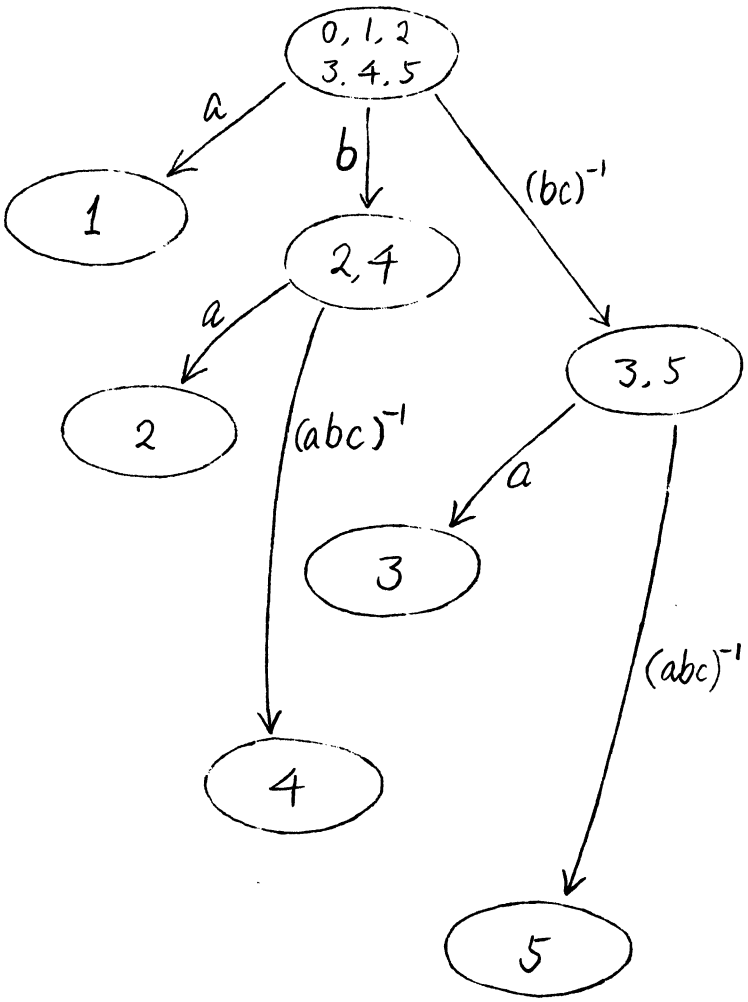


Figure 4.3: The compact suffix trie of $(abcbc)^{-1}$

2. The trie for all nl-strings of T is called the *nl-trie* of T; it can be used to recognize all substrings of nl-strings of T.

3. A *compact nr-trie* of T is the compact trie of the nr-trie of T, and a *compact nl-trie* of T is the compact trie of the nl-trie of T.

4. For any string x in $(\cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times \{1,2,\cdots,k\}))^*$, the *rn-set* of x in T is defined by $\text{rn-set}_T(x)=\{v \in N(T) \mid x$ is equal to a suffix of an rn-string ending at $v\}$; note that the rn-sets of T correspond to the end-sets of the rn-strings of T. Two strings x and y in $(\cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times \{1,2,\cdots,k\}))^*$ are *rn-equivalent* if $\text{rn-set}_T(x) = \text{rn-set}_T(y)$, and we denote this by $x \equiv_T^r y$. We denote by $[x]_T^r$ the equivalence class of x with respect to $\equiv_T^r$. Then, we can define the DAWG for the rn-strings of T, which we call *the rn-DAWG* of T, as a (partial) deterministic automaton with input alphabet $\cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times \{1,2,\cdots,k\})$, state set $\{[x]_T^r \mid x$ is a substring of some rn-string of $T\}$, initial state $[\lambda]_T^r$, and transitions $\{([x]_T^r, \sigma, [x\sigma]_T^r) \mid x$ and $x\sigma$ are substrings of some rn-strings of $T\}$. The rn-DAWG of T recognizes all substrings of the rn-strings of T.

5. For any string x in $(\cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times \{1,2,\cdots,k\}))^*$, the *ln-set* of x in T is defined by $\text{ln-set}_T(x)=\{v \in N(T) \mid x$ is equal to a suffix of an ln-string ending at $v\}$; note that the ln-sets of T correspond to the end-sets of the ln-strings of T. Two strings x and y in $(\cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times \{1,2,\cdots,k\}))^*$ are *ln-equivalent* if $\text{ln-set}_T(x) = \text{ln-set}_T(y)$, and we denote this by $x \equiv_T^l y$. We denote by $[x]_T^l$ the equivalence class of x with respect to $\equiv_T^l$. Then, we can define the DAWG for the ln-strings of T, which we call *the ln-DAWG* of T, as a (partial) deterministic automaton with input alphabet $\cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times \{1,2,\cdots,k\})$, state set $\{[x]_T^l \mid x$ is a substring of some ln-string of $T\}$, initial state $[\lambda]_T^l$, and transitions $\{([x]_T^l, \sigma, [x\sigma]_T^l) \mid x$ and $x\sigma$ are substrings of some ln-strings of $T\}$. The ln-DAWG of T recognizes all substrings of the ln-strings of T.

Since ad-strings and their reverses are also strings, much of the discussion in the previous section on suffix tries, compact suffix tries and DAWGs also applies to the structures just introduced.

In the following, we compare these structures for a given term tree T.

1. Since each node v of T gives rise to exactly one nr-string of length depth(v), the sum of the lengths of all nr-strings of T is

$$\sum_{v \in N(T)} \text{depth}(v),$$

which is an upper bound on the number of edges in the nr-trie of T. This upper bound is less than $| T |^2$.

2. Since each leaf of depth d gives rise to exactly d nonempty nl-strings, among which there is one string of length i for each integer i from 1 to d, the sum of the lengths of all nl-strings of T is thus

$$\sum_{v \in F(T)} \frac{1}{2}\text{depth}(v)(\text{depth}(v) + 1),$$

which is an upper bound on the number of edges in the nl-trie of T. This upper bound is less than $| T |^3$.

3. The compact nr-trie of T has at most $(2 \mid T \mid -1)$ vertices, and the compact nl-trie of T has at most $(2(\mid F(T) \mid \times \text{height}(T)) - 1)$, which is $O(\mid T \mid^2)$, vertices. The proof of this statement is similar to that of Proposition 4.2.

4. Let C be the compact nr-trie of T; then each vertex in C represents exactly one rn-set of T and each nonempty rn-set of T is represented by exactly one vertex in C. Similarly, if C is the compact nl-trie of T, then each vertex in C represents exactly one ln-set of T and each nonempty ln-set of T is represented by exactly one vertex in C. The proof of this statement is similar to that of Theorem 4.3.

5. The rn-DAWG of T has $O(\mid T \mid)$ states, and the ln-DAWG of T has $O(\mid T \mid^2)$ states.

Proposition 4.6 *Given a text tree T, a pattern tree P and a node v in T, if each of the root-to-leaf strings of P is equal to a prefix of an nl-string of v, then P matches T at v.*

Note that merely recognizing root-to-leaf strings of a pattern tree does not imply that the pattern occurs in the text; for solving either Problem 1 or Problem 2, we also need to know where these strings occur in the text in order to determine if the entire pattern occurs in the text.

Based on Proposition 4.6, we can derive from any of the above structures an algorithm for tree pattern matching. To illustrate this, we will discuss a tree matching algorithm based on the nl-trie of a given term tree in the next subsection.

4.3.2 A Root-to-Frontier Automaton Based on Text

We now present an algorithm which preprocesses the text tree to construct a root-to-frontier automaton which is used to scan pattern trees to solve tree matching problems.

Definition 4.13 *Given a term tree T over Σ and the nl-trie G of T, a (partial) root-to-frontier automaton based on G, denoted by $RN(T)$, is defined as follows.*

1. *The states of $RN(T)$ are the vertices of G; the initial state is the root of G, and all states of $RN(T)$ are final.*
2. *The transition function M of $RN(T)$ is defined by: $M(\sigma, q_0) = (q_1, \cdots, q_{\nu(\sigma)})$ if and only if (q_0, q_i) is an edge of G and label$(q_0, q_i) = (\sigma, i)$ for $1 \leq i \leq \nu(\Sigma)$.*

We also define a function S from the state set of $RN(T)$ to the power set of $N(T)$ as follows.

1. *For each state q of $RN(T)$, which is also a vertex of G, define $S(q)$ to be the set of nodes of T such that $v \in S(q)$ if and only if some nl-string in the set represented by q starts at v.*

Note that if the state of a node v in the input is q, the root-to-node string ending at v equals a prefix of the node-to-leaf string starting at w for every node w in $S(q)$.

A straightforward algorithm for constructing RN(T) and the associated S-function requires $O(\mid T \mid^4)$ time and space. The following algorithm uses the root-to-frontier automaton and its associated S-function so constructed to solve the tree matching problems.

Algorithm 4.1 *Given the RN(T) and its associated S-function for a text tree T, this algorithm finds all occurrences of a pattern tree P in T.*

1. *Scan P with RN(T).*
2. *If P is accepted, that is, all leaves of P are reached by RN(T), then compute the set*

$$L = \cap_{v \in F(T)}\{H(q) \mid q \text{ is the state of } v\},$$

and report matches found at all nodes in L. (If L is empty, no match is reported.) If P is not accepted, there is no match of P in T.

The time complexity of Algorithm 4.1 is $O(\mid P \mid \times \mid T \mid)$ since Step 1 of the algorithm takes $O(\mid P \mid)$ time and Step 2 requires $O(\mid P \mid \times \mid T \mid)$ time.

4.4 Generalizations Using Partrees

Since we have defined that a pattern tree has all its leaves labeled with $\otimes$, we use a variation of the partree introduced earlier to generalize the DAWG.

Definition 4.14 *Given a term tree T, an* internal partree *of T rooted at v is defined to be either the empty set or a subset of $I(T)$ connected at v.*

In other words, an internal partree of T is a partree T which contains no leaves of T. For example, considering the tree of Figure 4.6, each of the sets {2,4}, {2,4,5,11} and the empty set is an internal partree rooted at the node 2.

Since an internal partree is a partree, all the notions defined earlier for partrees also apply to internal partrees.

Definition 4.15 *Given an internal partree Q of a term tree T, the* end-set *of Q in T is defined by end-set$_T(Q) = \{v \mid$ there is an internal partree rooted at v which is equal to $Q\}$. In particular, end-set$_T(\mu) = N(T)$. Two internal partrees Q_1 and Q_2 of T are* end-equivalent, *denoted by $Q_1 \equiv_T Q_2$, if end-set$_T(Q_1)$ = end-set$_T(Q_2)$. We denote by $[Q]_T$ the equivalence class of Q with respect to $\equiv_T$.*

Given a term tree T, there is a one-to-one correspondence between the end-sets of T and the equivalence classes of T with respect to $\equiv_T$ such that an end-set E corresponds to an equivalence class $[Q]_T$ if and only if $E =$ end-set$_T(Q)$. We now define the analog of DAWG for trees, which we call TDAWG.

Definition 4.16 *The* TDAWG *for a term tree T over Σ is a (partial) frontier-to-root automaton with input alphabet Σ_+, state set $S = \{[Q]_T \mid Q$ is an internal partree of $T\}$, and initial state $[\mu]_T$. Its transition function $M : \cup_{k=1}^{\nu(\Sigma)}(\Sigma_k \times S^k) \rightarrow S$ is defined by:*

$$M(\sigma, [Q_1]_T, \cdots, [Q_{\nu(\sigma)}]_T) = \{[Q]_T \mid \begin{array}{l} [Q]_T \text{ contains an internal partree } R \text{ rooted} \\ \text{at a node } v \text{ such that } l(v) = \sigma \text{ and} \\ \text{the subpartree of } R \text{ rooted at child}(v,i) \text{ is} \\ \text{in } [Q_i]_T \text{ for } 1 \leq i \leq \nu(\sigma)\}. \end{array}$$

All states of the TDAWG are final.

Note that each state of the TDAWG corresponds to an end-set of T, and vice versa.

The TDAWG of T defines the set of term trees $\{P \mid$ the partree $I(P)$ of P is equal to some partree of $T\}$; in particular, the TDAWG of T recognizes all pattern trees which match T, so it can be used to check if a pattern tree occurs in T, that is, it can solve Problem 2.

Example 4.4 Given the term tree T in Figure 4.4, its TDAWG D is defined as follows:

1. The state set of D is $\{s_0, s_1, s_2, s_3, s_4\}$ where

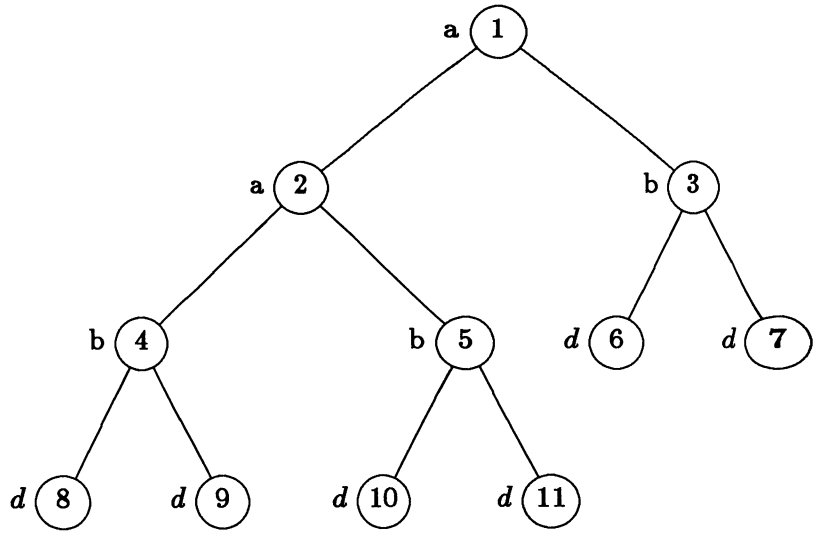


Figure 4.4: The term tree in Example 4.4

$s_0 = [\mu]_T$ which corresponds to the end-set $N(T)$,

$s_1 = [\{3\}]_T$ which corresponds to the end-set {3,4,5},

$s_2 = [\{1,3\}]_T$ which corresponds to the end-set {1,2},

$s_3 = [\{2,4,5\}]_T$ which corresponds to the end-set {2},

$s_4 = [\{1,2,3,4,5\}]_T$ which corresponds to the end-set {1}.

2. The transition function M of D is:

$$M(a,s_0,s_0) = s_2, \quad M(b,s_0,s_0) = s_1, \quad M(a,s_2,s_0) = s_4,$$
$$M(a,s_0,s_1) = s_2, \quad M(a,s_1,s_0) = s_3, \quad M(a,s_3,s_0) = s_4,$$
$$M(a,s_2,s_1) = s_4, \quad M(a,s_1,s_1) = s_3, \quad M(a,s_3,s_1) = s_4.$$

3. The initial state is s_0, and the set of final states is $\{s_0,s_1,s_2,s_3,s_4\}$.

We can also define constructively the analog of compact suffix trie for trees, which we will call CPDAG (Compact Position Directed Acyclic Graph).

Definition 4.17 *Given a term tree T, the CPDAG of T, denoted by $CPDAG(T)$, is the directed acyclic graph*[5] *defined as follows:*

1. *For each end-set E of T, create a vertex v of $CPDAG(T)$. We say that v* represents *E.*

2. *Given two vertices u and v in $CPDAG(T)$ which represent U and V respectively, (u, v) is an edge of $CPDAG(T)$ if $U \supset V$ and $CPDAG(T)$ does not have a vertex w representing W such that $U \supset W \supset V$.*

Notice the similarity between this definition and Theorem 4.3.

We will not define the structure analogous to suffix tries whose compact version is the CPDAG, because such a definition is rather tedious and it is unclear how this analog of the suffix trie can help us to solve the tree matching problem. We can also define analogously a structure similar to a suffix trie based on the euler strings of the subtrees of a given term tree; how this structure may be used for tree pattern matching is still under investigation.

Constructing the TDAWG for a term tree may be infeasible in general due to the large number of states and transitions it may have. Since a TDAWG is a frontier-to-root tree automaton with no unreachable states[6], each state of a TDAWG, except the initial state, must have at least one incoming transition, so the number of transitions of a TDAWG is at least the number of its states less one. Consider the TDAWG of a given term tree T. The number of states in the TDAWG is equal to the number of end-sets of T. Since each such end-set of T, except the end-set $N(T)$, is a nonempty subset of $I(T)$ and there are

[5] If the edges of the CPDAG are to be labeled, the label on an edge (u, v), where u represents U and v represents V, should reflect the difference between the maximal partrees of U less their leaves and the maximal partrees of V less their leaves.

[6] An *unreachable state* of a tree automaton is a state which can never be reached by the tree automaton no matter what the input is. If s is a state in a TDAWG representing an end-set E, then s can be reached when the input to the TDAWG is a term tree whose internal nodes form a partree which is equal to the partree formed by all the internal nodes of a maximal partree of E.

$2^{|I(T)|}-1$ nonempty subsets of $I(T)$, an immediate upper bound on the number of states of the TDAWG is $2^{|I(T)|}$. Whether this bound is tight remains open, but there are classes of text trees for which the number of states of their TDAWGs grows exponentially with their sizes, as the following theorem shows.

Theorem 4.7 *For every positive even integer n, let a perfect tree of height $n+1$ be labeled as follows:*[7]

1. *All nodes of depths from 0 to $n-1$ are labeled a.*
2. *For all i, $1 \le i \le 2^{n/2}$, the $(i+(i-1)2^{n/2})$-th node of depth n*[8] *is labeled c;*
3. *All other nodes of depth n are labeled b.*
4. *All leaves are labeled d.*

Then, the number of end-sets of T is at least $2^{(\frac{|T|+1}{4})^{1/2}}-2$, and so is the number of states in the TDAWG of T.

Proof: Let S be the set of nodes of depth $n/2$ in T; S is {2,3} in Figure 4.5. Given any nonempty proper subset N of $\{1,2,\cdots,2^{n/2}\}$, let L be a binary tree of height $(n/2+1)$ such that the nodes of depths from 1 to $(n/2-1)$ are labeled a and the nodes of depth $n/2$ are labeled b and the leaves are labeled d. Let L_N be the partree of L obtained by removing all its leaves and all the i-th nodes of depth $n/2$ for all $i \in N$. Then, for each $i \in N$ there is an internal partree, which is equal to L_N, rooted at the i-th node v of depth $n/2$ in T, and L_N does not equal to any internal partree of T rooted at any other node of T. Hence, $\{i$-th node of depth $n/2 \mid i \in N\}$ is an end-set of T. So each nonempty proper subset of S is an end-set of T. For example, {2} and {3} are both end-sets of the tree in Figure 4.5 which are end-set({2,5}) and end-set({3,6}) respectively.

[7] See Figure 4.5 for an example.

[8] The i-th node of depth d in T is the i-th node of depth d encountered in a breadth-first traversal of T.

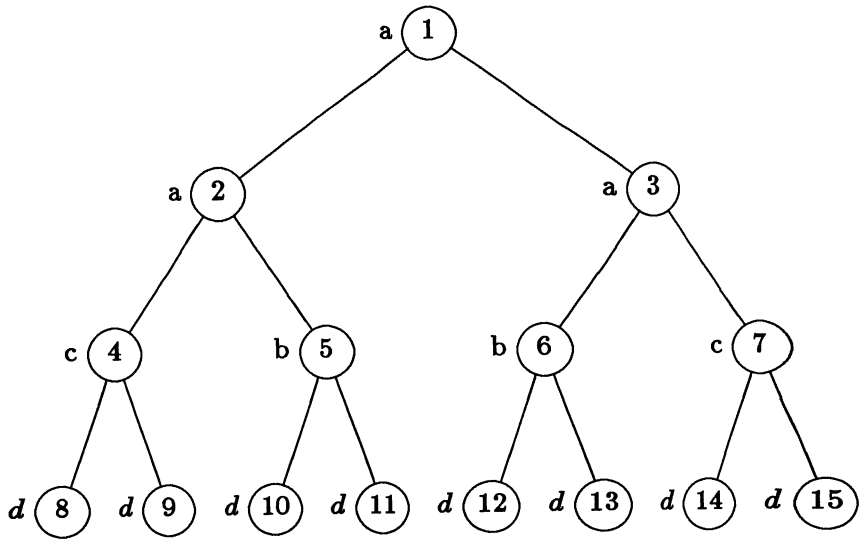


Figure 4.5: An example of a binary tree used in Theorem 4.7

Now, $| T | = 2^{n+2} - 1$, so $| S | = 2^{n/2} = (\frac{|T|+1}{4})^{1/2}$. Since there are $2^{|S|} - 2$ nonempty proper subsets of S, there are at least $(2^{(\frac{|T|+1}{4})^{1/2}} - 2)$ end-sets of T. Hence, the number of states in the TDAWG of T is at least $(2^{(\frac{|T|+1}{4})^{1/2}} - 2)$. □

Theorem 4.8 *In general, the TDAWG of a term tree T is not the minimal frontier-to-root automaton, in terms of either the number of states or the number of transitions of the automaton, which defines $\{P \mid$ the partree $I(P)$ of P is equal to some partree of $T\}$.*

Proof: This theorem is proved by an example as follows:

Let T denote the tree in Figure 4.4. The TDAWG D of T is given in Example 4.4. We observe that replacing the states s_2 and s_3 on the left hand side of the original transition functions by s_3 and s_2 respectively gives back exactly the same set of transitions as before, so we can combine s_2 and s_3 to give a new frontier-to-root automaton G defined below:

1. The state set of G is $\{s_0, s_1, s_{23}, s_4\}$. ($s_{23}$ is a new state replacing s_2 and s_3 in D.)

2. The transition function M of D is:

$$M(a, s_0, s_0) = s_{23}, \quad M(b, s_0, s_0) = s_1, \quad M(a, s_{23}, s_0) = s_4,$$
$$M(a, s_0, s_1) = s_{23}, \quad M(a, s_1, s_0) = s_{23}, \quad M(a, s_{23}, s_1) = s_4,$$
$$M(a, s_1, s_1) = s_{23},$$

3. The initial state is s_0, and the set of final states is $\{s_0, s_1, s_{23}, s_4\}$.

G defines the same set of binary trees as D. G has fewer states and few transitions than D, so D is not minimal in terms of either the number of states or the number of transitions. □

We have seen that the generalization of DAWG to TDAWG for solving tree matching problems is infeasible in general. In the next section, we will apply the ideas underlying TDAWGs to solve a special class of tree pattern matching problems, namely, those involving only perfect pattern trees.

4.5 Frontier-to-Root Automata for Perfect Patterns

We now give an algorithm which constructs a frontier-to-root automaton based on a text tree that scans perfect binary pattern trees to find all matches of the pattern in the text. This construction takes time linear in the number of nodes in the text tree.

4.5.1 Equivalence Classes and the PMA

Given a binary tree T and a partree Q rooted at some node v in T, the height of Q is defined to be the maximum integer i such that the i-th descendent of v is in Q. The height of an empty partree is defined to be -1. The partree Q is *perfect* if Q contains $2^{h+1} - 1$ nodes where h is the height of Q; in particular, the empty partree is perfect. We denote by H_T the maximum of the heights of the perfect internal partrees in T. We now define a class of equivalence relations on the nodes of T.

Definition 4.18 *Two nodes u and v of a binary tree T are said to be k-equivalent if there are equal perfect internal partrees of height k rooted at u and v; in this case, we write uE_kv.*

Note that E_k's for $k \geq 0$ only apply to internal nodes of T since there is no internal partree of height greater that -1 rooted at leaves of T.

Each E_k defined above is an equivalence relation. We denote by $[v]_k$ the equivalence class of v with respect to E_k. Also, we denote the set of all equivalence classes of E_k by $C_k(T)$ and define $C(T)$ to be $\bigcup_{k=-1}^{H_T} C_k(T)$. Note that the set of equivalence classes defined here corresponds to a subset of the set of equivalence classes induced by the equivalence relation "equal" between partrees. In the rest of this section, the word "class" stands for "equivalence class" induced by the E_k's.

We now enumerate the classes of a binary tree T in $\mathcal{T}_\Sigma$.

- Since $[u]_{-1} = [v]_{-1}$ holds for every two nodes u and v in T, there is only one class induced by E_{-1} which is $N(T)$.
- The number of internal nodes of T is $(\mid T \mid -1)/2$, and there can be at most $\mid \Sigma_2 \mid$ different labels of the internal nodes of T. So, the number of classes induced by E_0 is $\min\{(\mid T \mid -1)/2, \mid \Sigma_2 \mid\}$.
- In general, the number of classes induced by E_k, for $0 \leq k \leq H_T$, is no more than $\min\{(\mid T \mid +1)/2^{k+1} - 1, \mid \Sigma_2 \mid^{2^{k+1}-1}\}$, and the summing up the number of nodes inside every one of these classes is no more than $(\mid T \mid +1)/2^{k+1} - 1$.

Hence, the total number of classes is at most

$$1 + \sum_{k=0}^{H_T} (2^{-(k+1)}(\mid T \mid +1) - 1) = (\mid T \mid +1)(1 - 2^{-(H_T+1)}) - \frac{1}{2} H_T(H_T + 1).$$

The first term on the right hand side of this equation is at most $\mid T \mid$ since $2^{H_T+1} \leq \mid T \mid +1$, where equality holds only when T is perfect, so the total number of classes of T is no more

than $| T |$. The sum of the numbers of elements in all classes of T is at most

$$| T | + \sum_{k=0}^{H_T} (2^{-(k+1)}(| T | +1) - 1) =$$
$$| T | (2 - 2^{-(H_T+1)}) + (1 - 2^{-(H_T+1)}) - \frac{1}{2} H_T(H_T + 3)$$

which is clearly less than $2 | T |$. In other words, both the total number of classes and the total number of elements they contain are linear in $| T |$.

Definition 4.19 *A Pattern Matching Automaton based on a binary text tree T over Σ, denoted as PMA(T), is a (partial) frontier-to-root automaton with input alphabet Σ_2, state set $C(T)$, initial state $[v]_{-1}$, for some $v \in N(T)$; all states of PMA(T) are final. Its transition function $M : \cup_{k=1}^{H_T - 1}(\Sigma_2 \times C_k(T) \times C_k(T) \rightarrow C_{k+1}(T))$ is defined by:*

$$M(\sigma, [v_1]_k, [v_2]_k) = [v_3]_{k+1}$$

if $l(v_3) = \sigma$, $v_1 = child(v_3, 1)$ and $v_2 = child(v_3, 2)$, for any k, $-1 \leq k < H_T$.

Note that each state of PMA(T) corresponds to a set of nodes at which a perfect pattern tree may match. An example of a PMA will be given later in this section.

A PMA of a binary tree T is not necessarily a TDAWG of T whose input is restricted to perfect patterns. The former has at least as many states as the latter. We will discuss in next subsection how to obtain the latter from to former by combining appropriate states of the former. As pointed out above, PMA(T) has only a linear (in $| T |$) number of states; furthermore, each state of PMA(T) has at most one incoming transition, so the number of transitions of PMA(T) is also linear in $| T |$.

The following theorem helps us to construct the PMA of a binary tree.

Theorem 4.9 *Given a binary tree T, every pair of nodes in $N(T)$ are (-1)-equivalent. Two nodes u and v are k-equivalent, $k \geq 0$, if and only if $l(u) = l(v)$ and $child(u,i) E_{k-1} child(v,i)$, for $i = 1, 2$.*

Proof: There is an empty partree rooted at every node of T, so, by Definition 4.18, any two nodes in $N(T)$ are (-1)-equivalent.

if: The condition *child(u,i)*E_{k-1}*child(v,i), for* $i = 1, 2$ implies:

1. There are perfect partrees of height $k-1$ rooted at child(u,1), child(u,2), child(v,1) and child(v,2).
2. The perfect partrees of height $k-1$ rooted at child(u,1) and child(v,1) are equal, as are the ones rooted at child(u,2) and child(v,2).

(1) implies that there are perfect partrees of height k rooted at nodes u and v, and (2) together with the condition $l(u) = l(v)$ implies that these two partrees are equal. Thus, uE_kv follows.

only if: If two nodes u and v in T are k-equivalent for some $k \geq 0$, the perfect internal partree U of height k rooted at u is equal to the perfect internal partree V of height k rooted at v. This means that $l(u) = l(v)$ and the subpartree of U rooted at child(u, i) is equal to the subpartree of V rooted at child(v, i), for $i = 1, 2$. Since these subpartrees are perfect internal partrees of height $(k-1)$, it follows that child$(u, i)E_{k-1}$child(v, i), for $i = 1, 2$. □

4.5.2 Constructing the PMA

Given a binary text tree T, let each class be identified by a natural number and let $S(i)$ denote class i. We construct the classes of T in the following order: classes induced by E_{-1}, by E_0, $\cdots$, by E_{H_T}, based on Theorem 4.9. To simplify the construction algorithm, we assume that an additional variable $b(v)$ is associated with each internal node v of T. When constructing classes induced by E_{k+1} $(k \geq -1)$, $b(v)$ is defined by:

$$b(v) = \max\{i \mid v \in S(i) \text{ and } i \text{ identifies a class induced by } E_j \text{ where } j \leq k\}.$$

In the description of the following algorithm, $S_l(i)$ denotes a subset of $S(i)$, and $S_l(i)$ contains all the left children in $S(i)$; the variable c is the number of the next new class, and the value of I is the smallest of the class numbers of the k-equivalence classes during the construction of the $(k+1)$-equivalence classes.

Algorithm 4.2 *Given a binary tree T, this algorithm constructs the k-equivalence classes for all k, $-1 \leq k \leq H_T$, starting with that of E_{-1}; it also constructs the transition function M of $PMA(T)$.*

1. *Define $S(1)$ to be $N(T)$ and initialize $b(v)$ to be 1 for each node v in T. Let I be 1 and c be 2. $(k = -1)$*

2. *Repeat this step while $C_k(T)$ is nonempty, that is, while there are nonempty k-equivalence classes.*

   (a) *To construct the classes of E_{k+1} from those of E_k, a scheme reminiscent of a radix sort is used as follows.*

      i. *Partition each $S_l(i)$ so that u and v are in the same subset of $S_l(i)$ if and only if $b(child(parent(u),2))=b(child(parent(v,2))\geq I$, implying that the siblings[9] of the nodes u and v belong to the same class induced by E_k.*

      ii. *Partition each resulting subset D of $S_l(i)$ so that u and v are in the same subset of D if and only if $l(parent(u))=l(parent(v))$. For each resulting subset G of D,*

         - *define a new class consisting of all the parents of the nodes in G; identify this class by c; if it is the first class induced by E_{k+1}, assign c to a variable i; let v be a node in $S(c)$ and define the transition function value of $M(l(v), b(child(v,1)), b(child(v,2)))$ to be c; increment c by 1.*

   (b) *After constructing the E_{k+1}-equivalence classes, for each node v in a class of E_{k+1} identified by j, change $b(v)$ to j, and change I to be the current value of i.*

[9] Two different nodes are siblings of each other if they have the same parent.

The correctness of the above algorithm can be established based on Theorem 4.9.

Proposition 4.10 *Algorithm 4.2 can be implemented to run in* $O(\mid T \mid)$ *time for any binary tree* T.

Proof: At most $(\mid T \mid +1)/2^{k+1} - 1$ nodes are involved in constructing classes of E_{k+1} from those of E_k, for $-1 \leq k \leq H_T$, and the construction of the class of E_{-1} involves $\mid T \mid$ nodes. Each node involved in the construction of the classes of E_{k+1} is processed, during the construction of E_{k+1}, no more than a constant number of times. Therefore;

$$\mid T \mid + \sum_{k=-1}^{H_T} (2^{-(k+1)}(\mid T \mid +1) - 1) =$$

$$(\mid T \mid +1)(3 - 2^{-(H_T+1)}) - \frac{1}{2}H_T(H_T + 5) - 2$$

gives the total number of nodes involved in constructing the equivalence classes for all E_k's. Hence, Algorithm 4.2 requires $O(\mid T \mid)$ time. □

4.5.3 The Matching Algorithm

We now give an algorithm which uses the PMA of a text tree to solve tree matching problems.

Algorithm 4.3 *Given* $PMA(T)$ *for a binary tree* $T \in \Sigma$ *and a perfect binary pattern tree* $P \in \mathcal{P}_\Sigma$, *this algorithm finds all occurrences of* P *in* T.

- *Scan* P *with* $PMA(T)$. *If the state of* $root(P)$ *is* i, *report matches at each node contained in the set* $S(i)$; *otherwise,* $root(P)$ *is not reached by the automaton, and there is no match of* P *in* T.

Immediately, we obtain:

Theorem 4.11 *Given a text tree T and a perfect pattern tree P, after $O(|\ T\ |)$ preprocessing time, Algorithm 4.3 finds all occurrences of P in T in $O(|\ P\ |)$ time.*

Example 4.5 Let T denote the text tree of Figure 4.6; the states of PMA(T), with their corresponding equivalence classes, are listed below:

State 1: $S(1) = N(T)$,

State 2: $S(2) = \{1, 3, 5, 14\}$,

State 3: $S(3) = \{2, 4, 6, 7, 10, 11\}$,

State 4: $S(4) = \{3, 5\}$,

State 5: $S(5) = \{1\}$,

State 6: $S(6) = \{2\}$,

State 7: $S(7) = \{1\}$.

The transitions of PMA(T) are defined as follows:

$$M(a, 1, 1) = 2,$$

$$M(b, 1, 1) = 3,$$

$$M(a, 3, 2) = 5,$$

$$M(a, 3, 3) = 4,$$

$$M(b, 3, 2) = 6,$$

$$M(a, 6, 4) = 7.$$

The result of applying Algorithm 4.3 to a perfect pattern tree is illustrated in Figure 4.7; the algorithm reports that a match is found at node 2.

4.5.4 Reducing the PMA

We now define another kind of frontier-to-root automaton for binary trees which is obtained by reducing a PMA.

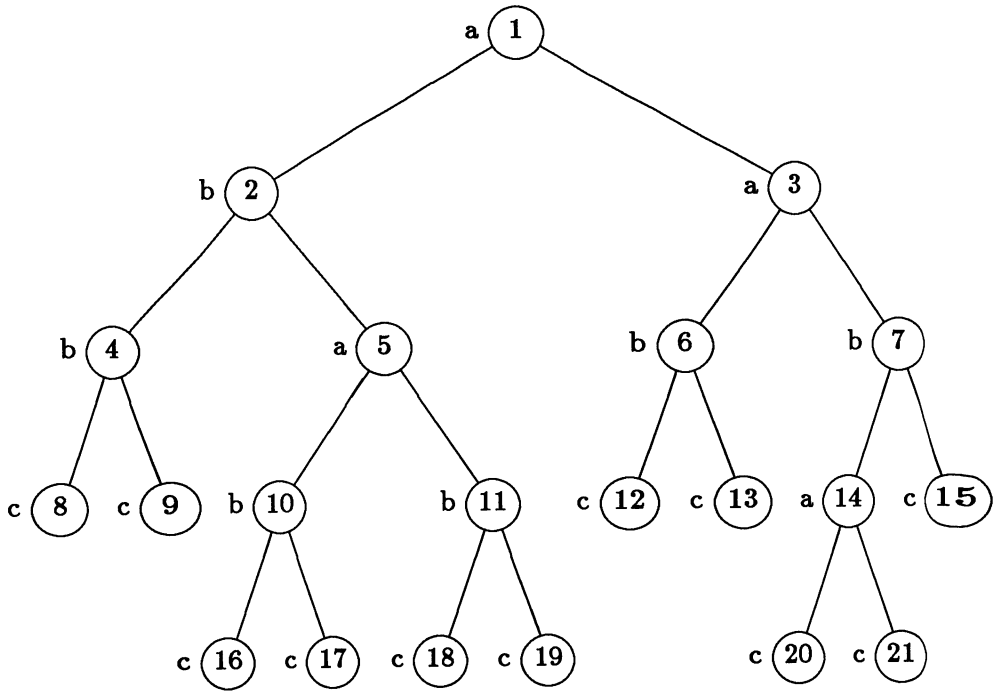


Figure 4.6: A sample text tree

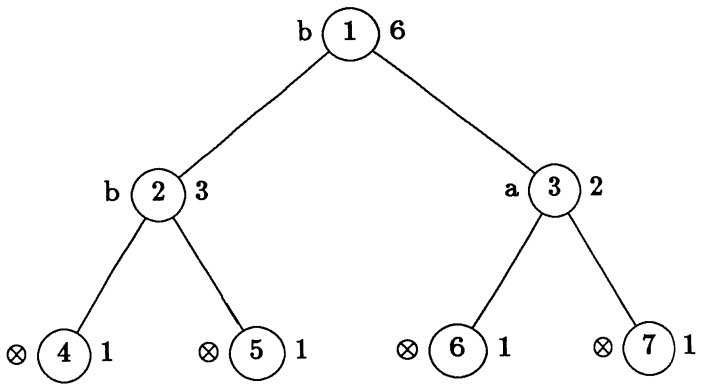


Figure 4.7: State tree resulting from Algorithm 4.3

Definition 4.20 *The PDAWG for a binary tree T over Σ is a (partial) frontier-to-root automaton with input alphabet Σ_+, state set $S = \{[Q]_T \mid Q$ is a perfect internal partree of $p\}$, and initial state $[\mu]_T$. Its transition function $M : \Sigma_2 \times S \times S \to S$ is defined by:*

$$M(\sigma, [Q_1]_T, [Q_2]_T) = \{[Q]_T \mid [Q]_T \text{ contains a perfect internal partree } R \text{ rooted at a node } v \text{ such that } l(v) = \sigma \text{ and the subpartree of } R \text{ rooted at child}(v, i) \text{ is in } [Q_i]_T \text{ for } i = 1, 2\}.$$

All states of the PDAWG are final.

In other words, the PDAWG of T is the TDAWG of T without those states corresponding to the end-sets which are not end-sets of perfect internal partrees of T.

Given a binary tree T, the PMA of T and the value of $S(i)$ for each state (or class) i of the PMA as those in Algorithm 4.2, we can construct the PDAWG of T as follows:

> If there are two different states s_1 and s_2 of PMA(T) such that $S(s_1) = S(s_2)$, remove s_2 from the state set of PMA(T) and replace all occurrences of s_2 in the transitions of PMA(T) by s_1. Continue performing this transformation on the modified automaton until no such s_1 and s_2 can be found in the automaton. Define the start state to be i such that $S(i) = N(T)$ and the set of final states to be its state set.

If the input is restricted to perfect binary trees, the PMA of T and PDAWG of T obtained by the above procedure defines the same set of perfect binary trees. Furthermore, if we have kept the $S(i)$ value of each remaining state i, then the PDAWG of T also find all occurrences of any perfect binary pattern which matches T. The proof of these statements is based on the observation that the state set of the PDAWG of T corresponds to a subset of the set of all end-sets of T. We will not get into the details of this proof here.

Example 4.6 The states of the PDAWG of the tree of Figure 4.6, with their corresponding end-sets, are listed below:

State 1: $S(1) = N(T)$,

State 2: $S(2) = \{1, 3, 5, 14\}$,

State 3: $S(3) = \{2, 4, 6, 7, 10, 11\}$,

State 4: $S(4) = \{3, 5\}$,

State 5: $S(5) = \{1\}$,

State 6: $S(6) = \{2\}$.

The transitions of PMA(T) are defined as follows:

$$M(a, 1, 1) = 2,$$

$$M(b, 1, 1) = 3,$$

$$M(a, 3, 2) = 5,$$

$$M(a, 3, 3) = 4,$$

$$M(b, 3, 2) = 6.$$

Scanning the tree of Figure 4.7 gives the same state tree and reports the same match as in Example 4.5.

Chapter 5

Other Approaches

In this chapter, some other approaches to the tree matching problems are discussed. Our purpose here is not to describe new algorithms, but rather is to offer new insight into the tree matching problem and to suggest directions for future research in this area. The descriptions of the algorithms in this chapter are not given in detail, because they are included only to illustrate the ideas behind the approaches.

5.1 Sliding the Pattern across the Text

In this section, we explore another analogy to derive algorithms for tree matching from those for string matching, and we also discuss the difficulties involved.

5.1.1 Overview

We can classify some string matching algorithms according to the directions in which they slide and scan the pattern string. For strings, sliding the pattern means moving the pattern across the text so as to compare it with different substrings of the text. For example, the Boyer-Moore algorithm [6] slides the pattern across to text from left to right and scans the

pattern from right to left, whereas the KMP algorithm both slides and scans the pattern from left to right. We can call these LR and LL algorithms, where the first letter is the sliding direction of the pattern and the second letter is the scanning direction of the pattern. Clearly, for string, the RL and RR algorithms are equivalent to the LR and LL algorithms. Fro tree pattern matching, a pattern tree can be slid and scanned either top-down or bottom-up. Hence, by analogy, we have the TB, TT, BT and BB algorithms. In this case, however, the TB and BT algorithms are not equivalent and neither are the TT and BB algorithms.

Sliding the pattern tree down the text tree involves splitting the text tree into several subtrees and sliding the pattern tree up the text tree involves merging subtrees of the text tree. In the next subsection, we discuss the TT and TB algorithms.

5.1.2 Splitting the Text Tree

Partrees and Fringes Revisited

Given a term tree T rooted at r, there is a one-to-one correspondence between the partrees of T rooted at r and the fringes of T. We say that a partree Q rooted at r and a fringe L of T *correspond* to each other if Q is obtained by deleting all subtrees rooted at the nodes in L, that is, if $Q = N(T) - \cup_{v \in L}\{\text{the subtree rooted at } v\}$. In particular, the empty partree corresponds to the fringe $\{r\}$, and the partree containing all nodes of T corresponds to the fringe $\emptyset$, which is also called the empty fringe.

A partree Q_1 is a *successor* of a partree Q_2 if they are rooted at the same node, $Q_1 \supset Q_2$ and $\mid Q_1 \mid - \mid Q_2 \mid = 1$; that is, Q_1 can be obtained from Q_2 by adding one more node which becomes a leaf of Q_1. We also say that Q_2 is a *predecessor* of Q_1.

The Algorithms

We will describe the TT and TB algorithms in terms of their similarity and their differences.

We first introduce some new terminologies. Given two nodes u and v in a term tree such that $u = a^i(v)$, for some $i \geq 0$, the string $route(u, v)$ is defined as:

$$\text{route}(u, v) = \begin{cases} \lambda & \text{if } u = v \\ \text{route}(\text{parent}(v)) \circ i & \text{if } v = \text{child}(\text{parent}(v), i) \end{cases}$$

Given a text tree T and a pattern tree P, we assume the existence of a function E from $(N(T) \times (\text{the set of fringes of } P))$ to (the set of fringes of T) so that given a node v in T and a fringe L of P, $E(v, L)$ is the set $\{u \mid v = a^i(u) \text{ for some } i \geq 0 \text{ and } \text{route}(v, u) = \text{route}(\text{root}(P), w) \text{ for some } w \in L\}$. This function may be visualized by placing the root of P at the node v in T so that $E(v, L)$ is the fringe of T which overlaps the fringe L of P. The computation of E depends on data structures by which these two trees are represented. Since we represent the trees using child and parent pointers, we can compute E by carrying out synchronously a pre-order traversal of P starting at root(P) and a pre-order traversal of the subtree of T rooted at v to search for the elements in L so as to determine the elements in $E(v, L)$.

The underlying principle of the TT algorithm is that each node of the text is only compared once, so the TT algorithm processes the text in time linear in the size of the text. Although in the TB algorithm a node of the text may be examined several times, some nodes of the text may never be examined at all, so the average time complexity may be sublinear[1], although the worst time complexity of the TB algorithm is quadratic in the sizes of the pattern and the text.

Let T and P be the text tree and the pattern tree, respectively. We first list the similarities between the TT and TB algorithms:

1. Both algorithms try to match the pattern at the root of the text. A starting fringe, which is a fringe of P, is provided for the TT algorithm to initiate comparisons. The initial starting fringe (before splitting the original text tree T) for the TT algorithm

[1]How to prove whether this complexity is or is not sublinear remains open.

is root(P). (The algorithms differ in how they carry out the comparisons, that is, in how they scan the pattern tree.)

2. For both algorithms, define the function New(S), where S is a subset of $N(P)$, to be the fringe of P corresponding to the minimal partree M which contains the roots of all subtrees of P which cannot possibly root-match P, because the nodes in ($S \cap$(the subtree rooted at $v \in M$)) cannot match the corresponding nodes of P when v is to be matched with the root of P.

3. When the algorithms find a mismatch after having compared a set S of nodes in P with the text tree, or when they have found a match in which case $S = I(P)$, they split the text tree into $\mid R \mid$ subtrees where R is E((root of the text[2]),New(S)). Then they treat the subtree rooted at every node of R as a new text tree and continue the pattern matching in these new text trees with the same pattern, whose root is to be matched with the root of each new text tree.

Now, we give the differences between the two algorithms:

1. Given a starting fringe SF of P and the text tree T_i with which P is to be root-matched, let Q be the partree corresponding to E(root(T_i),SF). The TT algorithm extends Q to its successor and compares the label of the additional node u with the label of the node v in P provided that route(root(T_i),u)=route(root(P),v) and that v is not a leaf of P. If $l(u) = l(v)$, the TT algorithm adds v to S and continues extending Q until Q equals the partree formed by $I(P)$, in which case the TT algorithm reports an occurrence of the pattern.

2. The TB algorithm compares the nodes of the pattern with those of the text bottom-up starting with the leaves of the partree formed by $I(P)$, and it does the comparisons in the reverse order to that used by the TT algorithm. It reports an occurrence of the pattern when all the nodes in $I(P)$ are matched.

[2]This text may also be a subtree of T resulting from previous splits of T

3. For the TT algorithm, the union of the starting fringes for all subtrees resulting from the same split of the text T_i is the fringe corresponding to the partree S rooted at root(P) which has been matched with a partree rooted at root(T_i) prior to the split. The starting fringe for the new text tree rooted at v corresponds to the subset of F consisting of all descendants of the node (of P) which has been compared with v.

We do not specify the exact order in which the pattern is to be scanned in the above description because there are many different possible scanning orders. One way to scan the pattern top-down is by means of a breadth-first traversal. In general, computing the function New for a given pattern P takes $O(2^{|P|})$ time.

One version of the TT algorithm for matching perfect binary patterns is described in [19] whose time complexity is linear in the total size of the text tree and the pattern tree. The scanning order is breadth-first. A version of the TB algorithm for matching perfect pattern trees is not hard to design where the scanning order is the reverse of the breadth-first order.

Whether the approaches discussed in this section can result in algorithms better than the ones described in Chapter 3 remains open. The algorithms outlined in this section play an introductory role to this approach. The main difficulty here is how to reduce the complexity of preprocessing, which is the calculation of the function New above. One way of avoiding this difficulty is to design algorithms for restricted classes of pattern trees which require less preprocessing. The algorithms applied to the class of perfect patterns are examples of this approach. Tree matching algorithms for more general classes of trees can be designed following the approaches discussed in this section, but how efficient and general these algorithms are is a topic for future research.

5.2 Partitioning the Term Tree

A number of proposed algorithms [22,14,21,23] are based on viewing a tree as a set of strings combined in some way. These algorithms solve the tree matching problems via

solving the related string matching problems. In this section, we discuss an algorithm based on partitioning a term tree into a set of partrees.

Our goal is to transform a given tree matching problem into an easier problem by reducing the original problem to a similar problem involving trees of smaller sizes. We use the observation that if the root of the pattern is labeled with σ, it can only root-match the nodes in the text that are also labeled with σ.

Given $T \in \mathcal{T}_\Sigma$ and $P \in \mathcal{P}_\Sigma$, let I_P and I_T denotes the partrees formed by $I(P)$ and $I(T)$, respectively, and let the roots of both trees be labeled σ. (This is not a restriction; if the root of T is labeled with some other symbol, we, essentially, remove the maximal partree rooted at the root(T) which does not contain any node labeled with σ, and do pattern matching for each of the remaining subtrees of T, reducing the original problem to several smaller ones.) We partition I_P into a set of partrees whose roots are all labeled with σ and whose internal nodes[3], except the root, are labeled with symbols from $\Sigma - \{\sigma\}$ and whose leaves either are labeled with σ or are leaves of I_P. We denote this set of partrees by S_P. We can also partition I_T into a set of partrees in a similar way; we denote this set by S_T^σ.

Definition 5.1 *Let Q_1 and Q_2 be two partrees.*

1. *Q_1* contains *Q_2 if Q_2 is equal to a partree formed by a subset of Q_1 which contains the root of Q_1.*

2. *Q_1 and Q_2 are* consistent *if there exists a partree which contains both Q_1 and Q_2.*

The *join* of a set of partrees is a partree with the minimum number of nodes which contains all partrees in the set.

We can construct a forest F_T of term trees based on T as follows:

[3]A node v in a partree Q is an internal node if v and at least one of its children are in Q.

Each node in the forest corresponds to a partree in S_T^σ. Suppose that a new node v corresponds to a partree Q in S_T^σ and that H is the set of all partrees in S_P which are contained in Q. Let C be the join of H. Then the degree of v is equal to the number of leaves of C which are labeled with σ. The label of v represents Q. Let D be the partree rooted at the root of Q where D is equal to C, and number the leaves of D which are labeled with σ in the order they appear in a preorder traversal of D; then, child(v,i) is the node corresponding to the partree in S_T^σ rooted at the leaf of D numbered i.

Finally, we can build a frontier-to-root tree automaton[4] for term trees, based on the pattern, that scans trees in F_T to solve the matching problems.

We have just raised a number of new questions:

1. How do we partition a tree into the desired sets of partrees?
2. Given a partree in S_T^σ, how do we find the join C mentioned above?
3. How do we construct the appropriate frontier-to-root automaton for the term trees in F_T?

The first question is easy to answer: Traverse the tree and collect the nodes labeled with σ, each of which represents the partree rooted at itself.

Answering the second question involves constructing a finite automaton which scans the partrees in S_T^σ; the number of states of such an automaton may be exponential in $\mid S_P \mid$. The idea behind this construction is to transform each set of pairwise consistent partrees into a set of strings which are then combined to form a trie.

There are a number of ways to answer the third question. One way is to simulate such a frontier-to-root automaton for term trees by a finite automaton which is obtained by

[4]Here we use the original definition of a tree automaton that reads the labels of all nodes, including the leaves, of its input trees.

modifying the finite automaton used in answering the second question. In fact, the modified finite automaton can solve the tree matching problem by scanning the trees in F_T only once. The algorithm for constructing this automaton is rather complicated, so it is not given in this thesis.

Preprocessing the pattern to construct the finite automaton takes time and space exponential in the size of the pattern. Processing the text to find all occurrences of the pattern in the text takes time linear in the size of the text. So, the entire algorithm may be implemented to run in time exponential in the size of the pattern and linear in the size of the text.

One question still remains: Are there ways of partitioning the pattern or the text which result in more efficient tree matching algorithms?

Chapter 6

Conclusions and Open Problems

6.1 Pattern Matching of Unary Trees

We can view a string as a unary tree without its leaf and, conversely, we can also regard a unary tree without its leaf as a string; in this case, we say that the string and the unary tree correspond to each other.

Until now, we have been generalizing string matching algorithms to give tree matching algorithms. In fact, we can also use tree matching algorithms to solve string matching problems by treating strings as unary trees.

When applied to tree matching problems involving only unary trees, Algorithm 3.1 is the Aho-Corasick algorithm [2] for string matching where the strings are scanned from top to bottom; by replacing each match set by its largest element (the subtree with the largest number of nodes), Algorithm 3.2 also becomes the Aho-Corasick algorithm, where the strings are scanned from bottom to top.

Given a unary tree, the nl-trie of the tree is the suffix trie for the corresponding string read from top to bottom, and the nr-trie of the tree is the suffix trie for the corresponding string read from bottom to top, and the TDAWG for the tree becomes the DAWG for the

corresponding string read from bottom to top, and the CPDAG (with appropriate labeling of its edges) of the tree is the compact suffix trie of the corresponding string read from top to bottom.

6.2 Pattern Matching for Generalized Patterns

Throughout this thesis, all leaves of pattern trees are assumed to be labeled with $\otimes$. This section shows that this restriction on pattern trees is not a serious one.

A *generalized pattern tree* over an alphabet Σ is a pattern tree over Σ except that the leaves of the tree are labeled by symbols from $\Sigma_0 \cup \{\otimes\}$ instead of only the don't care symbol $\otimes$. The definition of "root-match" in Section 2.1 applies to generalized patterns with the following addition to Definition 2.2:

> If each of the pattern and the text consists of exactly one node and the labels of the two nodes are equal, then the pattern tree root-matches the text tree.

The definition of "match" in Section 2.1 applies to general patterns without any modification.

Here are two strategies for solving tree matching problems for a generalized pattern tree P_g over Σ:

1. Suppose the symbols labeling the leaves of P_g are from the set $S \cup \{\otimes\}$, where S is a subset of Σ_0. Change the degree of each symbol in S to 1. Replace each leaf of P_g labeled with a symbol from S by a pair of parent-child nodes where the parent is labeled by the symbol labeling the original leaf and the child is labeled with $\otimes$. Similarly, replace each leaf of the text tree labeled with a symbol from S by a pair of parent-child nodes where the parent is labeled by the symbol labeling the original leaf and the child is labeled with any symbol of degree 0. Then, any of the algorithms for

matching ordinary pattern trees may be used on the modified pattern trees and text trees.

2. If all leaves of P_g are labeled with labels in Σ_0, then the only nodes in a text at which P_g may possibly occur are those nodes having at least one height(P_g)-th descendent but no (height$(P_g)+1$)-th descendent; we will call these nodes *candidate nodes*. The candidate nodes can be determined in one traversal of the text tree. Then, a straightforward procedure may be employed to check if the pattern occurs at each candidate node. Since no candidate node is an ancestor of another candidate node, this procedure takes no more node comparisons than the number of nodes in the text tree. The time complexity of this algorithm is clearly linear in the sizes of the text and the pattern.

6.3 Conclusions and Open Problems

Most existing algorithms for solving tree matching problems for general term trees consist of two phases: preprocessing the pattern tree and processing the text tree. For all of these algorithms, if the second phase takes time linear in the size of the text tree, the first takes time exponential in the height of the pattern tree; on the other hand, if the first phase takes time polynomial in the size of the pattern, the second phase takes $O(n \times h)$ time, where n is the size of the text and h is the height of the pattern.

There have been several algorithms for solving tree pattern matching for special classes of patterns. These algorithms are generally much simpler and more efficient than the ones designed for general pattern trees. There is a common feature shared by all these special classes of patterns, namely, within each such class, there is some notion of maximality for trees, which helps in designing an efficient matching algorithm for the class. For example, [14] describe one such algorithm which is in fact Algorithm 3.2 restricted to a special class of pattern trees.

Tree pattern matching provides a rich source of problems; in this thesis, we have barely scratched the surface. We only claim in Winston Churchill's words that "this is the end of the beginning". What remains to be done?

First, we have not implemented any of the algorithms discussed here, and we have little idea how well they would perform in practice. However, we are not alone in this regard; the literature is rife with unsubstantiated claims.

Second, there are many directions for further investigation. Initially, we were interested in substructure searching in trees, that is:

> Given a pattern tree P and a text tree T, we are to find if there exists a bijection between $N(P)$ and a subset of $N(T)$ which not only relates the nodes with the same labels but also preserves the ancestor relation between the nodes, that is, given any two nodes u and v in P such that $u = a^i(v)$ for some $i > 0$, if the bijection maps u to x and v to y, then $l(u) = l(x)$, $l(v) = l(y)$ and $x = a^j(y)$ for some $j > 0$.

This corresponds to string pattern matching for string patterns of the form $a_1 \otimes^* a_2 \otimes^* \cdots \otimes^* a_m$, where the a_i's are letters and $\otimes$ is the don't care symbol—the *subsequence problem*. Other obvious questions are approximate tree pattern matching, the maximal common substructure problem, and so on.

We close by stating some more open problems related to tree pattern matching.

1. The concept of "fringe" offers new insight into the understanding of tree structure. For example, we may define the *prefix* of a tree with respect to a given fringe to be the partree rooted at the root of the tree which corresponds to the fringe. Consequently, we may define the *suffix* of a tree with respect to a given fringe to be the original tree less the prefix of the tree with respect to the same fringe. Then a notion analogous to substring may be defined to be the difference between two prefixes of the tree where

one of the two prefixes includes the other. The questions are: Can we make use of these concepts to derive efficient tree matching algorithms? Are these concepts more useful in designing parallel tree matching algorithms?

2. Is there a non-trivial theoretical lower bound for the tree matching problems we have studied in terms of the sizes of the text and the pattern?

3. Can we obtain more efficient algorithms by preprocessing both the pattern and the text trees?

Bibliography

[1] A. V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, Academic Press, 1980.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[3] A. V. Aho and M. Ganapathi. Efficient tree pattern matching: An aid to code generation. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Langrages*, pages 334–340, Jan. 1985.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Reading, Mass., 1974.

[5] A. Blumer, J.Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Comput. Sci.*, 40:31–55, 1985.

[6] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, Oct. 1977.

[7] A. Bruggemann-Klein and D. Wood. *Drawing Trees Nicely with TeX.* Technical Report CS-87-05, University of Waterloo, Feb. 1987.

[8] D. R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Langrages*, pages 168–177, Jan. 1987.

[9] B. Commentz-Walter. A string matching algorithm fast on the average. In *Lecture Notes in Computer Science 71*, pages 118–132, Springer-Verlag, July 1979. 6th ICALP.

[10] Z. Galil. On improving the worst case running time of the Boyer-Moore string matching algorithm. *Commun. ACM*, 22(9):505–508, Sep. 1979.

[11] F. Gecseg and M. Steinby. *Tree Automata.* Akademiai Kiado, Budapest, 1984.

[12] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, Dec. 1978.

[13] C. M. Hoffmann and M. J. O'Donnell. An interpreter generator using tree pattern matching. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Langrages*, pages 169–179, San Antonio, Texas, 1979.

[14] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, Jan. 1982.

[15] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, pages 125–136, Denver, Colo., 1972.

[16] G. D. Knott. A numbering system for binary trees. *Commun. ACM*, 20(2):113–115, Feb. 1977.

[17] D. E. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, Pergammon Press, 1970.

[18] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in string. *SIAM J. Comput.*, 6(2):323–350, June 1977.

[19] K. Kojima. A pattern matching algorithm in binary trees. In *Lecture Notes in Computer Science 147*, pages 99–114, Springer-Verlag, 1983. RIMS Symposia on Software Science and Engineering.

[20] H. Kron. *Tree Templates and Subtree Transformational Grammars.* PhD thesis, University of California, Santa Cruz, 1975.

[21] H. Lang, M. Schimmler, and H. Schmeck. *Matching Tree Patterns Sublinear on the Average.* Technical Report, Dep. of Informatik, Univ. Kiel, Kiel, W. Germany, 1980.

[22] M. H. Overmars and J. van Leeuwen. *Rapid Subtree Identification.* Technical Report RUU-CS-79-3, University of Utrecht, the Netherlands, Feb. 1979.

[23] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. 1988. to appear in ICALP'88.

[24] R. Ramesh and I. V. Ramakrishnan. Optimal speedups for parallel pattern matching in trees. In *Lecture Notes in Computer Science 256*, pages 274–285, Springer-Verlag, May 1987. Rewriting Techniques and Applications.

[25] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string-searching. *SIAM J. Comput.*, 9(3):509–512, Aug. 1980.

[26] J. Steyaert and P. Flajolet. Patterns and pattern-matching in trees: An analysis. *Information and Control*, 58:19–58, 1983.

[27] J. W. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*, Prentice-Hall, Inc., 1973.

[28] D. Wood. *Theory of Computation.* John Wiley & Sons, 1987.