UMI Number: 9635199

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

ABSTRACT

KNOWLEDGE DISCOVERING FOR DOCUMENT CLASSIFICATION USING TREE MATCHING IN TEXPROS

by
Ching-Song Wei

This dissertation describes a knowledge-based system for classifying documents based upon the layout structure and conceptual information extracted from the content of the document. The spatial elements in a document are laid out in rectangular blocks which are represented by nodes in an ordered labelled tree, called the "layout structure tree" (L-S Tree). Each leaf node of a L-S Tree points to its corresponding block content. A knowledge Acquisition Tool (KAT) is devised to create a Document Sample Tree from L-S Tree, in which each of its leaves contains a node content conceptually describing its corresponding block content. Then, applying generalization rules, the KAT performs the inductive learning from Document Sample Trees of a type and generates fewer number of Document Type Trees to represent its type. A testing document is classified if a Document Type Tree is discovered as a substructure of the L-S Tree of the testing document; and then the exact format of the testing document can be found by matching the L-S Tree with the Document Sample Trees of the classified document type. The Document Sample Trees and Document Type Trees are called Structural Knowledge Base (SKB). The tree discovering and matching processes involve computing the edit distance and the degree of conceptual closeness between the SKB trees and the L-S Tree of a testing document by using pattern matching and discovering toolkits. Our experimental results demonstrate that many office documents can be classified correctly using the proposed approach.

KNOWLEDGE DISCOVERING FOR DOCUMENT CLASSIFICATION USING TREE MATCHING IN TEXPROS

by
Ching-Song Wei

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

May 1996

APPROVAL PAGE

KNOWLEDGE DISCOVERING FOR DOCUMENT CLASSIFICATION USING TREE MATCHING IN TEXPROS

Ching-Song Wei

Dr. Peter A. Ng, Dissertation Advisor Date
Chairperson of Department of Computer and Information Science
Professor of Computer and Information Science, NJIT

Dr. Michael Bieber, Committee Member Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Qianhong Liu, Committee Member Date
Assistant Professor of Computer and Information Science, NJIT

Dr. James A.M. McHugh, Committee Member Date
Associate Chairperson of Department of Computer and Information Science
Professor of Computer Science, NJIT

Dr. Jason T.L. Wang, Committee Member Date
Assistant Professor of Computer and Information Science, NJIT

Dr. H. T. Yeh, Committee Member Date
Supervisor, AT&T Bell Laboratories

BIOGRAPHICAL SKETCH

Author: Ching-Song Wei

Degree: Doctor of Philosophy

Date: May 1996

Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1996
- Master of Science in Mechanical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1988
- Bachelor of Science in Mechanical Engineering,
  National Cheng Kung University, Tainan, Taiwan R.O.C., 1980

Major: Computer Science

Publications:

C. S. Wei, Q. Liu, J. T. L. Wang, and P. A. Ng, "Knowledge Discovering for Document Classification Using Tree Matching in TEXPROS," submitted to *Information Sciences: An International Journal.*

C. S. Wei, J. T. L. Wang, and P. A. Ng, "Inductive Learning and Knowledge Representation for Document Classification," *Proceedings of the Third IEEE International Conference on Systems Integration*, pp. 1166-1175, August 1994.

D. T. Wang, Y. N. Hew, J. Lee, K. Chern, C. S. Wei, and P. A. Ng, "The Use of FFT on Sampled Boundary Distance," *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, Los Angeles, California, pp. 10-13, November 1990.

D. T. Wang, C. S. Wei, S. S. Chen, B. C. Sung, T. H. Shiau, and P. A. Ng, "Cross Correlation of Sampled Boundary Distances - An Approach to Object Recognition," *Proceedings of the First IEEE International Conference on Systems Integration*, Morriston, New Jersey, pp. 224-235, April 1990.

ACKNOWLEDGMENT

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Peter A. Ng, for his guidance, support, and constant encouragement throughout this work.

I wish to express my appreciation to Dr. Jason T. L. Wang and Dr. Qianhong Liu for their valuable suggestions and support.

Special thanks to Dr. Michael Bieber, Dr. James A.M. McHugh, and Dr. H. T. Yeh for actively participating in my committee.

I would also like to thank C. Y. Wang, Corrado Rizzi, and Steve Sawicki for their help, and to my employer, Pfizer Inc., for providing the educational assistance in the last three years. Finally, I want to express my appreciation to my wife, Chan Fang-Ling, to my daughter, Tina, to my son, Eric, and to my parents and brothers. Without their support and encouragement, my accomplishment would not be possible.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

Figure **Page**

CHAPTER 1

INTRODUCTION

Automatic document classification is one of the fundamental tasks in an effective Office Information System (OIS) [32]. A given document can be characterized by its content and structural organization. A common way of describing the structural organization is the layout structure which plays a significant role in document classification. For example, the type of a document can be identified at a glance over its layout structure without looking into its content, and perhaps by recognizing specific strings of characters at certain locations within the page. The *layout structure* (or *geometric structure*) of a document is the result of dividing repeatedly the layout of its content into smaller parts (that is, on the basis of its presentation). For example, a document image is composed of several blocks, each of which is a rectangular area containing a portion of document content. The *logical structure* (*conceptual structure*) of a document is the result of dividing repeatedly the content of a document into smaller parts on the basis of semantic meanings of the content. For instance, an article consists of a title, abstract, subtitles, and paragraphs [27]. In many cases, documents of the same class share a set of invariant layout features, which is called the *page layout signatures* [9]. Similarly, documents of the same class share a set of invariant logical features which is called the *logical layout signature.* The page layout signature and logical layout signature are actually only a small part of the whole layout structure. Many previous works in this area focused on paper documents of special types. The techniques work either by analyzing the layout structure or the logical structure of a document. A page layout recognition system for office documents, which was proposed by Esposito [9], can automatically detect and construct geometric characteristics of the layout components, such as height, width, spacing, and alignment. A significant number of documents were used for training the classification system. Two methods of learning from examples were employed, one is the conceptual learning and the other one is the parametric method. The former uses the inductive generalization, and the latter uses a statistical approach to find the linear discrimination function for classification. Both use only spatial relations of the layout components to determine the layout similarities and to derive the discrimination rules. Both layout similarities and discrimination rules are employed in the document type recognition step. This system considers only document type classification, but not document information extraction. A pattern recognition method for identifying letter-typed documents was proposed by Pagurek *et al* [21]. This method maps the relative positions of blocks into a matrix representation and then applies pattern matching to recognize major blocks such as *date, sender, receiver,* etc. The MAFIA system [16, 8], which was proposed by Lutz *et al.*, uses *a priori* defined type hierarchy, called the *conceptual structure definition*, to perform logical and content analysis of a document. It requires the time-consuming type hierarchy search to classify a document. Another system called ANASTASIL [7] uses a hybrid, modular knowledge representation, called the *geometric tree*, to perform a best-first search with a combination of "hypothesize and test" strategy. This system requires an exhaustive search on the geometric tree to identify the type of a document. A *document understanding* method proposed by Tsujimoto [29] transfers the layout structure of a document into its logical structure. The aim of this system is to extract the logical relationships between the document blocks of a newspaper. Schmdit and Putz proposed a rule-based recognition system, CAROL [24], to recognize automatically the important elements on the title pages of doctoral theses. The rules are generated using a machine learning method on sample documents.

In this dissertation, a system for document classification is presented and an approach is proposed to generate the knowledge of the layout structure and logical structure of any type of document. Figure 1.1 shows the overall architecture of the proposed document classification system. In this system, a document from a scanner or a facsimile is first digitized and thresholded into binary images and then encoded by the Optical Character Recognition (OCR) system. The OCR system separates the document's textual part from non-textual part, and the Page Layout Generation module converts the document into an encoded form. The encoded document is composed of the ASCII code of textual part (i.e., *character strings*, *sentences* and *paragraphs*) and the ASCII description of non-textual parts (i.e., logos, figures, pictures, *etc.*). A document input from E-mail is sent to the Code Form Generator to generate its basic block representation. The encoded document is then segmented either by the Nested Segmentation Algorithm [13] or by Adjacency Relation Segmentation [34] and then transformed into a Layout Structure Tree (L-S Tree) in which each leaf node corresponds to its content block in the document. The structural organization of a document type, such as MEMO in Figure 1.2, is segmented into blocks as depicted in Figure 1.3. The boundary of each block is identified by searching for a reasonable size of spacing between blocks. The geometric relation of these blocks can be described in term of L-S Tree structure as shown in Figure 1.4 if the document is segmented by the Adjacency Relation Segmentation Algorithm.

In the stage of document classification, a document is classified if one of the Document Type Trees can be discovered as a substructure of its L-S Tree. This process is called Document Type Tree Discovering. A modified algorithm of Discovering the Largest Approximately Common Substructures of Two Trees [25] is employed to perform the discovering process. This Document Type Tree represents a collection of Document Sample Trees of the type. Then, the exact format of the document type can be found by searching the closer match of the L-S Tree and one of these Document Sample Trees. This is called the process of Document Sample Tree Matching. The modified algorithm of Approximate Tree Pattern Matching

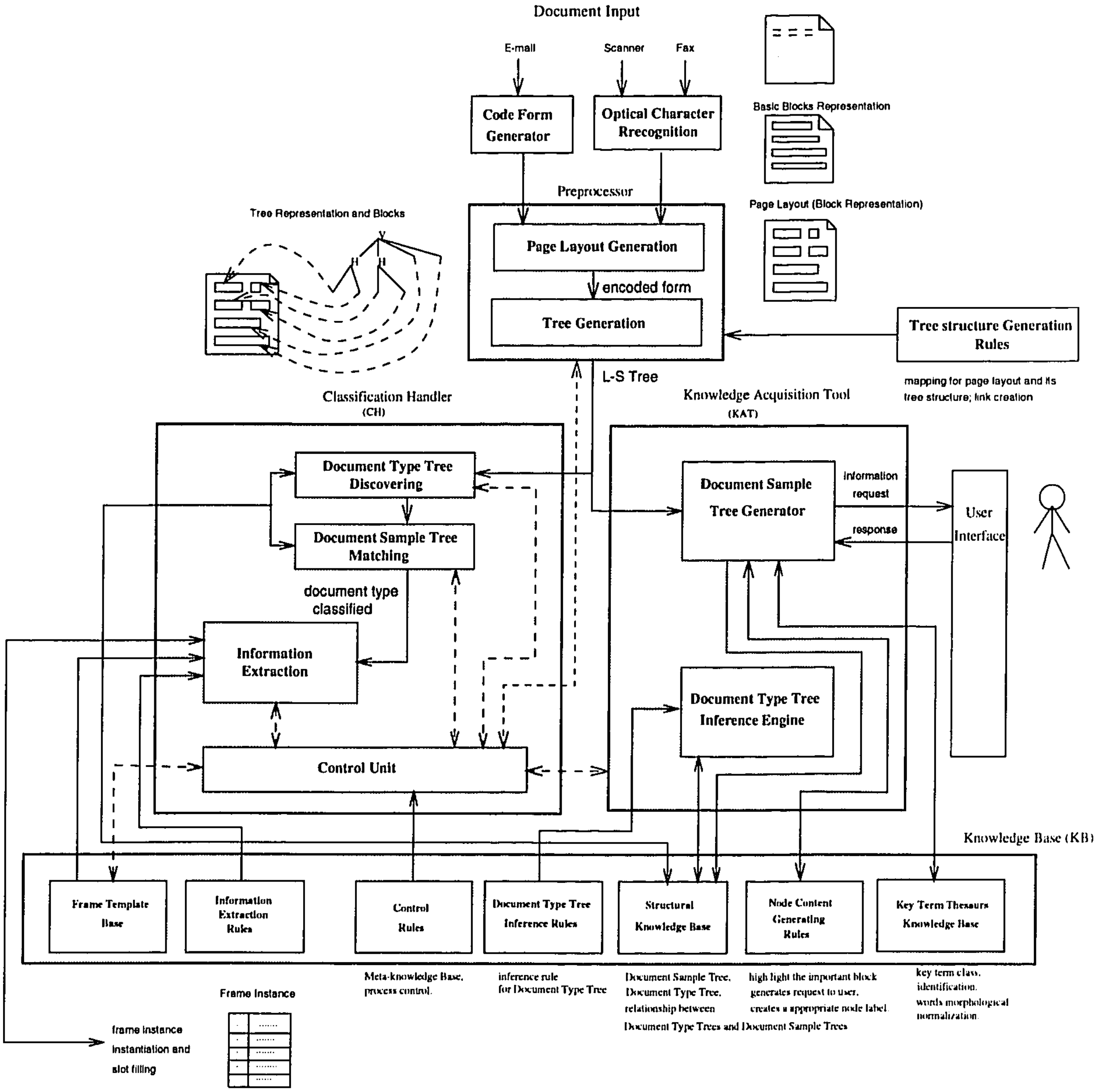
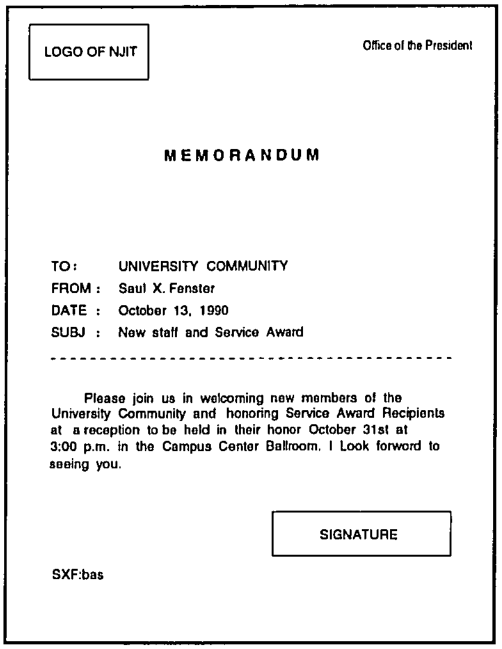

Document Input
E-mail
Scanner
Fax
Code Form Generator
Optical Character Rrecognition
Basic Blocks Representation
Page Layout (Block Representation)
Preprocessor
Tree Representation and Blocks
V
H
H
Page Layout Generation
encoded form
Tree Generation
Tree structure Generation Rules
mapping for page layout and its tree structure; link creation
L-S Tree
Classification Handler (CH)
Knowledge Acquisition Tool (KAT)
Document Type Tree Discovering
Document Sample Tree Matching
document type classified
Information Extraction
Control Unit
Document Sample Tree Generator
information request
response
User Interface
Document Type Tree Inference Engine
Knowledge Base (KB)
Frame Template Base
Information Extraction Rules
Control Rules
Document Type Tree Inference Rules
Structural Knowledge Base
Node Content Generating Rules
Key Term Thesaurs Knowledge Base
Meta-knowledge Base, process control.
inference rule for Document Type Tree
Document Sample Tree, Document Type Tree, relationship between Document Type Trees and Document Sample Trees
high light the important block generates request to user, creates a appropriate node label.
key term class. identification. words morphological normalization.
Frame Instance
frame instance instantiation and slot filling

Figure 1.1 Architecture of Knowledge Based Document Classification System.



LOGO OF NJIT

Office of the President

MEMORANDUM

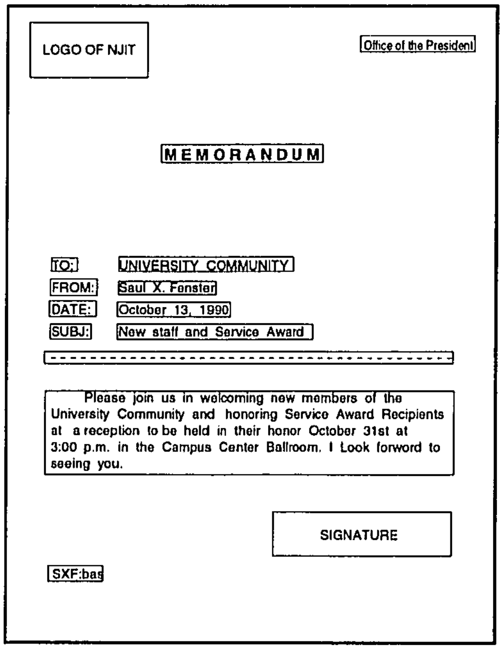
TO: UNIVERSITY COMMUNITY
FROM : Saul X. Fenster
DATE : October 13, 1990
SUBJ : New staff and Service Award

Please join us in welcoming new members of the University Community and honoring Service Award Recipients at a reception to be held in their honor October 31st at 3:00 p.m. in the Campus Center Ballroom. I Look forword to seeing you.

SIGNATURE

SXF:bas

Figure 1.2 A document example of MEMO document type.



LOGO OF NJIT

Office of the President

MEMORANDUM

TO: UNIVERSITY COMMUNITY
FROM: Saul X. Fenster
DATE: October 13, 1990
SUBJ: New staff and Service Award

Please join us in welcoming new members of the University Community and honoring Service Award Recipients at a reception to be held in their honor October 31st at 3:00 p.m. in the Campus Center Ballroom. I Look forword to seeing you.

SIGNATURE

SXF:bas

Figure 1.3 The block representation (page layout) for the example of MEMO document type in Figure 1.2.

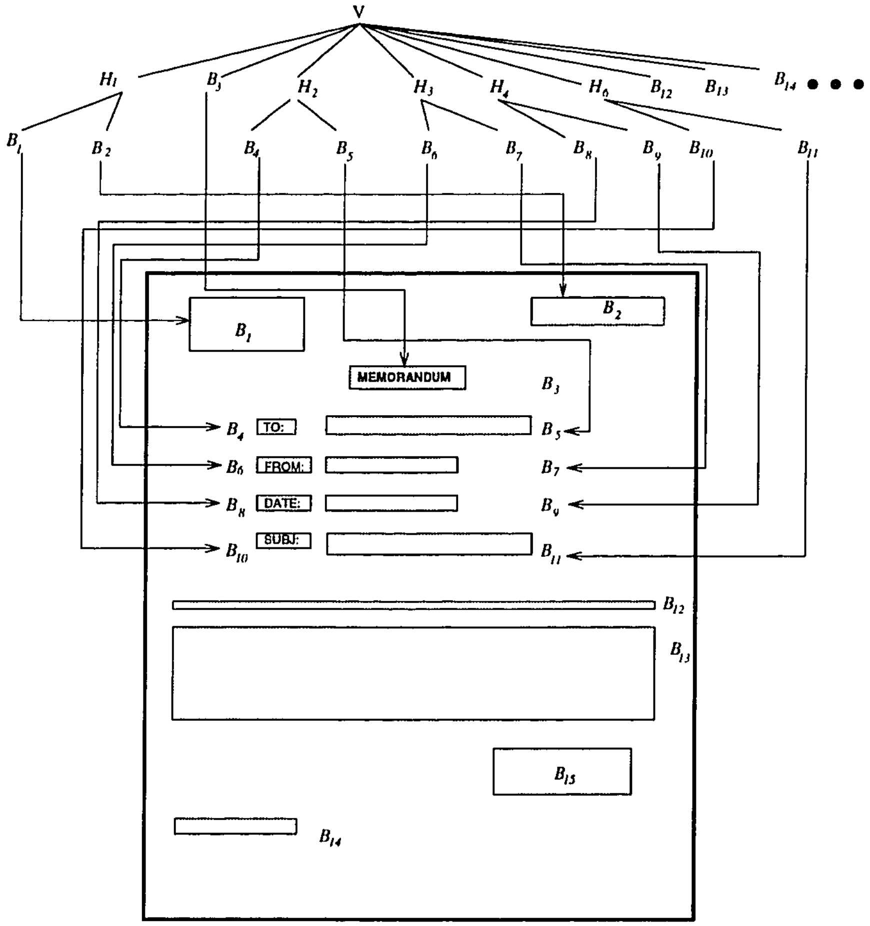
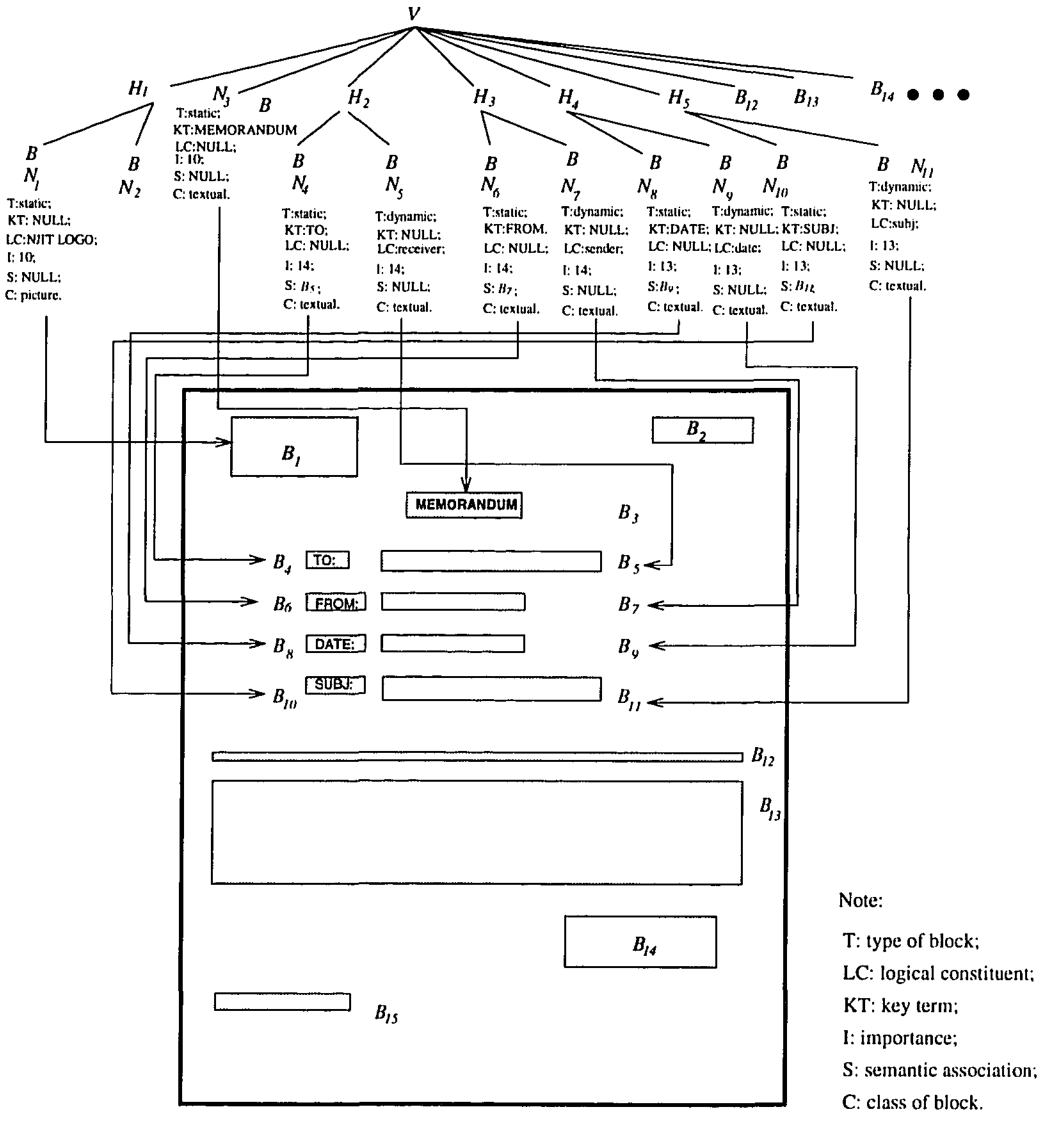


Figure 1.4 L-S Tree for the example of MEMO document type in Figure 1.2.

[30, 33] is applied to perform the matching process. Both processes use layout and conceptual analysis. Once the document type and document format of a document have been decided, some values of its corresponding *frame instance* [32] can be extracted automatically (the formal definition of frame instance will be discussed in Section 4.3). In the stage of learning process, the Knowledge Acquisition Tool is devised to learn the tree structure from the document samples and an inductive learning process is employed to derive the Document Type Trees from Document Sample Trees of each document type. The encoded document sample is transformed into L-S Tree and then sent to the Knowledge Acquisition Tool. With the help from user, Document Sample Trees are created whose leaf nodes contain conceptual information of their corresponding blocks. The information includes the type of block, key terms, logical constituents and others which describe the important semantical contents of the document. The key terms are the significant words that appeared in the document. The logical constituents are the conceptual description of major features which appeared in a document content of its type. These values will help classify the document type. One of the major features of this system is that it can be easily customized by training the system with user's document samples. By applying the Knowledge Acquisition Tool and inductive learning process the knowledge base can be built for the user's office environment.

If the example in the Figure 1.4 is a document sample, its Document Sample Tree is described in Figure 1.5. The key terms in the contents of the nodes corresponding to the blocks B_3, B_4, B_6, B_8, and B_{10} are "MEMORANDUM", "TO", "FROM", "DATE" and, "SUBJ" respectively. And the logical constituents corresponding to the blocks B_3, B_5, B_7, B_9, and B_{11} are NULL, *receiver*, *sender*, *date*, and *subject* respectively. These significant key terms and logical constituents appear only in the upper portion of the page. Associated with an image block, the node N_1 contains "*LOGO of NJIT*" as the logical constituent of this block.



Note:

T: type of block;

LC: logical constituent;

KT: key term;

I: importance;

S: semantic association;

C: class of block.

Figure 1.5 The Document Sample Tree for the example of MEMO document type in Figure 1.2.

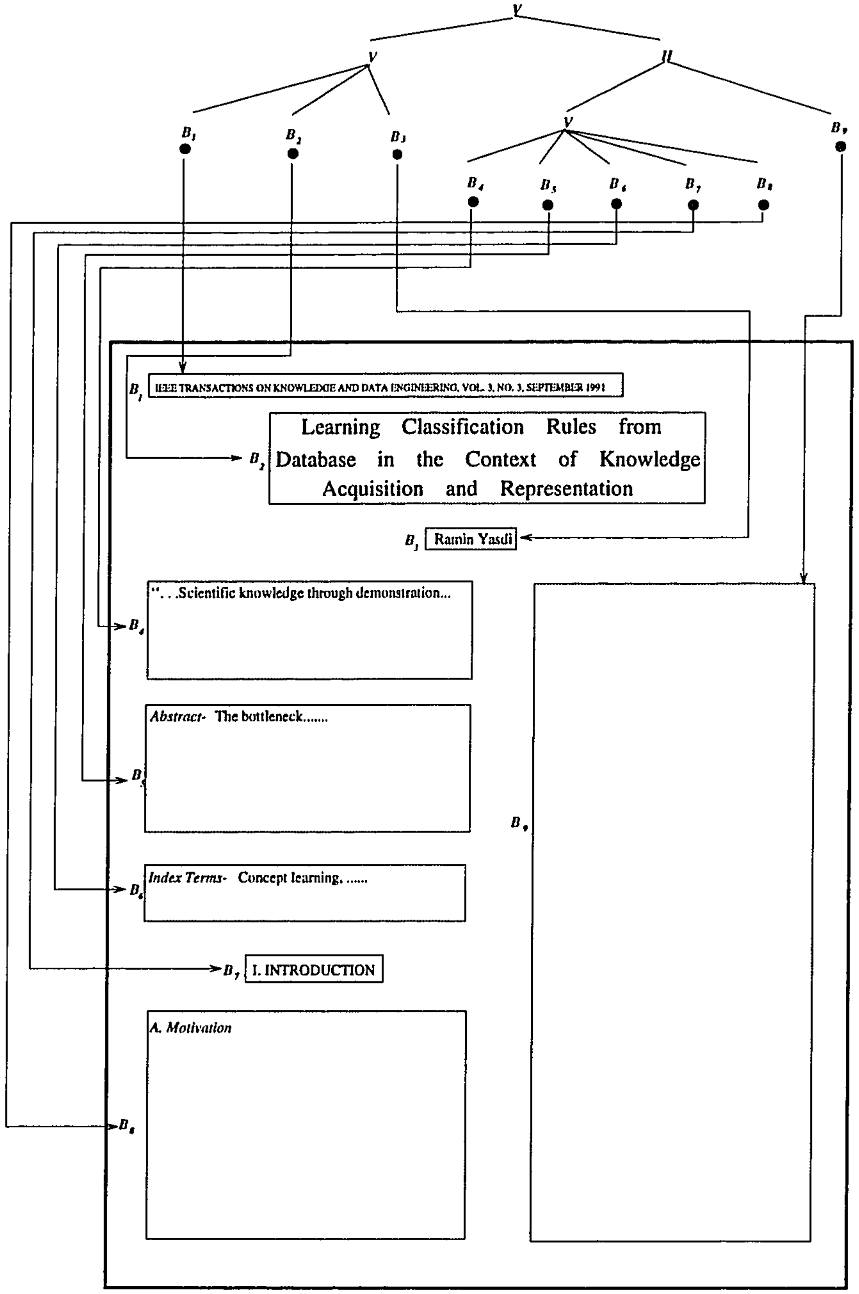

V
V
H
V
B_1
B_2
B_3
B_4
B_5
B_6
B_7
B_8
B_9
IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 3, NO. 3, SEPTEMBER 1991
Learning Classification Rules from Database in the Context of Knowledge Acquisition and Representation
Ramin Yasdi
"...Scientific knowledge through demonstration...
Abstract- The bottleneck.......
Index Terms- Concept learning, ...
I. INTRODUCTION
A. Motivation

Figure 1.6 The L-S Tree for a document example of JOURNAL PAPER document type.

Figure 1.6 shows another example of a L-S Tree in which the leaf nodes correspond to blocks of a technical paper published in a journal. The key terms in the node contents corresponding to the blocks B_1, B_5, B_6, and B_7 are "IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING", "*Abstract*", "*Index Terms*", and "Introduction" respectively. And the logical constituents corresponding to the blocks B_1, B_2, B_3, B_5 B_6, are *name of the journal*, *title of the paper*, *author*, *abstract*, and *index terms* respectively. The "IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING", "*Abstract*", "*Index Terms*" and "INTRODUCTION", which are fixed terms appearing in the first page of this journal, are selected as key terms. The contents of blocks B_2 and B_3 are distinct in different documents of the same journal. Therefore, no key term is selected from these blocks, but their logical constituents are *title of the paper* and *author* respectively. The symbols of nodes H and V denote the horizontally and vertically virtual blocks respectively, in which a group of blocks are laid out horizontally and vertically in a printed page.

Chapter 2 discusses the document layout structure including document image analysis. Chapter 3 describes the generation of L-S Tree and Document Sample Tree for a document. The tree matching operations are described in Chapter 4. The Document Type Tree Inference Engine and the components of classification system are discussed in Chapter 5. In Chapter 6, finding the Largest Common Substructure from segmented documents and their corresponding trees is presented; Chapter 7 describes the generalization of Document Sample Trees to yield Document Type Trees. In Chapter 8, the use of document classification system for identifying the type of document is discussed. Some experimental results of classifying a variety of documents are given in Chapter 9. Future research directions of this work are discussed in Chapter 10.

CHAPTER 2

DOCUMENT LAYOUT STRUCTURE ANALYSIS

After a document is scanned and digitized, the bitmap of the document image is analyzed and segmented into rectangular blocks which are called *basic blocks*, each of which individually contains one single text line, a vertical line, a horizontal line, a picture, or graphics. Some of the consecutive basic blocks containing textual lines can be assembled together to form a larger block. The result is called *block representation* of a document which can be further transformed to be a tree structure.

2.1 Document Image Analysis

The existing techniques for analyzing document image analysis are projection profile, run-length smoothing, and contour tracing [28]. The run-length smoothing method is employed in this thesis. A *run* is a set of adjacent 0's or 1's. The length of a *run* is the number of adjacent 0's or 1's in a binary sequence. The run-length smoothing algorithm (RLSA) scans row by row or column by column the binary sequences of any given document image [10, 37]. That is, the algorithm consists of vertical (row by row) and horizontal (column by column) smoothing. The smoothing converts a binary sequence f into an output sequence g according to the following rule: if the length of 0's in a run is less than or equal to a predefined threshold value C then these 0's are changed to 1's. For example, a binary sequence f which represents pixels in row by row or column by column direction is converted into g with the threshold $C = 4$.

$$f:\ 00011000000100111000111110000010000$$
$$g:\ 11111000000111111111111110000011111$$

The vertical and horizontal smoothing rules merge two runs of 1's together if the spacing between them is less than the predefined threshold. Since the horizontal and vertical spacing between document elements are different, the horizontal smoothing (processing binary sequence column by column) and the vertical smoothing (processing binary sequence row by row) use different threshold values (C's). By selecting the appropriate C's, the smoothing rules are used to construct the merged runs to form various blocks. Consequently, each block contains a single mode of content such as a single text line, a vertical line, a horizontal line, a picture, or graphics. The original RLSA [37] consists of four steps as follows:

1. A horizontal smoothing is applied to the original document image by a predefined threshold C_h.
2. A vertical smoothing is applied to the original document image by a predefined threshold C_v.
3. A logical AND operation combines two smoothing results of Steps 1 and 2.
4. An additional horizontal smoothing is applied to the output of Step 3 by a relatively small threshold C_a.

The selections of different values of C_h, C_v, and C_a affect RLSA to yield different resulting images. For a too small C_h the horizontal smoothing rule will link the characters within a word but can not bridge the inter-word space. A too large C_h, however, may cause text to be joined with non-text region. Likewise, the value of C_v may cause the similar effect. The relatively small threshold C_a of Step 4 is used to fill in the horizontal gaps between two consecutive words in a row. The original RLSA algorithm requires the scannings of whole image four times. An improvement [26] of reducing the RLSA algorithm to two steps can be done as follows:

Consider the original RLSA algorithm. Let A and B be the output of Steps 1 and 2, respectively. Step 3 is to perform $A \cap B$, which is equivalent to $A - (\neg B)$. Therefore the four steps of RLSA can be modified by combining Steps 2 and 3 into one step, which is

- If the run length of 0's in the vertical direction of the original image is greater than C_v, then reset the corresponding pixels in A to be 0's and leave A unchanged otherwise.

The three-step algorithm can be revised by processing the vertical smoothing before the horizontal smoothing.

1. A vertical smoothing is applied on the original document image using a predefined threshold C_v.

2. If the run length of 0's in the horizontal direction of the original image is greater than C_h, then reset the corresponding pixels in the output of Step 1 to 0's otherwise they remain unchanged.

3. An additional horizontal smoothing is applied to the output of Step 2 using a relatively small threshold C_a.

The three-step algorithm can be further improved by combining Steps 2 and 3 (both perform the horizontal smoothing) into one step by analyzing the relations between C_a and C_h as below. These three relations have effects on the above three-step algorithm.

(I) $C_h = C_a = C$. If the number of horizontally consecutive 0's of the original image is greater than C, then in the Step 2 the corresponding pixels must be uniformly set to 0's and will not be set to 1's in Step 3. If the number of horizontally consecutive 0's of the original image is less than or equal to C, the corresponding pixels remain unchanged in Step 2, and will be set to 1 in Step 3. From the above observation, Step 3 is able to decide whether changes take place on the output of Step 1 by checking on the original image. Since Step 3 is independent of Step 2, Step 2 and 3 can be combined together as follows:

- If the run length of 0's in the horizontal direction of the original image is greater than C, then set the corresponding pixels in the output of Step 1 to 0's otherwise set them to 1's.

It is noted that no matter what values are changed vertically in Step 1, the results obtained by the combination of Step 2 and 3, which checks on the original image horizontally, are independent of Step 1 and determine the final results of the smoothing. Therefore, the one-step horizontal smoothing algorithm applied to the original document (as Step 1 in the four-step RLAS) has the same function of above three-step algorithm. Hence, the one-step algorithm can replace the four-step RLSA algorithm when $C_a = C_h$.

(II) $C_h < C_a$. If the number of horizontally consecutive 0's of the original image is between C_h and C_a, then the corresponding pixels which remain 0's in Step 2 will be converted into 1's in Step 3. Therefore, Step 2 is redundant and can be removed.

(III) $C_a < C_h$. If the number of horizontally consecutive 0's of the original image is between C_h and C_a, then check the corresponding pixels in the output of Step 1 against C_a to determine whether 0's or 1's to be assigned to these pixels. Thus, the final improved algorithm which consists of only two steps is as follows:

Set predefined threshold values C_v, C_h and C_a.

Step 1 A vertical smoothing is applied to the original document image using a predefined threshold C_v.

Step 2 If the length of a 0's run in the horizontal direction of the original image (denoted by RL) is greater than a predefined threshold C_h, then reset the corresponding pixels in the output of Step 1 to 0's. If $RL \leq C_a$(a predefined threshold), then switch the corresponding pixels in the output of Step 1 to 1's. If $C_a < RL \leq C_h$ and that the run length of horizontally consecutive 0's in the output of Step 1 is less than or equal to C_a, then set the corresponding pixels in the output of Step 1 to 1's.

2.2 Basic Block Classification

The procedure described in the previous section can be used to divide document of mixed-mode (a mixture of text, graphics, and pictures) into basic blocks, each of which contains only a single-mode content. The next step is to classify the blocks into text, horizontal or vertical line, graphics and picture classes. In TEXPROS, a robust block classification algorithm based on clustering rules [26] is used. Let the origin of the document image be located at the upper-left corner. Each block is measured in terms of the following:

- Let (x_{min}, y_{min}) be the x- and y-coordinates of the upper-left corner and $(x_{min}+dx, y_{min}+dy)$ be the bottom-right corner of a block, where dx and dy respectively are the width and height of a block. (N) is the total number of black pixels in a block of the original image.

- Let (TH) be the horizontal transitions of white to black pixels in a block of the original image.

- Let (TV) be vertical transitions of white to black pixels in a block of the original image.

- Let (δx) be the number of columns in which black pixels exist, when a block of the original image is projected onto x-axis.

  Since the projection profile of a block onto y-axis in most cases contains black pixels in each row, it is redundant to measure the number of rows in which the black pixels exist. The following features used in block classification can be easily calculated:

- H is the height of each block , that is $H = dy$.
- R is the ratio of width to height (or aspect ratio), that is $R = \frac{dx}{dy}$.
- D is the density of black pixels in a block, that is $D = \frac{N}{dxdy}$.
- Let TH_x be the horizontal transitions of white to black pixels per unit width, where $TH_x = \frac{TH}{\delta x}$.
- Let TV_x be the vertical transitions of white to black pixels per unit width, where $TV_x = \frac{TV}{\delta x}$.

  We use δx instead of dx in the denominators of TV_x and TH_x because the values of TH_x and TV_x for all the characters in a text block are bounded by a range, which can be used to determine the text block. This will be discussed later.
- TH_y is the horizontal transitions of white to black pixels per unit height, where $TH_y = \frac{TH}{dy}$.
- TV_y is the vertical transitions of white to black pixels per unit height, where $TV_y = \frac{TV}{dy}$.

We observe that most of the office documents contain the text with the most common font and size of characters, and the mean value of heights of all the blocks in a document is approximately equal to the most common text's block-height. Therefore, the ratio of width to height, R, can be used to detect the block's orientation, such as horizontal or vertical lines. The mean horizontal transition TH_x and the mean vertical transition TV_x play important roles in text and non-textual discrimination. Both transitions are independent of variant character's fonts and sizes as long as the width to height ratio of a character is not varied significantly [26].

Let TH_x^{max} and TH_x^{min} denote the maximum and minimum values of TH_x of all characters respectively. Let TV_x^{max} and TV_x^{min} denote the maximum and minimum values of TV_x of all characters, respectively. Intuitively, both TH_x and TV_x of any text block are within the range of:

$$TH_x^{min} \leq TH_x \leq TH_x^{max}$$

$$TV_x^{min} \leq TV_x \leq TV_x^{max}$$

Let H_m be the average height of the most common blocks. The rule-based basic block segmentation algorithm is described as follows [26]:

Let c_1, c_2, c_3, c_4, c_5, c_6, c_{h1}, and c_{h2} be predefined constants.

Rule 2.1: if $c_1 H_m < H < c_2 H_m$, the block with the height H belongs to a text.

Rule 2.2: if $H < c_1 H_m$ and $c_{h1} < TH_x < c_{h2}$, this block belongs to a text.

Rule 2.3: if $H < c_1 H_m$, and $0.9 < TV_x < 1.1$, this block is a horizontal line.

Rule 2.4: if $\delta x < c_1 H_m$, $R < 1/c_3$, and $0.9 < TH_x < 1.1$, this block is a vertical line.

Rule 2.5: if $H > c_2 H_m$, $c_5 < \frac{\delta x}{\delta x} < c_6$, and $c_{h1} < TH_x < c_{h2}$, this block is a text.

Rule 2.6: if $D < c_4$, this block belongs to graphics.

Rule 2.7: otherwise, this block is a picture.

2.3 Block Representation of a Document

The output of the segmentation algorithm given in Section 2.2 is a basic block consisting of a single text line, a vertical line, a horizontal line, a picture, or graphics. Some of the basic blocks containing text lines can be assembled together to form a larger block by exploiting a number of "perceptual" criteria such as the same starting or ending columns, the same spacing, *etc.* This process groups together different blocks within the document to form a *block representation* which is the layout structure of the document. The criteria used to assemble two blocks of vertically adjacent text lines are as follows:

1. The class of blocks is textual.

2. The spacing between two blocks is less than or equal to $c * dy$, where dy is the smaller height of the blocks, and c is a pre-defined constant.

A block could be (1) a textual block which may contain strings of characters, words, sentences, or paragraphs, or (2) a non-textual block which may contain pictures, graphics, vertical lines, or horizontal lines. Formally, each block outputted from the Character Recognition System is represented by a quadruple (ID, *Type*, *Location*, *Dimension*), where ID is the unique number of each block; *Type* indicates one of the text, picture, graphics, and line (horizontal line or vertical line) classes; *Location* is specified by the coordinates of the upper-left corner (x_{min}, y_{min}) and the coordinates of the bottom-right corner ($x_{min} + dx$, $y_{min} + dy$) with respect to the origin of the document page (the upper-left corner of the document page); and *Dimension* is represented by (dx, dy).

CHAPTER 3

GENERATION OF L-S TREE AND DOCUMENT SAMPLE TREE

In this dissertation, the layout structure of a document is described by a tree structure. There are two methods that can be used to transform the layout structure of a document to a tree structure: one is of top-down approach which is the Nested Segmentation Algorithm [13] and the other one is of bottom-up approach which is the Adjacency Relation Algorithm.

3.1 Adjacency Relation Algorithm

In the Adjacency Relation Algorithm algorithm, the concept of a virtual block is used for describing the geometric relation of blocks. A virtual block is an imaginary block containing textual, non-textual, or smaller virtual blocks.

There are three types of virtual blocks: the virtual block H which contains blocks that are placed next to each other horizontally; the virtual block V which contains blocks that are placed next to each other vertically; and a virtual block I that contains blocks which are placed next to each other neither horizontally nor vertically. Note that H, V and I can contain textual, non-textual or virtual blocks. The definitions of these three types of virtual blocks are given in the following subsections.

3.1.1 Horizontally Adjacent Blocks

A *bounding box* S of blocks $A_1, A_2, \ldots,$ and A_n, where $n \geq 1$, is a minimum rectangle enclosing blocks $A_1, A_2, \ldots,$ and A_n; and satisfies the following geometric relations:

(1) $(x_s)_{min}$ = minimum of $((x_{A_1})_{min}, (x_{A_2})_{min}, \ldots,$ and $(x_{A_n})_{min})$:

the minimum x-coordinate of bounding box S is the minimum of the minimum x-coordinates of blocks $A_1, A_2, \ldots, A_n$.

(2) $(x_s)_{max}$ = maximum of $((x_{A_1})_{max}, (x_{A_2})_{max}, \ldots,$ and $(x_{A_n})_{max})$:

the maximum x-coordinate of bounding box S is the maximum of the maximum x-coordinates of blocks $A_1, A_2, \ldots, A_n$.

(3) $(y_s)_{min}$ = minimum of $((y_{A_1})_{min}, (y_{A_2})_{min}, \ldots,$ and $(y_{A_n})_{min})$:

the minimum y-coordinate of bounding box S is the minimum of the minimum y-coordinates of blocks $A_1, A_2, \ldots, A_n$.

(4) $(y_s)_{max}$ = maximum of $((y_{A_1})_{max}, (y_{A_2})_{max}, \ldots,$ and $(y_{A_n})_{max})$:

the maximum y-coordinate of bounding box S is the maximum of the maximum y-coordinates of blocks $A_1, A_2, \ldots, A_n$.

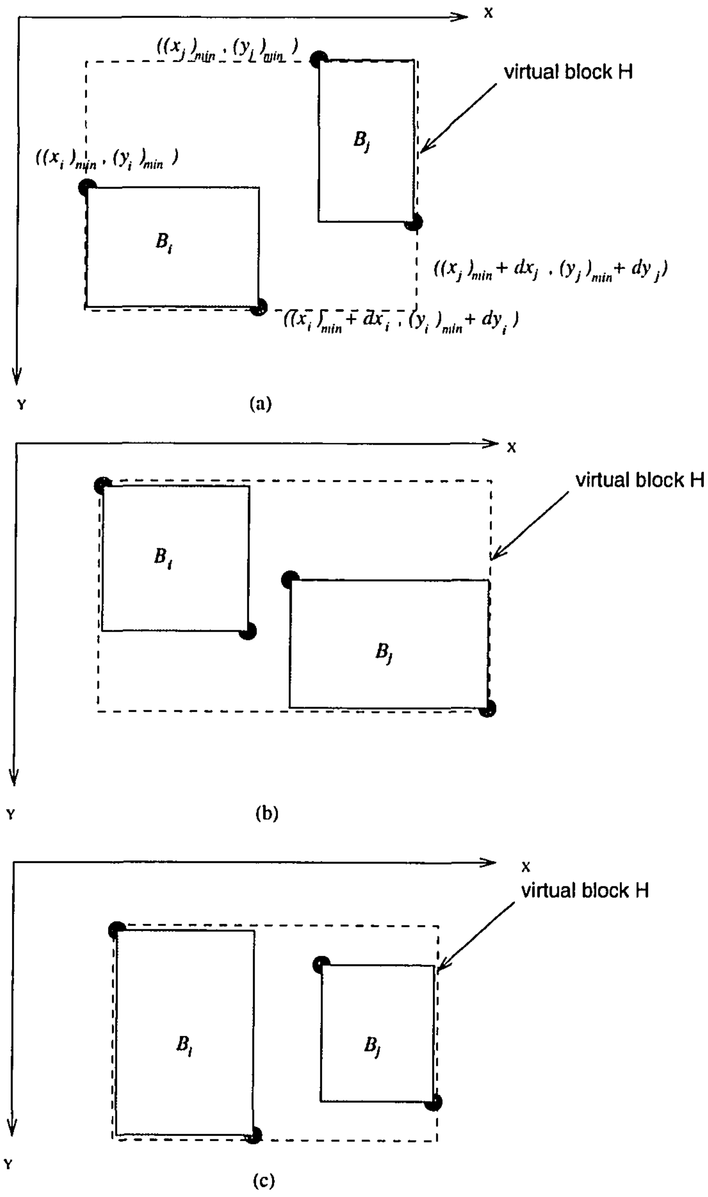
Let $\leftrightarrow$ denote *horizontally adjacent* relation. A block B_1 is said to be *horizontally adjacent* to a block B_2 (denoted by $B_1 \leftrightarrow B_2$), if their projections on y-coordinates are overlapped, and B_1 is located to the left of B_2, and there exists a bounding box that contains no other block or part of other block but B_1 and B_2. Figure 3.1 shows the geometrical relation for horizontally adjacent blocks B_i and B_j.

Let $\mathbf{B} = \{B_k | 1 \leq k \leq n\}$ be a finite set of blocks of a document page layout. If $B_i \leftrightarrow B_j$, then at least one of following geometric relations of overlapping coordinates must be true:

(1) $(y_j)_{min} \leq (y_i)_{min} \leq (y_j)_{min} + dy_j$ (in Figure 3.1(a));

(2) $(y_j)_{min} \leq (y_i)_{min} + dy_i \leq (y_j)_{min} + dy_j$ (in Figure 3.1(b)); or

(3) $(y_i)_{min} \leq (y_j)_{min},$ and $(y_i)_{min} + dy_i \geq (y_j)_{min} + dy_j$ (in Figure 3.1(c)) or vice versa.


X
Y
$((x_j)_{min}, (y_j)_{min})$
$((x_i)_{min}, (y_i)_{min})$
virtual block H
B_j
B_i
$((x_j)_{min} + dx_j, (y_j)_{min} + dy_j)$
$((x_i)_{min} + dx_i, (y_i)_{min} + dy_i)$
X
Y
virtual block H
B_i
B_j
X
Y
virtual block H
B_i
B_j

(a)

(b)

(c)

Figure 3.1 Geometrical relations for horizontally adjacent block B_i and B_j.

3.1.2 Horizontally Virtual Blocks

Let $\mathbf{B}$ be a finite set of blocks of a document page layout. A horizontally virtual block H is an imaginary block containing several horizontally adjacent blocks $\{B_1, B_2, \ldots, B_i\} \subset \mathbf{B}$, where $B_1, B_2, \ldots,$ and B_i appear from left to right in row direction (i.e., $B_1 \leftrightarrow B_2, B_2 \leftrightarrow B_3, \ldots, B_{i-1} \leftrightarrow B_i$); and this imaginary block is the bounding box of $B_1, B_2, \ldots,$ and B_i containing no other block or part of other block but $B_1, \ldots,$ and B_i. Any bounding box of (B_k, B_{k+1}), $1 \leq k \leq i-1$ contains only B_k and B_{k+1}. Any of B_k's can itself be a virtual block. For any block $B \in \mathbf{B}$ and $B \notin \{B_1, B_2, \ldots, B_i\}$, the following geometric relations of B, $\{B_1, B_2, \ldots, B_i\}$, and H must be true.

(1) $(x_1)_{max} \leq (x_2)_{min}, (x_2)_{max} \leq (x_3)_{min}, \ldots,$ and $(x_{i-1})_{max} \leq (x_i)_{min}$; and

(2) $(x_H)_{min} \geq (x_B)_{min} + dx_B$ or $(x_B)_{min} \geq (x_H)_{min} + dx_H$; and

(3) $(y_H)_{min} \geq (y_B)_{min} + dy_B$ or $(y_B)_{min} \geq (y_H)_{min} + dy_H$.

The tree structure of a horizontally virtual block H is given as follows. A node H is created with children $B_1, B_2, \ldots, B_i$ appeared from left to right according to appearing orders of their corresponding blocks. Thus, $H_i(B_1,\ B_2, \ldots, B_n)$ represents the horizontally virtual block H_i created for enclosing blocks $B_1,\ B_2, \ldots, B_n$. Figure 3.2 depicts a tree representing horizontally virtual block $H_1(B_1, B_2, B_3)$ which contains blocks $B_1, B_2,$ and B_3.

3.1.3 Vertically Adjacent Blocks

Let $\updownarrow$ denote *vertically adjacent* relation. A block B_1 is said to be *vertically adjacent* to a block B_2 (denoted by $B_1 \updownarrow B_2$) if their projections on x-coordinates are overlapped, and B_1 is located on the top of B_2, and there exists a bounding box that contains no other block or part of other block but B_1 and B_2. Figure 3.3 shows the geometrical relation for vertically adjacent blocks B_i and B_j.

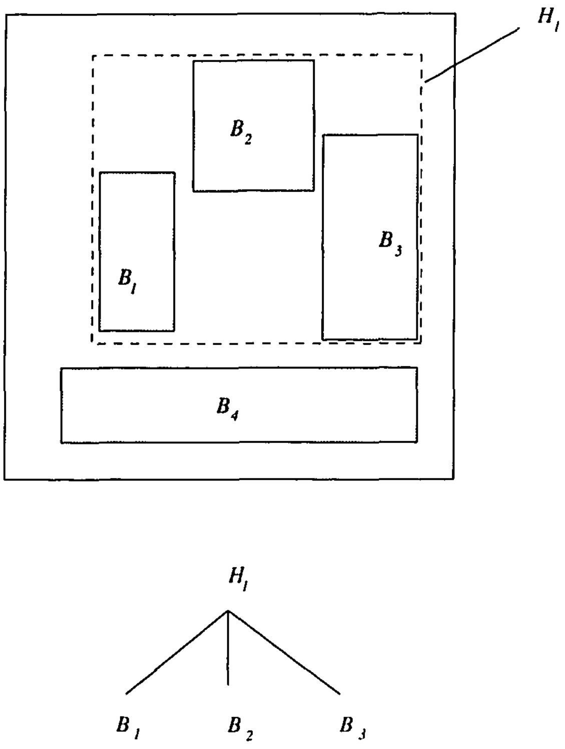


Figure 3.2 Example of a horizontally virtual block.

Let $\mathbf{B}$ $=\{B_k|1 \leq k \leq n\}$ be a set of blocks of a document page layout. Let B_i, B_j be blocks. If $B_i \updownarrow B_j$, then at least one of following geometric relations of overlapping coordinates must be true:

(1) $(x_j)_{min} \leq (x_i)_{min} \leq (x_j)_{min} + dx_j$ (in Figure 3.3(a));

(2) $(x_j)_{min} \leq (x_i)_{min} + dx_i \leq (x_j)_{min} + dx_j$ (in Figure 3.3(b)); or

(3) $(x_j)_{min} \leq (x_i)_{min}$, and $(x_j)_{min} + dx_j \geq (x_i)_{min} + dx_i$ (in Figure 3.3(c)) or vice versa.

3.1.4 Vertically Virtual Block

Let $\mathbf{B}$ be a finite set of blocks of a document page layout. A vertically virtual block V is an imaginary block containing several vertically adjacent blocks $\{B_1, B_2, \ldots,$ and $B_i\}$, where $B_1, B_2, \ldots, B_i$ are ordered from top to bottom in the column direction (i.e., $B_1 \updownarrow B_2, \ldots, B_{i-1} \updownarrow B_i$); and the imaginary block is a bounding box for $B_1, B_2, \ldots,$ and B_i containing no other block or part of another block but $B_1, B_2, \ldots, B_i$. Any bounding box of (B_k, B_{k+1}), $1 \leq k \leq i-1$ contains only B_k and B_{k+1}. Any of B_k's can itself be a virtual block. For any block $B \in \mathbf{B}$ and $B \notin \{B_1, B_2, \ldots, B_i\}$, the following geometric relations of B, $\{B_1, B_2, \ldots, B_i\}$, and V must be true.

(1) $(y_1)_{max} \leq (y_2)_{min}, (y_2)_{max} \leq (y_3)_{min}, \ldots,$ and $(y_{i-1})_{max} \leq (y_i)_{min}$; and

(2) $(x_V)_{min} \geq (x_B)_{min} + dx_B$ or $(x_B)_{min} \geq (x_V)_{min} + dx_V$; and

(3) $(y_V)_{min} \geq (y_B)_{min} + dy_B$ or $(y_B)_{min} \geq (y_V)_{min} + dy_V$.

The tree structure of a vertically virtual block V is given as follows. A node V is created having its children $B_1, B_2, \ldots,$ and B_i, ordered from left to right according

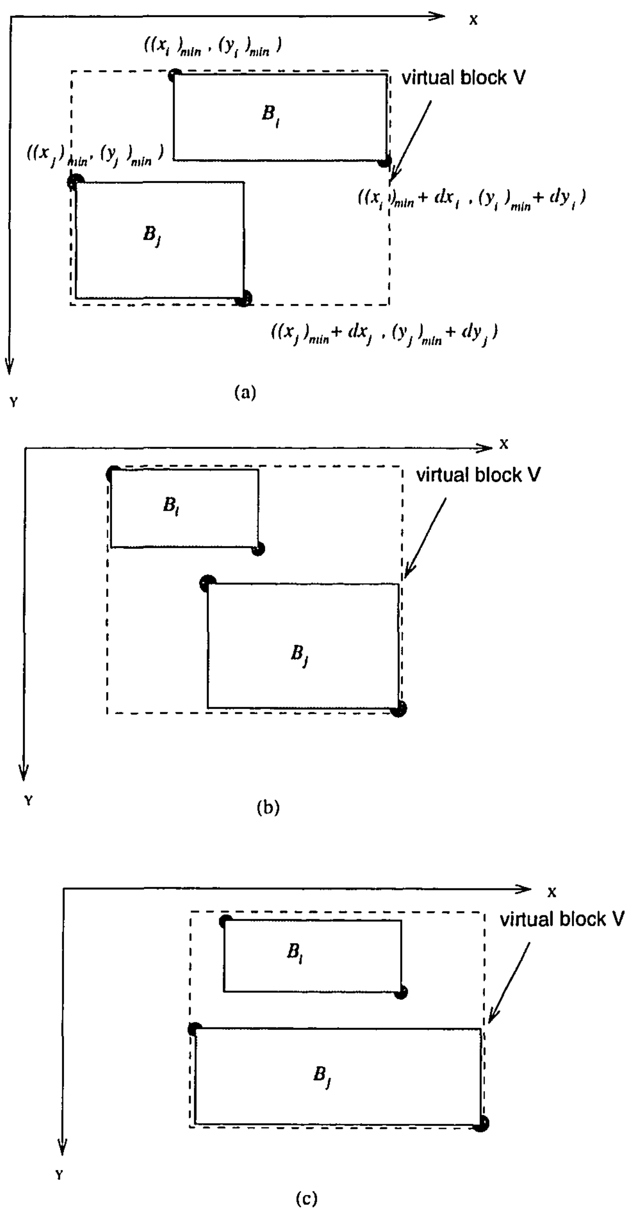


Figure 3.3 Geometrical relations for vertically adjacent block B_i and B_j.

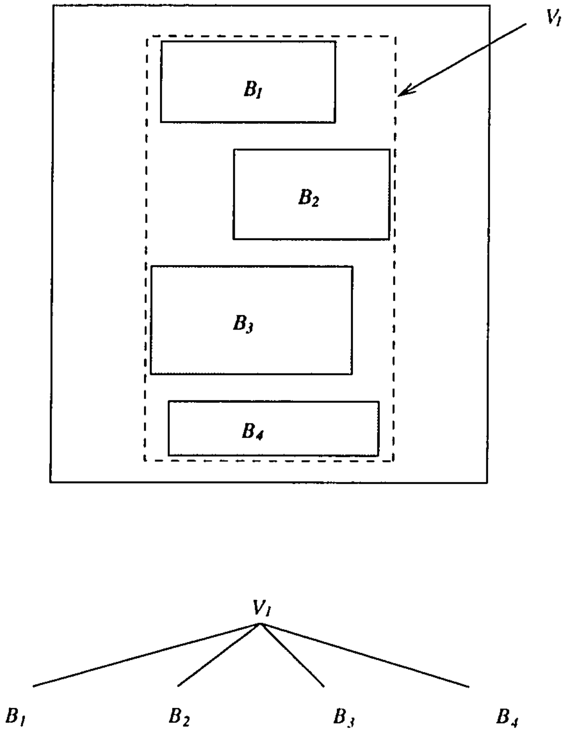


Figure 3.4 The tree representation for a vertically virtual block.

to the order of their blocks from top to bottom. For example, Figure 3.4 shows a vertically virtual block V_1 of the blocks B_1, B_2, B_3, and B_4. These blocks can be represented by a tree structure, in which B_1, B_2, B_3, and B_4 appear from left to right as the children of the parent V_1. Thus, $V_i(B_1, B_2, \ldots, B_n)$ represents the vertically virtual block V_i created for enclosing blocks $B_1, B_2, \ldots, B_n$.

3.1.5 Independently Virtual Block

An independently virtual block I is an imaginary block which is a bounding box containing several blocks that are neither in vertically adjacent nor horizontally adjacent relation. Thus $I_i(B_1, B_2, \ldots, \text{and } B_n)$ represents tree structure for the independently virtual block I_i created for enclosing blocks $B_1, B_2, \ldots, B_n$, which are ordered from left to right according to the $(x)_{min}$-coordinates of their *Locations* in left to right sequence.

3.1.6 Properties of Virtual Blocks

Let $\leftrightarrow$ be the *horizontally adjacent* relation and $\mathbf{B} = \{B_i \mid 1 \leq i \leq n\}$.

(1) If $B_1 \leftrightarrow B_2$, then $B_2 \leftrightarrow B_1$ is not true.

Proof. This is followed by definition in Section 3.1.1.

(2) For $B_1 \leftrightarrow B_2$ and $B_2 \leftrightarrow B_3$, $H(B_1, B_2, B_3)$ may or may not exist.

Proof. This can be proven by an example in Figure 3.5 where $B_1 \leftrightarrow B_2$ and $B_2 \leftrightarrow B_3$ are true. Since the bounding box S for B_1, B_2, and B_3 includes part of B_4, S violates the definition in Section 3.1.2. Therefore $H(B_1, B_2, B_3)$ does not exist.

(3) If there exists a $H(B_1, B_2, B_3)$ then $H(B_2, B_3)$ and $H(B_1, B_2)$ also exist, and $B_1 \leftrightarrow H(B_2, B_3)$ and $H(B_1, B_2) \leftrightarrow B_3$.

Proof. By definition of Section 3.1.2, if $H(B_1, B_2, B_3)$ exists, then within the bounding box of B_1, B_2, and B_3, $B_1 \leftrightarrow B_2$ and $B_2 \leftrightarrow B_3$. For $B_1 \leftrightarrow B_2$ or $B_2 \leftrightarrow B_3$, we can construct $H(B_1, B_2)$ or $H(B_2, B_3)$ respectively. Since $B_1 \leftrightarrow B_2$ and $H(B_2, B_3)$ are true, by definition of bounding box and horizontally virtual box, $B_1 \leftrightarrow H(B_2, B_3)$ can be shown to be true. Similarly, since $H(B_1, B_2)$ exists and $B_2 \leftrightarrow B_3$ is true, then $H(B_1, B_2) \leftrightarrow B_3$ is true.

Let $\updownarrow$ be the *vertically adjacent* relation.

(1) If $B_1 \updownarrow B_2$, then $B_2 \updownarrow B_1$ is not true.

Proof. This is followed by definition in Section 3.1.3.

(2) For $B_1 \updownarrow B_2$, and $B_2 \updownarrow B_3$, then $V(B_1, B_2, B_3)$ may or may not exist.

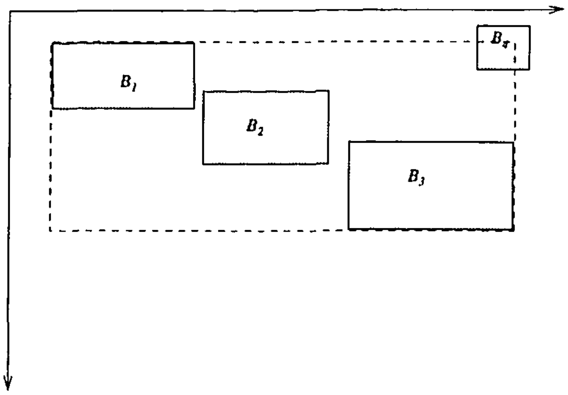


Figure 3.5 $B_1 \leftrightarrow B_2$ and $B_2 \leftrightarrow B_3$, but $H(B_1, B_2, B_3)$ is not true.

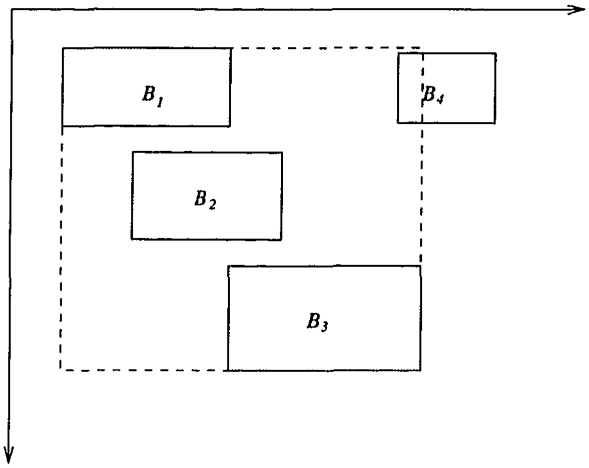


Figure 3.6 $B_1 \updownarrow B_2$ and $B_2 \updownarrow B_3$, but $V(B_1, B_2, B_3)$ is not true.

Proof. It is similar to item(2) above and can be shown by a counter-example in Figure 3.6.

(3) If there exists a $V(B_1, B_2, B_3)$, then $V(B_2, B_3)$ and $V(B_1, B_2)$ also exist, and $B_1 \updownarrow V(B_2, B_3)$ and $V(B_1, B_2) \updownarrow B_3$;

Proof. It is similar to the proof for item(3) above.

3.1.7 Tree Structure Transformation Algorithm

The sequence of blocks from OCR is based on the ordering of the $(y)_{min}$-coordinates of the blocks, and then on (x_{min})-coordinates if the (y_{min})-coordinates are the same. Therefore, a set of blocks can be viewed as an array of a link list in which the first element is the topmost block and the last element is the block at the bottom of the page. This list is also called a block list. A transformation procedure from a block list to a tree representation is given below:

1. Create a block list containing all the blocks to be considered in a given page layout.

2. Find and create all the possible horizontally virtual blocks which will enclose all the possible blocks in page layout.

3. For each of the horizontally virtual blocks found in step 2, a H node is created in the corresponding tree structure, with children nodes appeared from left to right according to the appearing order from left to right of enclosed blocks.

4. Replace the enclosed blocks by the horizontally virtual blocks in the block list.

5. If no horizontally virtual block can be found, then find and create all the possible vertically virtual blocks which enclose all the possible blocks in a page layout.

6. For each of the vertically virtual blocks found in Step 5, a V node is created in the corresponding tree structure, with children nodes appeared from left to right according to their appearing order from top to bottom of enclosed blocks.

7. Replace the enclosed blocks with vertically virtual blocks in the block list.

8. Go to step 2, until no more horizontally or vertically virtual block can be found.

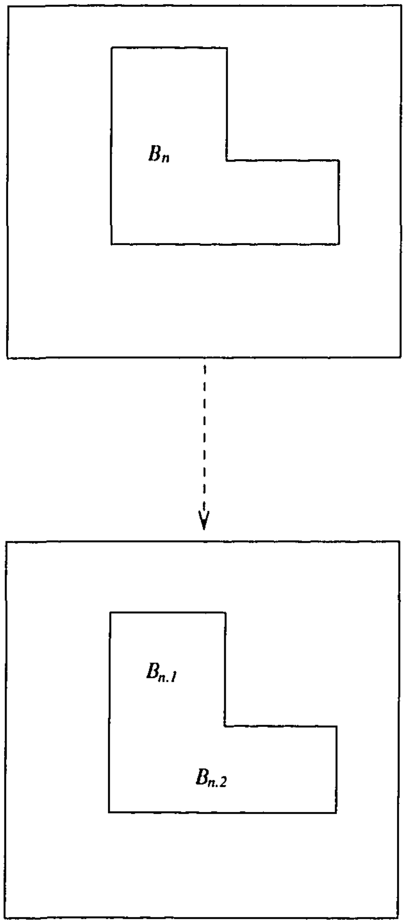


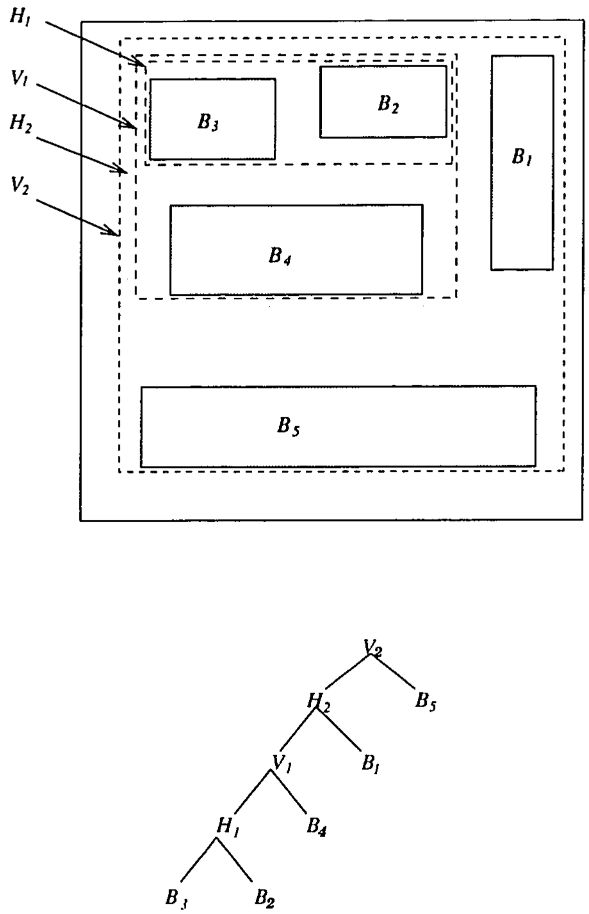
Figure 3.7 L-shape block.

9. If the block list contains more than one block, then create an independently virtual block to enclose these blocks.

10. End of process.

The distance of two horizontally adjacent blocks is defined as the shortest distance of their projections on the x axis. Similarly, the distance of two vertically adjacent blocks is defined as the shortest distance of their projections on the y axis.

If an L-shape textual block is found, such as the one in Figure 3.7, this block will be divided by a horizontal line to form two separate rectangle blocks, and their block numbers will have one additional digit to indicate that they are from the



The virtual block is greater than it is supposed to be for visualization purpose.

Figure 3.8 Example of a set of blocks and its tree structure.

same block. The same algorithm is then applied to generate the corresponding tree representation. The reason that one can divide the block horizontally is that there will always be spacing between two text lines. Therefore the spacing will be a good place for the divider.

Consider Figure 3.8. Initially, the block list contains B_1, B_2, B_3, B_4, and B_5 (ordered in y-coordinates). The algorithm will try to find the horizontal virtual blocks starting from B_1. Since there does not exist a block B_m such that $B_1 \leftrightarrow B_m$, the searching process switches to B_2. Since $B_2 \leftrightarrow B_3$, $H_1(B_3, B_2)$ is created. The search process is ended since there are no more blocks which can be enclosed in H_1. In the corresponding tree structure, the node H_1 is created. B_2 and B_3 are also replaced by H_1 in the block list. Now the block list contains B_1, H_1, B_4 and B_5. Then we search for horizontally virtual blocks from B_4. There is no block B_m such that $B_4 \leftrightarrow B_m$ and the same is true for B_5. Next, we search the vertically virtual blocks starting from B_1. There is no other block vertically adjacent to B_1. Then a V_1 is found which encloses H_1 and B_4. In the tree structure the node V_1 has H_1 and B_4 as its left and right child. We replace H_1 and B_4 by V_1 in the block list. Now the block list contains B_1, V_1 and B_5. Next H_2 is found which contains V_1 and B_1. A node H_2 is created in tree structure. The block list now contains H_2 and B_5 only. Finally a vertically virtual block V_2 is created to enclose H_2 and B_5. The final tree representation is $V_2(H_2(V_1(H_1(B_3, B_2), B_4), B_1), B_5)$.

3.2 Nested Segmentation Algorithm

The Nested Segmentation Algorithm employs top-down method to cut a document into segments until no segment can be further divided. Each segment is a rectangular portion of a document containing at least one block. There are two types of segments: *basic segment* which contains only a block, and *composite segment* which is composed of smaller segments. A document layout is first cut horizontally or vertically into segments which are at level 1. (Each horizontal (or vertical) cut is called a H (or V) cut.) All the composite segments at level i are further cut into a number of smaller segments at level $i + 1$. The segmentation preprocess terminates when all the segments cannot be further divided.

Let D represent the set of all the segments contained in a segmented document layout, and let $d[i]$ represent the ith segment of document D according to the segmentation ordering. Each segment is associated with a quadruple $(Id, Type, Orientation, Composition)$, where Id is the identifier (id) of the segment; $Type$ indicates whether the segment is basic or composite; $Orientation$ specifies whether the segment can be further divided vertically or horizontally if the Nested Segmentation Algorithm is used, or whether the segment can be enclosed by an H (horizontal) virtual block or V (vertical) virtual block if the Adjacency Relation Segmentation Algorithm [34] is used; the *Composition* specifies the *id*s of the segments contained in this segment. When a segment is basic, the segment *id* is the *id* of the block contained in the segment.

For example, a segment $d[0]$ contains several segments $d[1], d[2], \ldots, d[n]$, at the same level, which are located in the order either from top to bottom within this segment if this segment is divided horizontally, or from left to right if this segment is divided vertically. Then, the value of *Composition* is $(d[0],\ (d[1]d[2], \ldots, d[n]))$. For a document layout, the left to right relation of documents in V cut, top to bottom relation of documents in H cut and parent to child relation between levels are all significant in a document. The detail of the nested segmentation algorithm is shown in [13].

The L-S Tree generated by Nested Segmented Algorithm is an ordered labelled tree in which a node corresponds to a segment of the nestedly segmented document. Each node is labelled as indicating one of the three available types of nodes: basic node (*B_node*), horizontal node (*H_node*), and vertical node (*V_node*). A *B_node* represents a basic segment which cannot be further divided. An *H_node* represents a composite segment, which is divided horizontally into smaller segments. These smaller segments contained in the composite segment are represented as the children of the *H_node*. The order of the children of an *H_node* appearing from left to right is the appearing locations of the smaller segments from top to bottom. Similarly, a *V_node* represents a composite segment which is divided vertically into smaller segments. The order of the children of a *V_node* appearing from left to right represents the appearing locations of the smaller segments in the composite segment

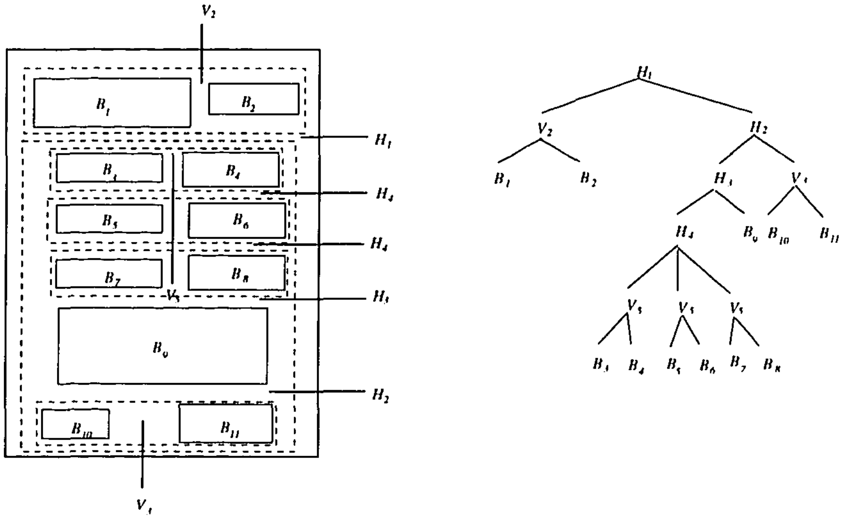


Figure 3.9 An example of document page layout segmentation and its corresponding L-S Tree.

from left to right. Figure 3.9 depicts the segments and the resulting trees of a document segmented by the Nested Segmentation Algorithm.

3.3 Knowledge Acquisition for Document Sample Tree

The conceptual structure of a document is its logical constituents such as *sender, receiver, subject,* and *date* in the document type of MEMO. Therefore, the conceptual structure can be represented by a set of attribute name and attribute type pairs. The conceptual structure can be described as (MEMO{(*Receiver*, *string*), (*Sender*, *string*), (*Subject*, *string*), (*Date*, *string*), (*Content*, *text*)}). Each document type has its unique conceptual structure, but there are more than one layout structures associated with a document type [32].

Conceptually, a document can be divided into two parts: *structured* and *unstructured* parts. The relative locations of structured parts of documents of the same type always remain the same. The structured parts can be further classified as *static* and *dynamic* parts. The static part has a fixed relative location and the same semantics among the documents of the same document type. On the other hand, the dynamic parts of different documents of the same document type are varied. For example, the static parts of a memo document in Figure 1.3 include the key terms "MEMORANDUM," "TO," "FROM," "DATE," "SUBJ," and the image block LOGO OF NJIT, *etc.* The dynamic part refers to various strings such as "UNIVERSITY COMMUNITY", " Saul X. Fenster", "October 13, 1990", "New Staff and Service Award", *etc.* Some key terms appeared in the static part can be different among documents even though they have the same meanings. For example, "MEMORANDUM", "MEMO" and "NOTICE" are used in different documents to refer to the same key term of memorandum. A *thesaurus* is therefore implemented for storing the terms which are semantically equivalent. When two terms belong to the same class in the thesaurus, they are semantically equivalent.

The main body of a document may be the structural parts or the unstructured parts. In the case of MEMO document type, its main body is the content of the memo. In the Figure 1.3, it starts with "Please joint us ...". In the case of 1040 Tax Return form type, the major components are structural parts such as first name, last name, social security number, spouse's social security number, etc.

The function of Knowledge Acquisition Tool (KAT) [2, 8, 11, 15, 22, 23, 24, 38] is to acquire the necessary classification knowledge from a user and converts the knowledge into a tree representation that can be used by a knowledge-based system. KAT consists of a Document Sample Tree Generator module and a Document Type Tree Inference Engine.

Given a L-S Tree and its corresponding sample document, the Document Sample Tree Generator module of the KAT (as shown in Figure 1.1) generates a Document Sample Tree by activating the User Interface to provide the user with Pop-Up windows for entering information of the *structured* part of a sample document. The information includes: (1) *type of block*, which can be static, dynamic or mixed.

Table 3.1 Node content for the block B_4 in Figure 1.5.

attribute	**value**
type of block	static
key term	TO
logical constituent	$NULL$
importance	14
semantic association	B_5
class of block	textual

(2) *key term*, which is the content of a block if it is of a static type (the strings appearing in the block), or the static part of a block's content if it is of a mixed type (a mixed type contains material from both fixed and variable parts), or null otherwise. (3) *logical constituent*, which is the conceptual description for the dynamic block, or NULL otherwise. (4) *importance*, which indicates to what extent the node contributes to the process of identifying a document type. The $Importance_{node}(NC)$ [12] is defined as follows. Let S be a set document sample trees of document type K and let NC be a node content in the document sample tree S_i, where $S_i \in$ S.

$$Importance_{node}(NC) \;=\; |\{S'|S' \in \text{S and } \exists NC' \in S',\ NC == NC'\}|.$$

The symbol $|.|$ denotes the cardinality of the indicated set. Intuitively, the importance of a node content, say, containing the key term "MEMO" in a Document Sample Tree of MEMO document type, is measured by the number of occurrences of this term appeared in the set of Document Sample Trees of MEMO type. (5) a collection of identifications of dynamic blocks that have semantic association with a static node. (6) *class of block*, which can be textual, image or graphics. This information forms a node content of a basic node in a L-S tree for the sample document. Two node contents are shown in Tables 3.1 and Table 3.2 for nodes B_4 and B_1 in Figure 1.5.

Table 3.2 Node content for block B_1 in Figure 1.5.

attribute	**value**
type of block	static
key term	*NULL*
logical constituent	*NJIT LOGO*
importance	10
semantic association	*NULL*
class of block	picture

In Table 3.1, its key term is "TO" and the logical constituent is *NULL*. The key term is a specific string that appears as content in its corresponding block content of a document. The logical constituent is the major conceptual description appearing in a document which describes the semantics of text content and therefore defines its type. In block B_3, "MEMORANDUM" is the key term. In the thesaurus, "MEMO" is the class for strings such as "MEMORANDUM", "MEMO" and "NOTICE". Therefore, "MEMORANDUM", "MEMO" and "NOTICE" are semantically equivalent. Table 3.2 describes a non-textual block. The logical constituent is *NJIT LOGO* because B_1 contains a NJIT logo which is a match with the image of NJIT LOGO.

CHAPTER 4

TREE MATCHING

Approximate Tree By Example (*ATBE*) [30, 33] is a system designed to support constructing, comparing and querying sets of ordered, labelled trees. In these trees the nodes are labelled and the order from left to right among siblings is significant. *ATBE* allows inexact match of trees which is appropriate for our document classification application because two documents with the same type may not have the same tree structures even when they share the same features. Mostly the layouts of documents of the same type such as letter type are different. Consider documents of letter type. If we disregard most of the unimportant contents for the document type classification, which are mostly the unstructured parts of the textual content, and consider only to the layout blocks containing key terms and logical constituents (such as *logo, date, sender, receiver, saluting words, ending words,* and *signature* in the letter type), then the tree models for this document type are limited to several different trees only. Our experiments showed that there are only 6 different trees found for 20 different letters, without taking unimportant textual content into consideration. Otherwise, there will be 20 different trees for these different letters.

4.1 Tree Edit Operation

Tree Editing Distance [33] is used to measure the difference between two trees. Informally the distance of the trees T_1 and T_2 is the cheapest cost among all transformations from T_1 to T_2, or visa versa. There are three types of edit operations: *relabel*, *delete*, and *insert*. The representation for these operations is $u \rightarrow v$, where u and v is either a node or the null node (Λ). Then $u \rightarrow v$ represents a relabel operation if $u \neq \Lambda$ and $v \neq \Lambda$; a delete operation if $u \neq \Lambda$ and $v = \Lambda$; and an insert operation if $u = \Lambda$ and $v \neq \Lambda$. Let T_2 be the tree obtained from the application of edit operation

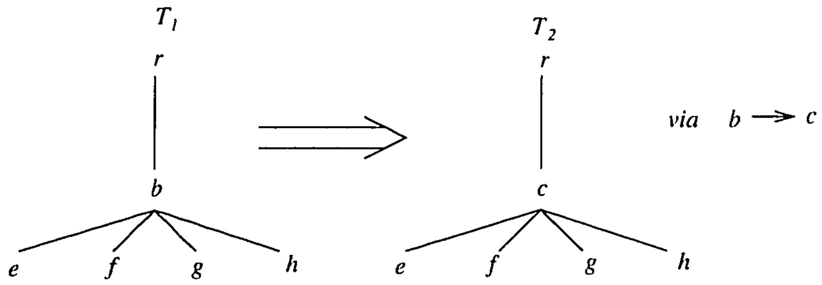
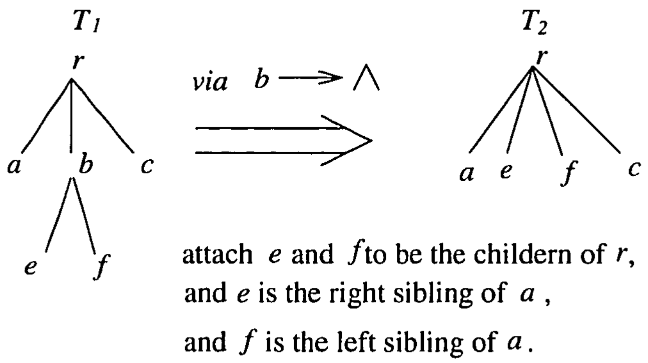


Figure 4.1 Relabelling of a node label (b) to label (c).



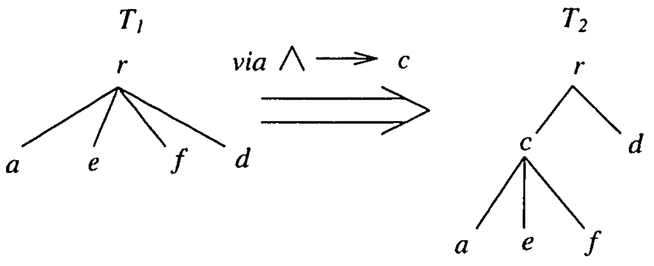
attach e and $f$to be the childern of r,
and e is the right sibling of a ,
and f is the left sibling of a.

Figure 4.2 Deletion of a node b.

$u \rightarrow v$ on the tree T_1, and is denoted by $T_1 \Rightarrow T_2$ via $u \rightarrow v$. Figures 4.1, 4.2 and 4.3 illustrate the edit operations which are self-explanatory. Each edit operation has a (user-defined) cost function γ.

Let S be a sequence of edit operations $s_1, s_2, \ldots, s_k$ applied to a tree T to generate a tree T'. Let γ be a cost function for $S = s_1, s_2, \ldots, s_k$ by letting $\gamma(S) = \sum_{i=1}^{k} \gamma(s_i)$. Then the editing distance from the tree T to tree T', denoted $dist(T, T')$, is defined to be the minimum cost of all sequences of edit operations that can transform T into T'

$$dist(T, T') = \min\{\gamma(\text{S})|\text{S is a sequence of edit operations transforming} T \text{ into } T'\}.$$



insert the node *c* to be the left sibling of *d*,
and move *a*, *e* , and *f* to be the children of *c*.

Figure 4.3 Insertion of a node *c*.

4.2 Mapping

A *Mapping* of two given tree structures is a graphical specification which specifies a sequence of edit operations corresponding to each node in two trees. Let $T[i]$ represent the ith node of the tree T according to some given order (e.g., preorder). A mapping from a tree T to another tree T' is a triple (M, T, T'), where M is any set of pairs of integer (i, j) satisfying the following conditions:

1. $1 \leq i \leq |T|$, $1 \leq j \leq |T'|$, where $|T|$ and $|T'|$ are the numbers of nodes in the tree T and T' respectively.

2. For any pair of (i_1, j_1) and (i_2, j_2) in M,

   - $i_1 = i_2$ if and only if $j_1 = j_2$ (one to one);
   - $T[i_1]$ is to the left of $T[i_2]$ if and only if $T'[j_1]$ is to the left of $T'[j_2]$ (relative position preserved); and
   - $T[i_1]$ is an ancestor of $T[i_2]$ if and only if $T'[j_1]$ is an ancestor of $T'[j_2]$ (ancestor order preserved).

A mapping from T to T' in Figure 4.4 is (1,1), (2,2), (4,3), (5,5), (6,6). A dotted line from a node u in T to a node v in T' shows that u should be changed to v if $u \neq v$, or that u remains unchanged if $u = v$. The nodes in T which are

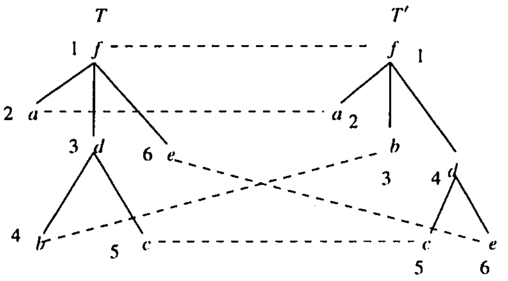


Figure 4.4 A mapping from T to T'.

not touched by a dotted line should be deleted; and the nodes in T' which are not touched by a dotted line should be inserted into T.

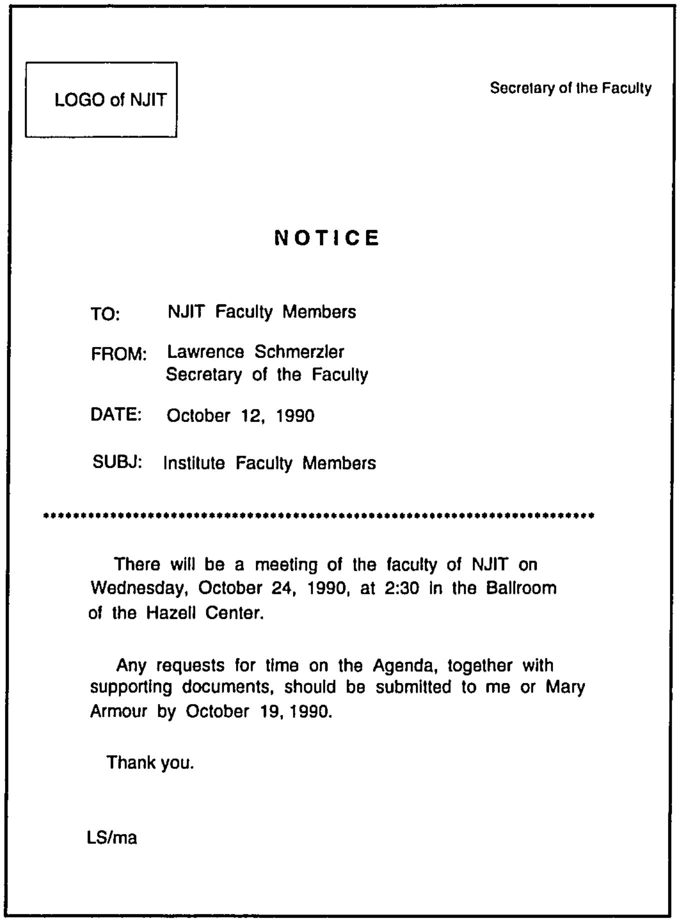
Let M be a mapping from T to T'. Let I and J be the sets of ordered nodes in T and T', respectively, which are not touched by a dotted line in M. Then we can define the cost of M:

$$\gamma(M) = \sum_{(i,j)\in M} \gamma(T[i] \rightarrow T'[j]) + \sum_{i\in I} \gamma(T[i] \rightarrow \Lambda) + \sum_{j\in J} \gamma(\Lambda \rightarrow T'[j]).$$

Given a sequence of edit operations S from T to T', it can be shown that there exits a mapping M from T to T' such that $\gamma(M) \leq \gamma(S)$. Conversely, for any mapping M, there exists a sequence of edit operations S such that $\gamma(S) = \gamma(M)$. Hence we have

$$dist(T, T') = \min\{\gamma(M) | M \text{ is a mapping from } T \text{ to } T'\}.$$

The *ATBE* algorithm is modified to find the distance between L-S Tree of a testing document and a Document Sample Tree. Given a Document Sample Tree T_D and a L-S Tree of a testing document T_{L-S} to be classified. Let M be the best mapping yielding the edit distance between T_D and T_{L-S}. A node $N_{T_D} \in T_D$ maps to a node $N_{T_{L-S}} \in T_{L-S}$. The mapping between node $N_{T_D} \in T_D$ and node $N_{T_{L-S}} \in T_{L-S}$ is an "effective mapping" if one the the following three conditions is satisfied. (1) A static node $N_{T_D} \in T_D$ is effective mapping to node $N_{T_{L-S}} \in T_{L-S}$ if the key term of node N_{T_D} is identical to the block content of $N_{T_{L-S}}$. (2) A mixed node $N_{T_D} \in T_D$ is



LOGO of NJIT

Secretary of the Faculty

NOTICE

TO: NJIT Faculty Members

FROM: Lawrence Schmerzler
Secretary of the Faculty

DATE: October 12, 1990

SUBJ: Institute Faculty Members

••

There will be a meeting of the faculty of NJIT on Wednesday, October 24, 1990, at 2:30 in the Ballroom of the Hazell Center.

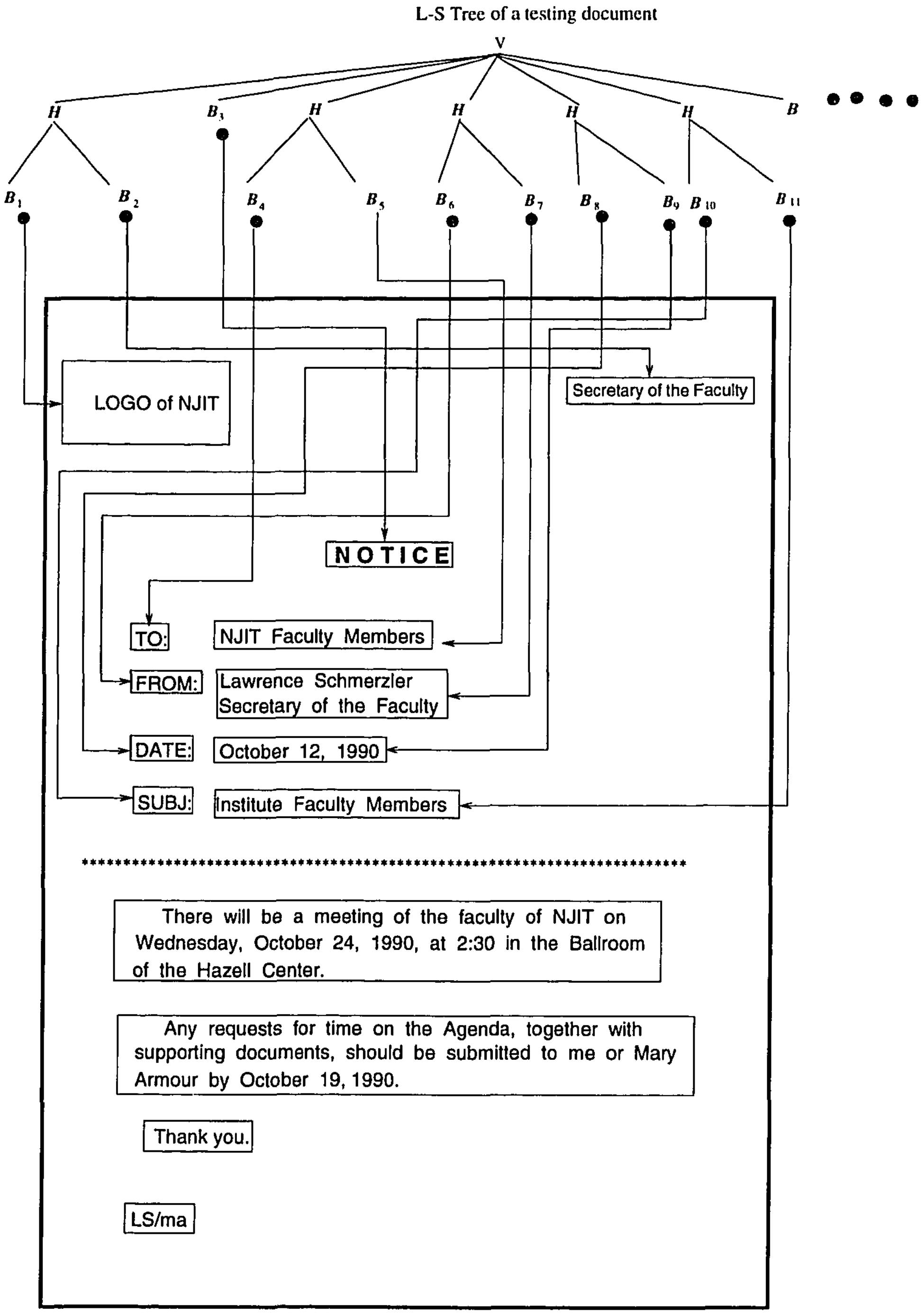
Any requests for time on the Agenda, together with supporting documents, should be submitted to me or Mary Armour by October 19, 1990.

Thank you.

LS/ma

Figure 4.5 A testing document of MEMO document type.

effective mapping to node $N_{T_{L-S}} \in T_{L-S}$ if there exists a string S in the block content of node $N_{T_{L-S}}$ such that the key term of N_{T_D} is same as S. (3) A dynamic node $N_{T_D} \in T_D$ is effective mapping to node $N_{T_{L-S}} \in T_{L-S}$ if the semantic attribute of block content of node $N_{T_{L-S}}$ is the same as the logical constituent of N_{T_D}. Consider an incoming MEMO document as shown in Figure 4.5. Its corresponding L-S Tree (which is generated by Adjacency Relation Algorithm) is shown in Figure 4.6. The mapping in Figure 4.7 illustrates the mapping from the L-S Tree of Figure 4.6 to the Document Samples Tree of Figure 1.5.



LOGO of NJIT

Secretary of the Faculty

NOTICE

TO: NJIT Faculty Members

FROM: Lawrence Schmerzler
Secretary of the Faculty

DATE: October 12, 1990

SUBJ: Institute Faculty Members

****

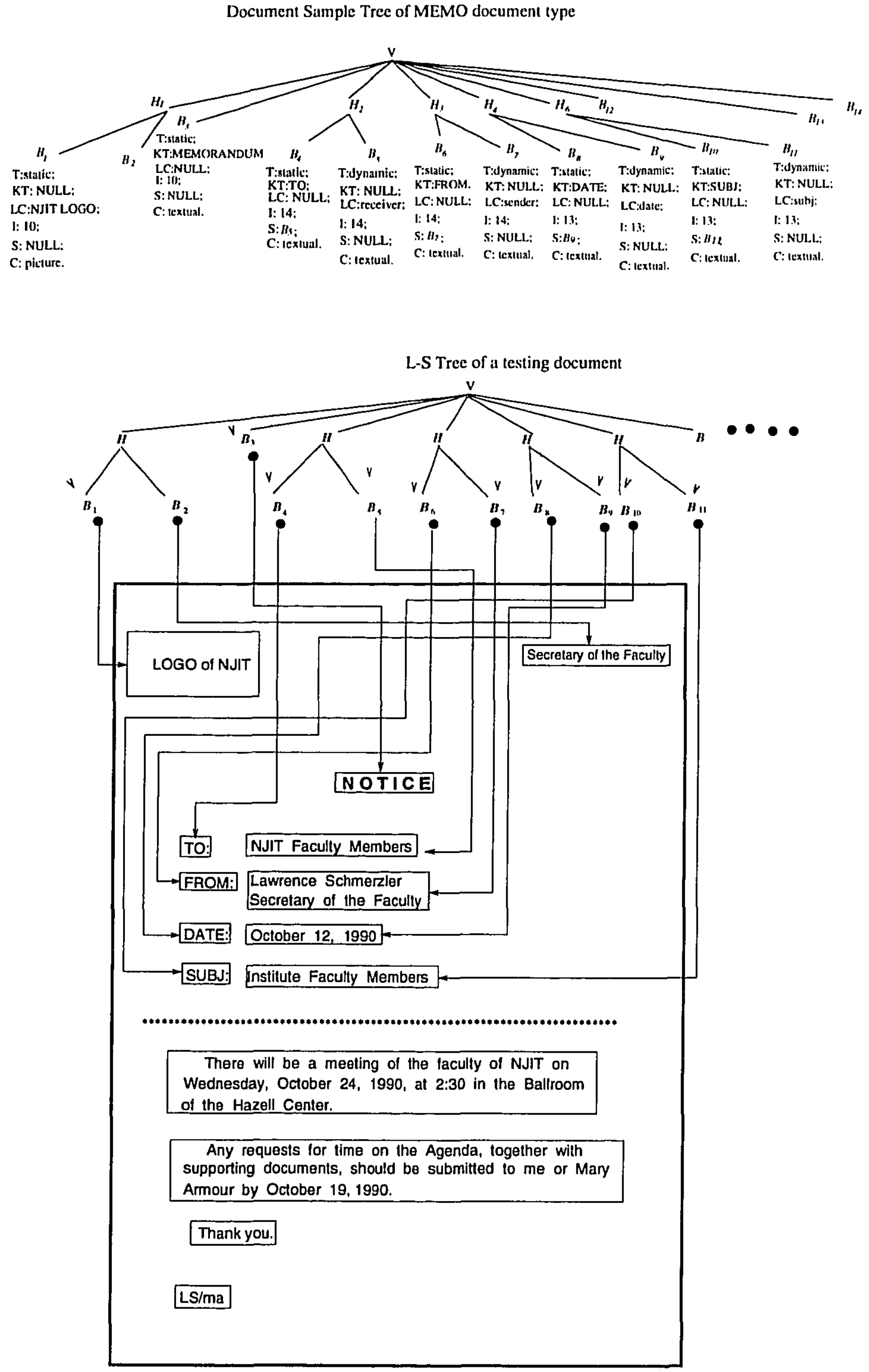
There will be a meeting of the faculty of NJIT on Wednesday, October 24, 1990, at 2:30 in the Ballroom of the Hazell Center.

Any requests for time on the Agenda, together with supporting documents, should be submitted to me or Mary Armour by October 19, 1990.

Thank you.

LS/ma

Figure 4.6 The L-S Tree of the testing document in Figure 4.5.



There will be a meeting of the faculty of NJIT on Wednesday, October 24, 1990, at 2:30 in the Ballroom of the Hazell Center.

Any requests for time on the Agenda, together with supporting documents, should be submitted to me or Mary Armour by October 19, 1990.

Figure 4.7 The best mapping between a L-S Tree and a Document Sample Tree.

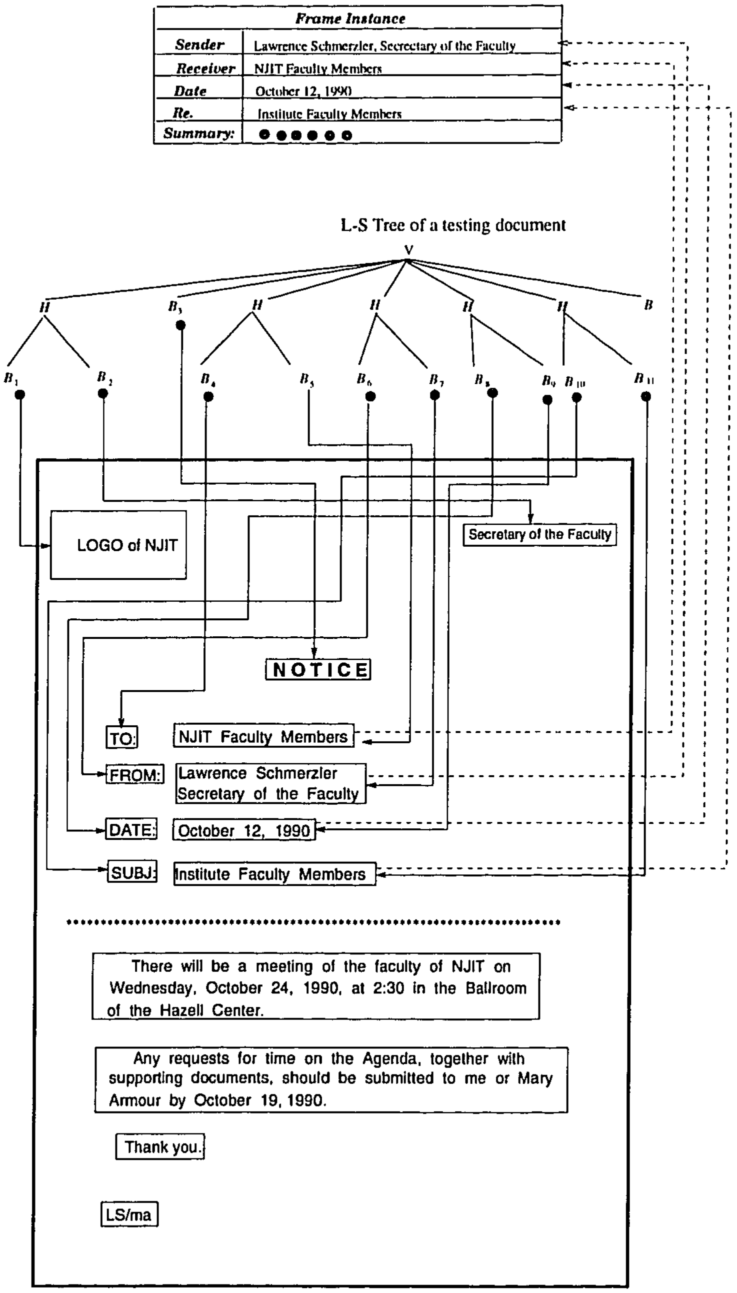
4.3 Frame Instance and Structured Blocks

The documents having similar properties are classified into a document class. Each class is associated with a type which describes the properties for the class of documents [32]. A type of MEMO class is defined as follows:

```
Define type for Memo begin
    subtype of document;
    attribute:
        Sender:string(30);
        Receiver:string(30);
        Date:string(20);
        Subject:string(30);
        Content: TEXT;
        Remark: TEXT;
```

There are five attributes in the above type definition. These attributes can be grouped into a tabular form called the *frame template.* Each document in the MEMO class is associated with a *frame instance* which is an instantiation of the frame template.

Once the type of a given document has been decided, we can extract information for the slots in its corresponding frame instance [33] of its type from the content of the blocks which are pointed to by the L-S Tree. Figure 4.8 shows certain information extracted from a testing MEMO document . The key terms are "TO", "FROM", "DATE", and "SUBJECT", which are contained in nodes B_4, B_6, B_8, and B_{10} respectively. The information is extracted for these frame instance slots directly from their right adjacent blocks which are pointed to by their right siblings in the L-S Tree.



Frame Instance	
Sender	Lawrence Schmerzler, Secretary of the Faculty
Receiver	NJIT Faculty Members
Date	October 12, 1990
Re.	Institute Faculty Members
Summary:	●●●●●●

L-S Tree of a testing document

LOGO of NJIT

Secretary of the Faculty

NOTICE

TO: NJIT Faculty Members

FROM: Lawrence Schmerzler Secretary of the Faculty

DATE: October 12, 1990

SUBJ: Institute Faculty Members

••

There will be a meeting of the faculty of NJIT on Wednesday, October 24, 1990, at 2:30 in the Ballroom of the Hazell Center.

Any requests for time on the Agenda, together with supporting documents, should be submitted to me or Mary Armour by October 19, 1990.

Thank you.

LS/ma

Figure 4.8 Information extraction from L-S Tree.

CHAPTER 5
CLASSIFICATION SYSTEM

The proposed classification system has four major components: Preprocessor, Knowledge Acquisition Tool (KAT), Classification Handler (CH) and Knowledge Base (KB). Its system flow diagram is depicted in Figure 1.1. The Preprocessor has two modules, namely, the Page Layout Generation module, and the Tree Generation module. The former is to assemble the basic blocks into a large block (the block representation) according to a number of "perceptual criteria" as discussed in Section 2.2. The latter is to transform the block representation into tree representation. Therefore, the output of the Preprocessor is a tree representation (L-S Tree) of a document with leaves pointing to their corresponding textual blocks of the original document contents or the descriptions of non-textual blocks in the document.

To identify and classify a testing document, various information is needed such as Document Type Trees and Document Sample Trees. The Knowledge Acquisition Tool (KAT) is used to acquire this information by learning from examples in the training stage. This information is also acquired when the system encounters a new type or a new format of a document during the classifying stage. The KAT consists of the Document Sample Tree Generator and the Document Type Tree Inference Engine. The Document Sample Tree Generator processes the requests and responses with the user's interactions through the User Interface to create the node contents for important blocks. The User Interface provides windows capabilities to help the user to find the important blocks in a document, and to fill in the values of attributes in their corresponding node contents. After generating all the Document Sample Trees for all the training samples of various document types, the Document Type Tree Inference Engine will induce the Document Type Trees for each document type by considering examples of its type [36, 35].

The Classification Handler is in charge of the classification process. It consists of Control Unit, Document Type Discovering module and Document Sample Tree Matching module. The Control Unit is to control the process flow in the classification process.

The last component of classification system is the Knowledge Base. It consists of Structural Knowledge Base, Node Content Generation Rules, Document Type Tree Inference Rules, Control Rules, Key Term Thesaurus, Information Extraction Rules and Frame Template Base. The Structural Knowledge includes the Document Type Trees and Document Sample Trees. The Node Content Generation Rule Base contains rules which are used to generate the node contents for important blocks. This rule base supports the KAT to build the Document Sample Trees through the user interface dialog. The Key Term Thesaurus contains key terms of various classes to perform the morphological normalization of key words in documents of the same type. The Control Rule Base is to support the Control Unit to control the process of classifying document. The Information Extraction Rule Base is to support the Information Extraction module to fill in the slots of a frame instance.

5.1 Knowledge Acquisition Tool (KAT)

The process of knowledge acquisition decides what kind of knowledge is needed, how it is used and how the knowledge can be elicited and encoded into a program [3]. The activity of building a knowledge base system may be viewed as a modeling or theory process, rather than a direct translation of knowledge which is available in the world into programs.

For the purpose of classifying documents, the process of knowledge acquisition can be summarized as three tasks:

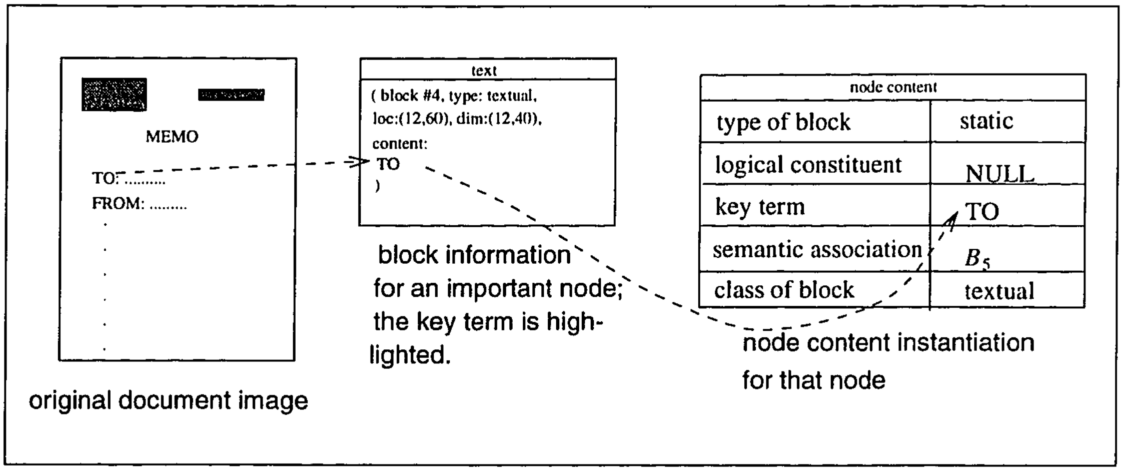
- *Identify the kind of knowledge to be acquired and how it is used.* The present work focuses on classification knowledge used to classify documents. The knowledge includes the representation of the layout structure, logical structure and major factual description of features of document contents. The layout structure is used to specify the geometrical relation among blocks. The logical structure describes the semantic structure of a document.

- *Design a knowledge representation.* The tree structures are used to represent the knowledge for classifying the document. The *H* node and *V* node of a tree describe the geometrical relations of blocks of a document layout. The node content specifies the logical structure and major factual description of features of document content.

- *Devise a technique for eliciting knowledge.* In the learning or classifying stage, the user will help the classification system to elicit information regarding the important features of a document content during the generation of the Document Sample Tree. The Document Type Tree Inference Engine infers the Document Type Trees by generalizing document sample trees. The generalization rule will be discussed in Chapter 7. The Document Sample Tree Generator can communicate with the user, using dialogs through text window, node content window, and pop up window (as depicted in Figure 5.1) provided by the user interface, and transforms the user's input regarding the document's layout structure and content into classification knowledge.

5.2 Document Sample Tree Generator

Given the L-S Tree of a document, the Document Sample Tree Generator will output its corresponding Document Sample Tree. As shown in Figure 5.1, the *text* window describes the block information and the content of an important block, and a *node content* window describes the node content of its corresponding block. Given the window of an original document image, the user is requested to fill in the values of



text

(block #4, type: textual,
loc:(12,60), dim:(12,40),
content:
TO
)

block information for an important node; the key term is highlighted.

node content	
type of block	static
logical constituent	NULL
key term	TO
semantic association	B_5
class of block	textual

node content instantiation for that node

Figure 5.1 A screen layout of KAT for a MEMO document.

type of block, *logical constituent*, and *semantic association* for each of the important blocks which are selected by users. In Figure 5.1, KAT displays a text window and a node content window for the block B_4 containing "TO", which is considered to be an important block. Then, the user copies the word "TO" from the text window to the appropriate slot of the node content window as the value of key term. The user will also fill the slots of *logical constituent* and *semantic association* with *NULL* and B_5 respectively. The classification system automatically fills in the *class of block* with "textual". This completes the knowledge elicitation process for this block. The Node Content Generating Rule Base contains the rules that support the KAT to decide which blocks are the important blocks in the original document image, and to pop up the appropriate text or nontext windows for the blocks.

The example of a memo document is given in Figure 1.5. The classification system will call the user's attention to the blocks B_1, B_2, B_3, B_4, B_5, B_6, B_7 B_8, B_9, B_{10}, B_{11}, B_{14}, and B_{15}, by displaying them on the screen. By browsing through all these blocks the user fills out the tables for the important blocks, B_1,B_3, B_4, B_5, B_6, B_7 B_8, B_9, B_{10}, B_{11} and disregards the blocks B_2, B_{14}, and B_{15} which contain no important information for classification.

A sample rule for creating a node content is given as follows:

Rule: if the class of a block = "textual", then the user keys in the values of *type of class, logical constituent*, and *semantic association*, and copies the high-lighted text to the value of *key term.*

The value of *class of block* is provided by OCR (Optical Character Recognition) system.

5.3 Document Type Tree Inference Engine

The Document Type Tree Inference Engine employs inductive learning approach to generating Document Type Trees from Document Sample Trees of each document type. The Document Type Trees allow that a small set of trees is possibly used to identify the type of a document during the document type classification process. Once the type of a document is recognized, the Document Sample Trees of the type are used to do the format recognition and information extraction by searching a closer match of the L-S Tree with its segmented contents of the document and one of the Document Sample Trees (including its node contents).

Inductive learning [4, 5, 14, 17, 19, 20, 18] is a process of acquiring knowledge by drawing inductive inferences from facts provided by experts, users, and others. Such a process involves operations of transforming, generalizing, modifying and refining knowledge representations. During the learning process, the Document Sample Trees of each document type are the training examples for the Document Type Tree Inference Engine. The Document Type Tree Inference Engine can acquire a description of a class of Document Sample Trees of the same document type by generalizing user-provided training examples (positive example). The Document Sample Trees which belong to the same class are called the positive examples and the rest of Document Sample Trees are considered as negative examples with respect to the class. This approach is called *learning from examples.*

5.3.1 Observational Statements

Observational statements [6, 36] are used to specify facts (in which each consists of training events of the same document type) found in Document Sample Trees. A document sample is first transformed into a Document Sample Tree, from which a set of observational statements containing the *path* and *node content* pairs can be derived by applying the background knowledge in the KAT (Knowledge Acquisition Tool). The set of observational statements of a training Document Sample Tree is referred as a training event of the tree. The definition of a path is as follows:

Definition 5.1 A path of node N, denoted as path(N), is a character string containing labels and numbers as follows:

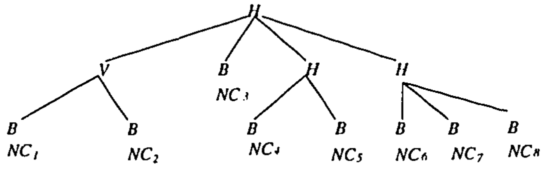
$$path(N) = (Label)_1 n_1 (Label)_2 n_2 \ldots (Label)_j n_j (Label)_{j+1},$$

where N could be an intermediate node or a leaf; j is the depth of node N; $(Label)$ is the label associated with each node such as H, V or B; $(Label)_k$, $1 \leq k \leq j+1$, is the n_{k-1}th child of $(Label)_{k-1}$.

The format of observational statements can be represented in a tabular form as shown in Table 5.1. The observational statements inferred from a tree example are shown in Figure 5.2. NC_n's, $1 \leq n \leq 8$, stand for node contents containing important information of their corresponding blocks.

Table 5.1 The format of observational statements.

Path	*Node content*
path(node of N_1)	node content of N_1
path(node of N_2)	node content of N_2
...	...
path(node of N_n)	node content of N_n



Path	Node content
H1V1B	NC_1
H1V2B	NC_2
H1V	NULL
H2B	NC_3
H3H1B	NC_4
H3H2B	NC_5
H3H	NULL
H4H1B	NC_6
H4H2B	NC_7
H4H3B	NC_8
H4H	NULL

Figure 5.2 The training event of a tree example.

5.3.2 Inductive Paradigm

An inference process is to find out plausible assertions that can explain the training sample trees. We use these assertions to classify the new events such as input testing documents in the classification stage. The inductive inference process attempts to derive a complete and consistent description of a concept (also referred as a document type) from a fact which is the set of training events (sets of observational statements) of training sample trees of the type. In our case, the training sample trees are the Document Sample Trees of different document types, and the description of a concept is the Document Type Trees of a document type. These Document Type Trees are the generalization of Document Sample Trees of the same document type. The inputs of inference process are facts (training events of document types) F and background knowledge.

Training events are derived from the training sample trees that represent the specific knowledge about types and formats of documents. A fact describes a document type represented by various training events, as in Figure 5.3. And the facts F can be denoted as

$$F: \ \{e_{i,1}, \ldots, e_{i,n_i} \overset{c}{\Longrightarrow} K_i \},$$

where $1 \leq i$ is an unique type id, and $e_{i,j}, (1 \leq j \leq n)$, is a training event defining the jth sample tree of the document type K_i.

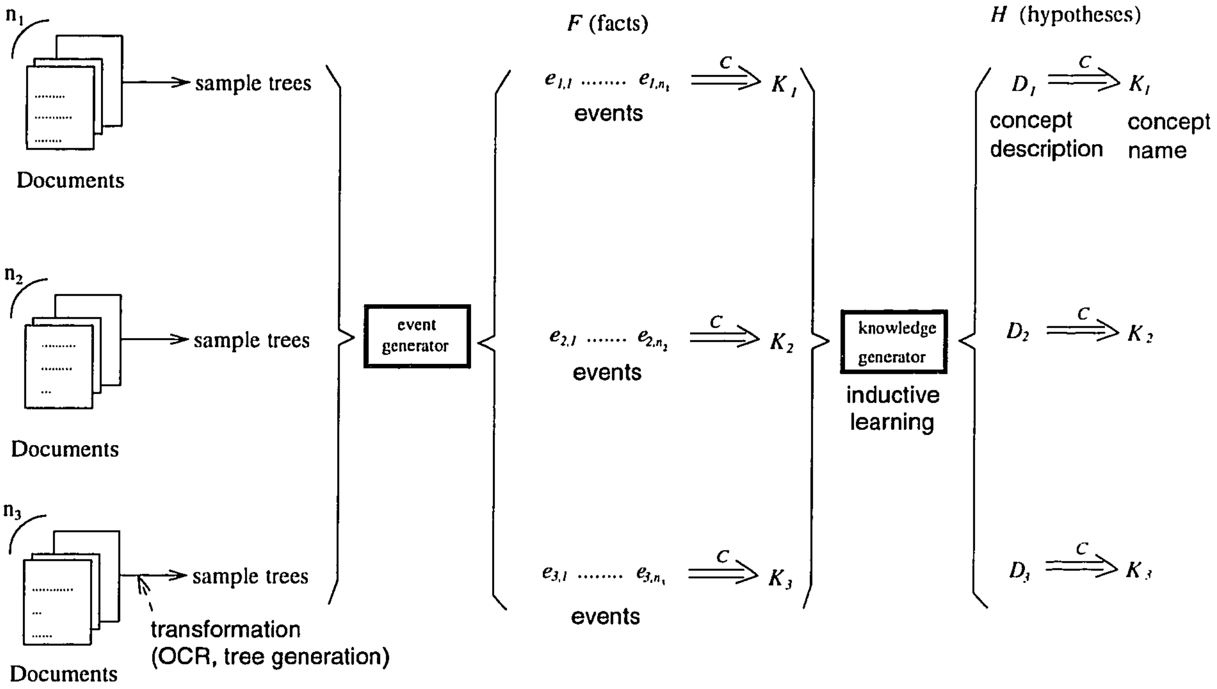
Background knowledge includes the problem-related domain knowledge for extracting the facts of incoming samples. This also includes the definitions and assumptions that are posed on the observational statements and generated hypotheses.

The output of inductive inference process is the inductive assertions (called hypotheses) H generated by applying the generalization rules and background knowledge on F, that is $F \lhd H$. H can be defined as a set of concept recognition rules:

$$H : \{D_i \overset{c}{\Longrightarrow} K_i\},$$

where $i \in I$ and D_i is a concept description of document type K_i.

Intuitively, let's consider two training sample trees *tree*1 and *tree*2 of document type K_i. The sample tree *tree*1 contains four key terms "TO", "FROM", "DATE", and "SUBJ", and the sample tree *tree*2 contains three key terms "TO", "FROM", and "DATE". Therefore their corresponding training events are $e_{i,1}$ and $e_{i,2}$. By applying the generalization rules (of the inductive inference process), only three key terms "TO", "FROM", and "DATE" are selected to be the concept description D_i in the hypotheses H to imply the facts F. In Figure 5.3, we assume that there are three possible types of documents K_1, K_2, and K_3 such as letter, memo, and journal. During the inductive learning stage, the user preclassifies all the training documents, and let every $e_{i,j}$ imply only one document type K_i.



n_1 , n_2 and n_3 represent the number of samples for document type 1, 2 and 3 respectively.

$D_1 \longrightarrow (\textit{Document Type Trees})_1$

$D_2 \longrightarrow (\textit{Document Type Trees})_2$

$D_3 \longrightarrow (\textit{Document Type Trees})_3$

Figure 5.3 Inductive learning process for Document Type Trees.

CHAPTER 6

FINDING COMMON SUBSTRUCTURES FROM SEGMENTED DOCUMENTS

A document is classified if there is a Document Type Tree to be a substructure of its L-S Tree. Then, the exact format of the document can be found by searching the closer match of the L-S Tree with its segmented contents of the document and one of the Document Sample Trees (including its node contents) represented by the identified Document Type Tree. In previous chapters, we addressed document page layout segmentation (namely, dividing a document page into several segments, which are in turn, divided into smaller segments), and then the formation of a L-S Tree for it. We also presented the construction of Document Sample Trees. The Document Sample Trees of the same document type can be generalized to a fewer Document Type Trees. That is, the Document Type Trees can be considered as the the Largest Common Substructures of the Document Sample Trees of the same document type.

In this chapter, we will investigate the problem of finding the Largest Common Structures between Document Sample Trees, taking the corresponding segmented document samples into consideration. The Nested Segmentation Algorithm is adopted in the generation of L-S Tree.

6.1 Longest Common Subsequence

Let $N = (NC_{1,1}, NC_{1,2}, \cdots, NC_{1,r})$, and $M = (NC_{2,1}, NC_{2,2}, \cdots, NC_{2,s})$ be two sets of node contents. The longest common subsequence between two sets of node contents N and M, denoted as $LCS(N, M)$, is defined as follows: There exists $N' \subseteq N$ and $M' \subseteq M$, and $N' = (NC_{1,m_1}, NC_{1,m_2}, \cdots, NC_{1,m_t})$, and $M' = (NC_{2,n_1}, NC_{2,n_2}, \cdots, NC_{2,n_t})$ such that t is a maximum and $NC_{1,m_1} = NC_{2,n_1}$, $NC_{1,m_2} = NC_{2,n_2}, \cdots$, $NC_{1,m_t} = NC_{2,n_t}$, and $m_1 < m_2$, $m_2 < m_3$, $\cdots$, $m_{t-1} < m_t$, and $n_1 < n_2$, $n_2 < n_3$, $\cdots$, $n_{t-1} < n_t$. That is, a subsequence of N (or M) is obtained by removing zero or more, but not necessarily contiguous, nodes from N (or M). Then, the $LCS(N, M)$ is a longest sequence that is a subsequence of both N and M [1].

6.2 Edit Operations of Document Segments

During the comparison of two documents, the appearing orders of their segments are significant. Two basic segments are equivalent if their block contents are identical. Two composite segments are equal if they contain the same types of H or V cuts. There are four types of edit operations: *relocate*, *change_block_content*, *delete_segment*, and *insert_segment*. Let D_2 be the document that results from the application of an edit operation to document D_1. A *relocate* operation is represented as $(u \rightarrow v)$, where u, v are either H, V or V, H. This operation transforms D_1 to D_2 by reconstructing segments which are separated by u cut within a composite segment x of D_1 to be v cut within x of D_2. That is, the left-to-right (or top-to-bottom) ordering of segments in the segment x of D_1 will be changed to top-to-bottom (or left-to-right) ordering of segments in the segment x of D_2. Figure 6.1 illustrates a *relocate* operation, where $H_{i,j}$ or $V_{i,j}$ stands for the jth cut at the level i, and $d[k]$ stands for the kth segment.

A *change_block_content* operation can be represented by $(u \rightarrow v)$, where u, v are the contents of the basic segments in D_1 and D_2. Figure 6.2 depicts this operation.

The equivalent tree edit operations of *relocate* and *change_block_content* are *relabel* for an intermediate node (from H to V, or vice versa) and *relabel* for a leaf node (from one regradless of its node content to one with its node content) respectively. The cost of *relocating* segments within a composite segment x (or *relabelling* for an intermediate node i) is the total number of segments within x (or the total number of immediate children of node i); and the cost for *change_block_content* (or *relabelling* a leaf node) is 1.

A *delete_segment* is represented as $(u \rightarrow v)$, where u is a segment in a document and v is the null segment (Λ). If u is a composite segment, there are two cases as follows: (*case 1*) if the type of cut at the level $j-1$ in the segment which contains segment u is not the same as the type of cut within the segment u at level j, a *relocate* operation is performed for u before u is removed, and then assigns all the segments within the segment u at the level j to the segment at the level $j-1$; (*case 2*) if the type of cut at the level $j-1$ is the same as the type of cut within the segment u at level j, this operation just removes the segment u and assigns all the segments within the segment u at the level j to the segment at the level $j-1$. If segment u is a basic segment, then remove simply the content of this basic block. Figure 6.1 describes a *delete_segment* operation. The *delete_segment* is equivalent to the tree edit operation of *delete*. The cost for (*case 1*) is (number of segments within u) + 1 and the cost for (*case 2*) is 1. The cost of *delete_segment* for a basic segment is 1. In the tree edit operations, (*case 1*) corresponds to *delete* for a non-leaf node N whose label is not identical to that of its parent. The cost of this case is (number of immediate children of node N) + 1. (*case 2*) corresponds to *delete* for a non-leaf node N whose label is identical to that of its parent and the cost for this case is 1. The cost of *delete* for a leaf is 1. The *delete* operation is not allowed to applied on the root of a tree.

An *insert_segment* can be represented by $(u \rightarrow v)$, where u is a null segment (Λ) and v is a segment. The operation of *insert_segment* will create a segment v at level j to enclose a set of consecutive segments at level j with type of cut c_{new} and change their level from j to $j+1$ if v is a composite segment; or the operation will create a basic segment directly if v is a basic segment. For a composite segment v, there are two cases as follows: (*case 1*) if the new cut c_{new} within v is not the same as

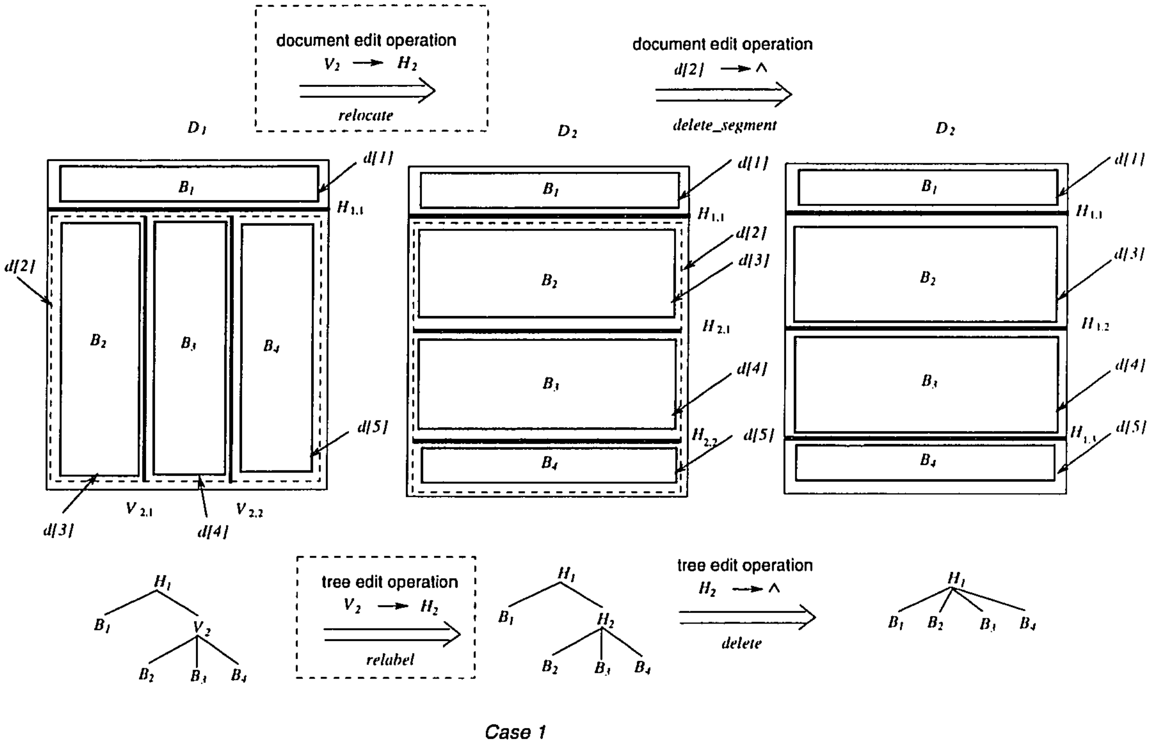


Figure 6.1 An example of a *delete_segment* (*relocate*) operation and its equivalent tree edit operation *delete* (*relabel*).

the original cut $c_{original}$, then a *relocate* operation ($c_{original} \dashrightarrow c_{new}$) is followed after inserting the segment v; (*case 2*) if the new cut c_{new} is the same as the original cut $c_{original}$ at level j, then inserting the segment v only. Figure 6.2 depicts the operation of *insert_segment*. The cost of *insert_segment* for a basic segment is 1. The cost for (*case 1*) is (number of segments in v) + 1, and the cost for (*case 2*) is 1. The *insert_segment* operation is equivalent to the tree edit operation of *insert*. In the tree edit operations, (*case 1*) corresponds to *insert* a non-leaf node N whose label is not identical to that of its parent. The cost of this case is (number of immediate children of node N) + 1. (*case 2*) corresponds to *inserting* a non-leaf node N whose label is identical to that of its parent. The cost for this case is 1. The cost of *inserting* a leaf is 1.

Let γ be the cost function as we discuss above that assigns each edit operation $u \dashrightarrow v$ a nonnegative real number $\gamma(u \dashrightarrow v)$. γ can be extended to a sequence of edit operations $S = s_1, s_2, \ldots, s_m$ by letting $\gamma(S) = \sum_{i=1}^{m} \gamma(s_i)$. The editing distance from document D_1 to document D_2, denoted as $dist(D_1, D_2)$, is defined to be the minimum cost of all sequences of edit operations which transform D_1 to D_2 as:

$$dist(D_1, D_2) = \min \{\gamma(S) | S \text{ is a sequence of edit operations transforming } D_1 \text{ to } D_2\}.$$

6.3 Mappings of Two Segmented Documents

The mapping of two documents is a graphical specification of which a sequence of edit operations can apply to each segment in two documents. The mapping in Figure 6.3 shows a way to transform D_1 to D_2. It corresponds to a sequence of edit operations: (*delete_segment* for $d[2]$ in D_1, *insert_segment* for $d[3]$ to enclose $d[4]$ and $d[5]$ in D_2).

Given two documents D_1 and D_2 which consist of segments $d_1[1], d_1[1], \ldots, d_1[|D_1|]$ and segments $d_2[1], d_2[1], \ldots, d_2[|D_2|]$, respectively, a mapping from D_1 to D_2 is a

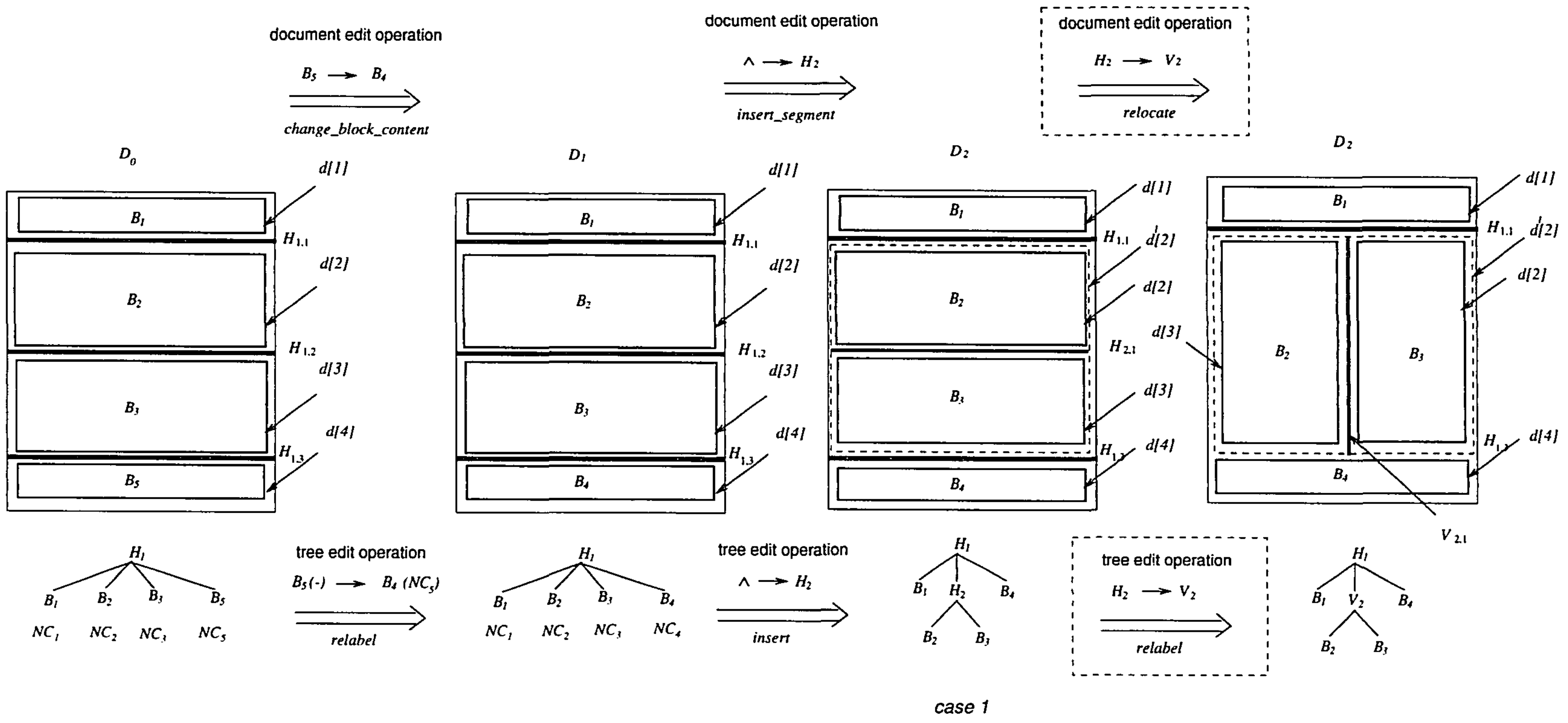


Figure 6.2 An example of a *insert_segment* (*change_block_content*) operation and its equivalent tree edit operation *insert* (*relabel*).

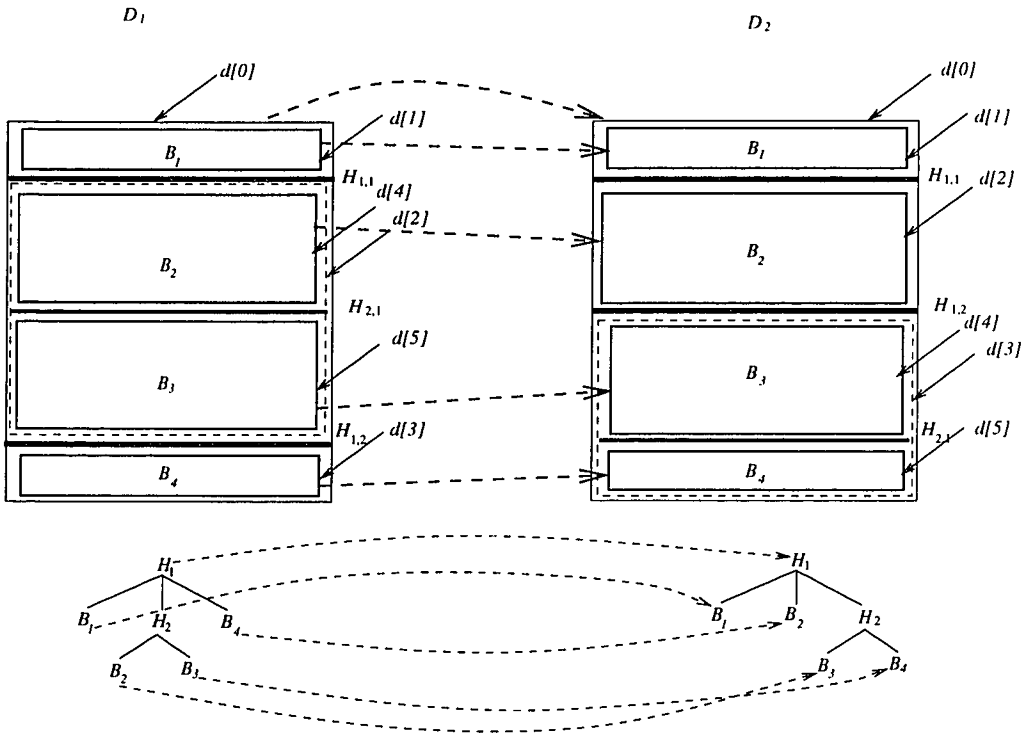


Figure 6.3 Mappings of two segmented documents and their corresponding trees.

triple (M_e, D_1, D_2) where M_e is any set of ordered pairs of integers (i, j) satisfying the following conditions:

1. $1 \leq i \leq |D_1|$ and $1 \leq j \leq |D_2|$, where $|D_1|$ and $|D_2|$ are the numbers of segments in documents D_1 and D_2, respectively.

2. For any pair if (i_1, j_1) and (i_2, j_2) in M_e,

   - $i_1 = i_2$ if and only if $j_1 = j_2$ (one-to-one);

   - $d_1[i_1]$ is on the top or to the left of $d_1[i_2]$ if and only if $d_2[j_1]$ is on the top or to the left of $d_2[j_2]$ (relative position preserved);

   - $d_1[i_1]$ is contained in $d_1[i_2]$ if and only if $d_2[j_1]$ is contained in $d_2[j_2]$ (composition relation preserved).

Let M be a mapping from D_1 to D_2. Let I and J be the sets of segments in D_1 and D_2, respectively, not touched by any dotted line in M. Then we can define the cost of M:

$$\gamma(M) = \sum_{(i,j)\in M} \gamma(d_1[i] \to d_2[j]) + \sum_{i\in I} \gamma(d_1[i] \to \Lambda) + \sum_{j\in J} \gamma(\Lambda \to d_2[j]).$$

Given a sequence of edit operations S, it can be shown that there exists a mapping M from D_1 to D_2 such that $\gamma(M) \leq \gamma(S)$; conversely, for any mapping M, there exists a sequence of edit operations S such that $\gamma(M) = \gamma(S)$.

Hence, we have

$$dist(D_1, D_2) = \min \{\gamma(M) | M \text{ is a mapping from } D_1 \text{ to } D_2\}.$$

For example, the mapping in Figure 6.3 is {(0, 0), (1, 1), (4, 2), (5, 4), (3, 5)}, and the $dist(D_1, D_2) = 2$, since the minimum cost of mapping involves the deletion of $d[2]$ in D_1 and insertion of $d[3]$ as in D_2.

6.4 Largest Common Subregion of Segmented Documents

A *sub_document* $D[i]$ of a document D represents a set of segments at any levels within the segment $d[i]$. A *subregion* of D is a portion of a document layout with some sub_documents removed. The size of a document D, denoted as $|D|$, is the total number of segments in D at any levels. The operation of removing sub_document $D[i]$ means deleting segment $d[i]$ and all the segments contained in the segment $d[i]$. A set of segments S_c in D is said to be a set of consistent sub_documents removal in D, if (1) $d[i] \in S_c$ implies that $1 \leq i \leq |D|$, and (2) $d[i], d[j] \in S_c$ implies that neither is within the other in D. We use $Remove(D, S_c)$ to represent the document layout D with all sub_documents in S_c removed. Let $subportion(D)$ be the set of all possible sets of consistent sub_document removals in D. Given two documents D_1 and D_2, the *Largest Common Subregion* between D_1 and D_2 can be found by locating the $Remove(D_1, S_{c1})$ and $Remove(D_2, S_{c2})$ such that

$$\max\{|Remove(D_1, S_{c1})| + |Remove(D_2, S_{c2})|\} \text{ where}$$

$$dist(Remove(D_1, S_{c1}), Remove(D_2, S_{c2})) = 0,$$

$$S_{c1} \in (subportion(D_1)) \text{ and}$$

$$S_{c2} \in (subportion(D_2)).$$

6.5 Largest Common Substructure of Trees

A substructure of an ordered labelled tree T is a tree with certain subtrees removed from T. Given two ordered labelled trees T_1 and T_2, the algorithm of *Largest Common Substructure* (*LCSstr*) of T_1 and T_2, denoted as $LCSstr(T_1, T_2)$, is to find a substructure S_1 of T_1, and a substructure S_2 of T_2, such that the distance of S_1 and S_2 is 0 and there does not exist any other substructure S_1' of T_1 and S_2' of T_2 such that the distance between S_1' and S_2' is 0 and the total size of S_1' and S_2' is greater than the total size of S_1 and S_2 [25]. It is still possible that there exists some other substructures S_1'' of T_1 and S_2'' of T_2 such that the distance between S_1'' and S_2'' is 0 and the total size S_1'' and S_2'' is equal to the total size of S_1 and S_1.

Let $T[i]$ stand for the subtree rooted at node $t[i]$. The operation of cutting at the node $t[i]$ removes $T[i]$ from T. A set of nodes $S_{node} \in T$ is said to be a set of consistent subtree cuts in T if $t[i], t[j] \in S_{node}$, $1 \leq i, j \leq |T|$ and neither one is an ancestor of the other in T. Intuitively, S_{node} contains all the roots of the removed subtrees in T. Let $Cut(T, S_{node})$ represent the tree T with subtree removed at all nodes in S_{node}, and let $Subtree(T)$ be the set of all the possible sets of consistent subtree cuts in T. To find Largest Common Substructures of trees T_1 and T_2, we first locate the $Cut(T_1, S_{node,1})$ $Cut(T_2, S_{node,2})$ and then calculate $\max\{|Cut(T_1, S_{node,1})| + |Cut(T_2, S_{node,2}|\}$ where $dist(Cut(T_1, S_{node,1}), Cut(T_2, S_{node,2})) = 0$, $S_{node,1} \in Subtree(T_1)$ and $S_{node,2} \in Subtree(T_2)$. In inductive learning process, every pair of Document Sample Trees T_1

and T_2 will be generalized to discover the $LCSstr(T_1, T_2)$ with distance 0. Therefore, $Cut(T_1, S_1)$ and $Cut(T_2, S_2)$ are identical. The original algorithm of $LCSstr$ only locates the first substructure of $\max\{|Cut(T_1, S_1)| + |Cut(T_2, S_2|\}$ [25]. A *modified* $LCSstr$ algorithm is proposed to discover some $LCSstr$'s of T_1 and T_2, which will be discussed in generalization Rule 7.5 in Section 7.5. In Figure 6.4, T_{LCSstr} is an example of the $LCSstr$'s of T_1 and T_2. But the subsequences (B_3, B_4, B_6) and (B_3, B_4, B_5) are the LCS's of the nodes (B_3, B_4, B_5, B_6) of T_1 and (B_3, B_4, B_6, B_5) of T_2. We discover another $LCSstr$ of T_1 and T_2 by replacing the nodes (B_3, B_4, B_6) with (B_3, B_4, B_5) in the T_{LCSstr}'s shown in Figure 6.4.

6.6 Relation of Largest Common Substructure and Largest Common Subregion

The Largest Common Substructure algorithm only takes trees as input and locates the shared common substructure of maximum size without taking the corresponding document segments into account. This may result in a false Largest Common Substructure between two segmented document samples. For example, consider two segmented sample documents D_1 and D_2 and their corresponding Document Sample Trees T_1 and T_2 in Figure 6.4. $LCSstr(T_1, T_2)$ is T_{LCSstr}. The root H in T_{LCSstr} represents H_2 node in T_1 and H_1 node in T_2. H_2 cut in D_1 divides sub_document $D_1[2]$ into half, and H_1 cut in D_2 divides the whole document D_2 into three smaller segments. They have completely different geometrical meanings of segmentations. We use the following preprocesses to eliminate such an error, and to ensure that there are one-to-one correspondences between Largest Common Substructure and Largest Common Subregion.

Given two segmented sample documents and their Document Sample Trees, we first preprocess them by removing the non-basic leaf nodes from Document Sample Trees and removing the segments which are not basic segments or do not contain basic

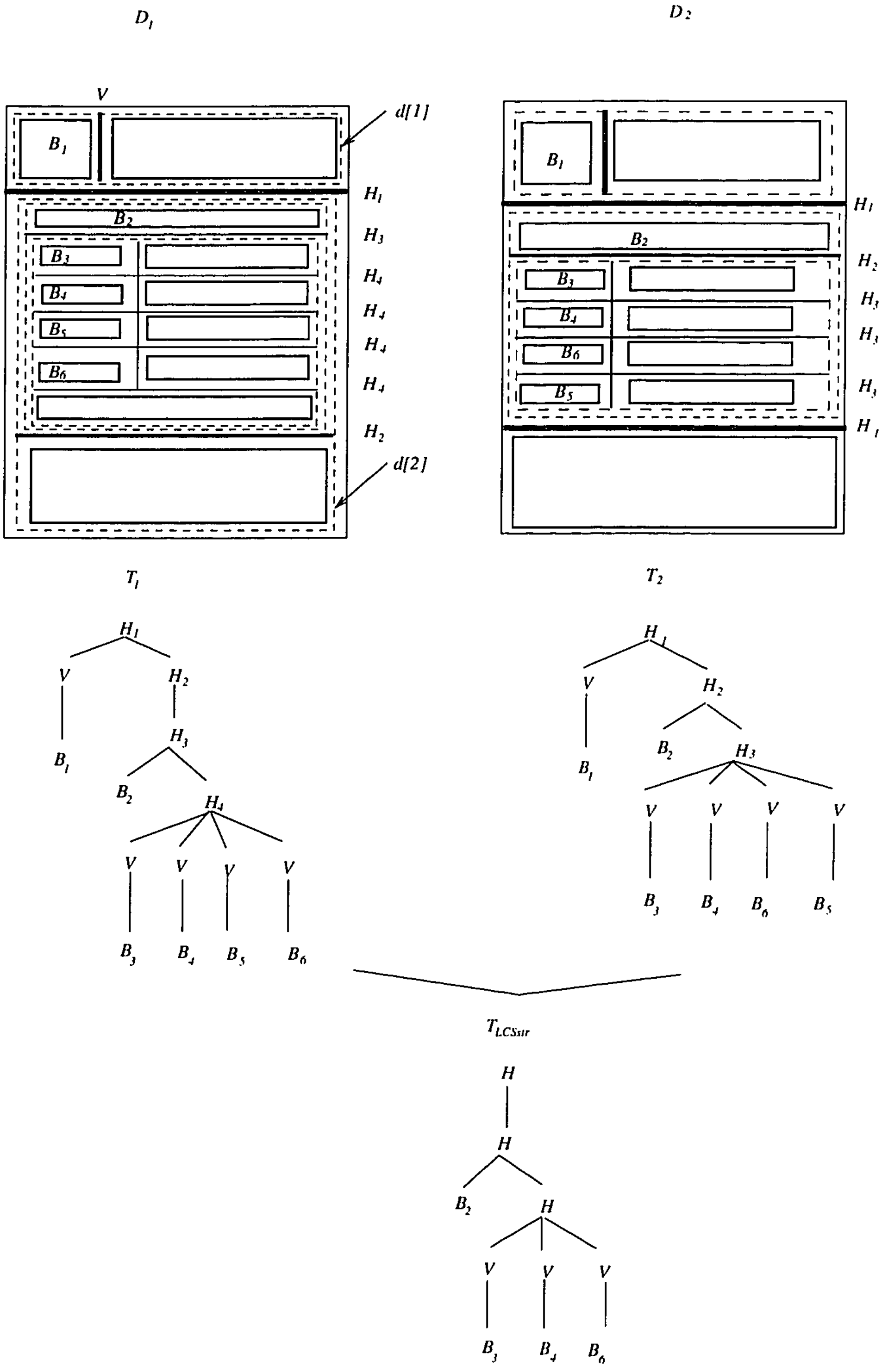


Figure 6.4 Segmented documents D_1 and D_2 and their Document Sample Trees T_1 and T_2.

segments at any lower level. This is called *preprocess 1.* For example, a document sample $D_{sample1}$ as shown in Figure 6.5 has its Document Sample Tree $T_{sample1}$ on its right. After preprocess 1, all the leaf nodes which correspond to the blocks containing no key terms in $T_{sample1}$ are deleted. Now, $T_{sample1}$ becomes $T'_{sample1}$. Similarly, all the basic segments containing no key terms in $D_{sample1}$ are removed. Now, $D_{sample1}$ becomes $D'_{sample1}$. We observed that the segment $d[2]$ in $D'_{sample1}$ contains only one segment $d[3]$. Therefore, segment $d[2]$ and its H cut are redundant and should be removed. In its corresponding sample tree, H_2 node is deleted, and the subtree of H_3 becomes the child of H_1. Now, $D'_{sample1}$ becomes $D''_{sample1}$. This is called preprocess 2 which eliminates the mapping error during the discovery of Largest Common Substructure of two trees for Largest Common Subregion. The algorithm of preprocess 2 is described as follows: if a H node has only a single H child node, or if a V node has only a single V child node as shown in Figure 6.6, then we can remove this node from the Document Sample Tree. Equivalently, in a segmented document, if a segment $d[i]$ contains only one segment $d[k]$ which is not a basic segment, then we remove the segment $d[i]$ and all cuts within $d[i]$.

Lemma 5.1 *Given two segmented documents D'_1 and D'_2 and their corresponding Document Sample Trees T'_1 and T'_2, the preprocessed segmented documents are D_1 and D_2 and the preprocessed Document Sample Trees are T_1 and T_2. After preprocessing (using preprocess 1 and preprocess 2 as discussed above) the tree structure of Largest Common Subregion of D_1 and D_2 (LCSreg(D_1 ,D_2)) is the Largest Common Substructure of T_1 and T_2 (LCSstr(T_1, T_2)).*

Proof. The algorithm of discovering the Largest Common Substructure of two trees T_1 and T_2, which is T_{LCSstr}, has been proved in [25]. The definitions of *edit* operations and *remove* operations of Largest Common Subregion in document D are equivalent to the definitions of *edit* operations and *cut* operations of Largest Common Substructure in tree T, stated as follows:

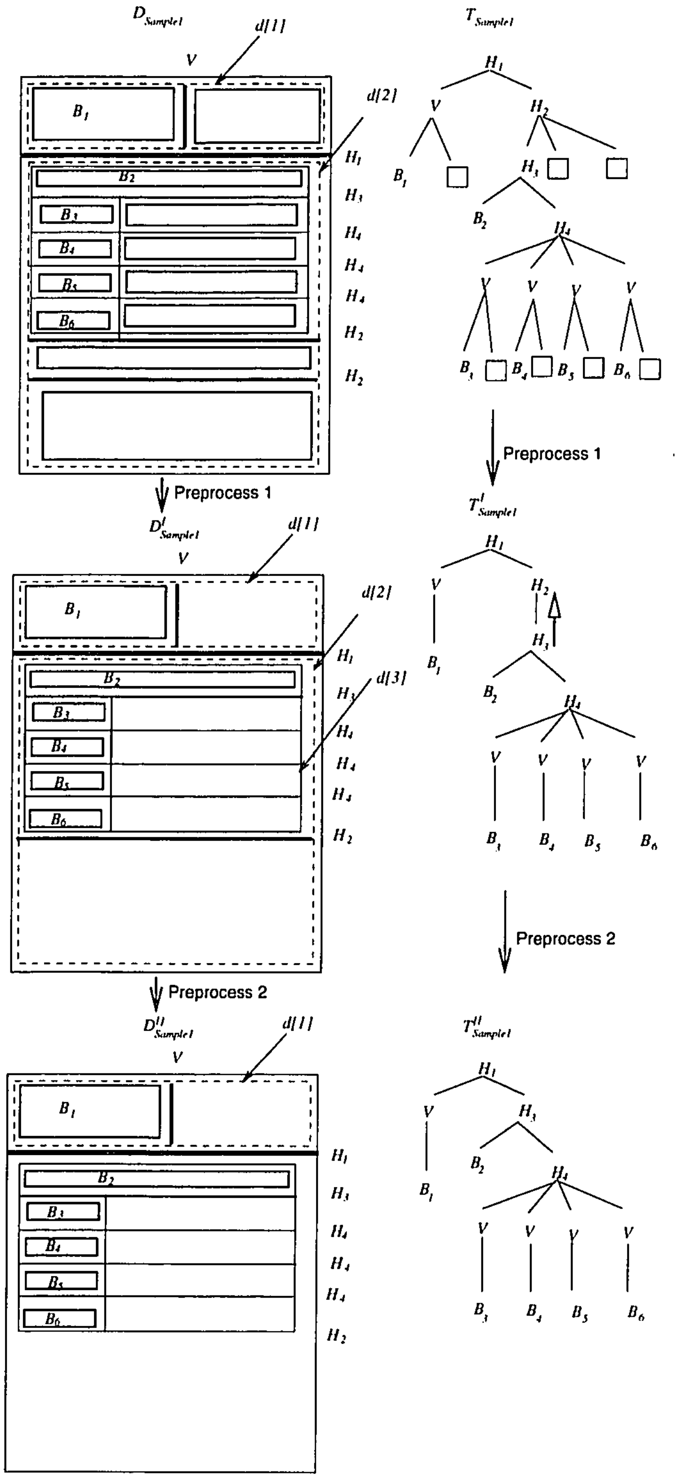


Figure 6.5 An example of preprocesses for segmented document and its Document Sample Tree.

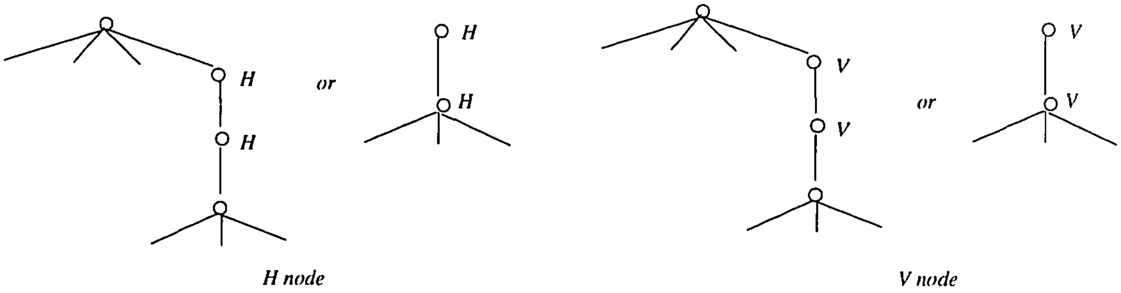


Figure 6.6 The H (or V) node has only a single H (or V) child node.

1. Each edit operation in D has one and only one equivalent edit operation in T as described in Section 6.2.

2. Each segment in D has a one-to-one mapping to a node in T.

3. If a composite segment $d[j]$ in D maps to a non-leaf node $t[i]$ (i.e., either a H or V node) in T, all the segments in sub_document $D[j]$ have one-to-one mapping relations to the nodes in subtree $T[i]$.

4. The operation $remove(D, S_{doc})$ which represents document D with a set of consistent sub_documents S_{doc} removed is equivalent to $cut(T, S_{tree})$ in tree T with a set of consistent subtrees S_{tree} removed.

Since every preprocessed segmented document corresponds to one preprocessed Document Sample Tree, the Largest Common Subregion also corresponds to a unique tree T_{LCSreg}. According to the definition of mapping between two segmented documents, the mapping from D_1 to D_2 is the same as the mapping from T_1 to T_2. The tree structure of $LCSreg$ will be T_{LCSreg} which is equal to T_{LCSstr}. If we can find a $LCSreg'$ of D_1 and D_2, such that $|LCSreg'| \geq |LCSreg|$, then the tree structure of $LCSreg'$, which is $T_{LCSreg'}$, has the relations $|T_{LCSreg'}| \geq |T_{LCSreg}|$ and $|T_{LCSreg'}| \geq |T_{LCSstr}|$. The latter relation contradicts the fact that T_{LCSstr} is the Largest Common Substructure. This concludes the proof.

CHAPTER 7

GENERALIZING DOCUMENT SAMPLE TREES

In Section 5.3, we presented the Document Type Tree Inference Engine, which employs inductive learning approach to generating a fewer Document Type Trees from a large number of Document Sample Trees of a document type, using a set of generalization rules, which will be presented in this section.

7.1 Importance of a Tree

The importance of a node N_i, $Importance_{node}(N_i)$ is defined in Section 3.3. Intuitively, the importance of a node content, say, containing the key term "MEMO" in a Document Sample Tree of MEMO document type, is measured by the number of occurrences of this term appeared in the set of Document Sample Trees of MEMO type. The importance of a tree T, $Importance_{tree}(T)$, is defined as

$$Importance_{tree}(T) = \frac{\sum_{i=1}^{n}(Importance_{node}(N_i))}{|T|},$$

where n is the total number of basic nodes in tree T; $N_i \in T$; and $|T|$ is the size of the tree T. The function $Importance_{node}(N_i)$ returns the importance of node N_i.

7.2 Degree of Generalization

In the dissertation, the inductive learning process employs a generalization method to find all the proper maximal characteristic descriptions that satisfy the completeness condition. Finding a generalization tree (T_{LCSstr}) of T_1 and T_2 is to find a $LCSstr(T_1, T_2)$. A generalization tree of T_1 and T_2 is not valid if the size of T_{LCSstr} is too small comparing to T_1 and T_2. For instance, if T_1 and T_2 trees are generalized to be a node H only, then it is too general because it matches any subtree rooted with H. We define two indices to measure the degree of generalization: the

Degree of Structure Generalization (*DSG*), and the *Degree of Node Content Generalization* (*DNCG*). Given two Document Samples Trees T_1 and T_2, $LCSstr(T_1, T_2)$ $= Cut(T_1, S_1)$ ($= Cut(T_2, S_2)$) is found, where S_1 is the set of all roots of the removed subtrees in T_1 and S_2 is the set of all roots of the removed subtrees in T_2. The DSG's of T_1 and T_2 with respect to T_2 and T_1 respectively are calculated using the following formulas:

$$DSG_{T_1|T_2} = \frac{|T_1| - |Cut(T_1, S_1)|}{|T_1|}$$

$$DSG_{T_2|T_1} = \frac{|T_2| - |Cut(T_2, S_2)|}{|T_2|}.$$

The formulas for calculating the $DNCG$'s of T_1 and T_2 with respect to T_2 and T_1 respectively are given as follows:

$$DNCG_{T_1|T_2} = \frac{|\{N_i|N_i \in T_1, N_i \text{ is a basic node}\}| - |\{N_i|N_i \in Cut(T_1, S_1), N_i \text{ is a basic node}\}|}{|\{N_i|N_i \in T_1, N_i \text{ is a basic node}\}|}$$

$$DNCG_{T_2|T_1} = \frac{|\{N_i|N_i \in T_2, N_i \text{ is a basic node}\}| - |\{N_i|N_i \in Cut(T_2, S_2), N_i \text{ is a basic node}\}|}{|\{N_i|N_i \in T_2, N_i \text{ is a basic node}\}|}.$$

The $DSG_{T_1|T_2}$, $DSG_{T_2|T_1}$, $DNCG_{T_1|T_2}$ and $DNCG_{T_2|T_1}$ for generalizing T_1 and T_2 must satisfy the following criteria: Let DSG_{T_1,T_2} be $\max\{DSG_{T_1|T_2}, DSG_{T_2|T_1}\}$. Let $DNCG_{T_1,T_2}$ be $\max\{DNCG_{T_1|T_2}, DNCG_{T_2|T_1}\}$. Then

$$DSG_{T_1,T_2} \leq C_{DSG}, \text{ and}$$

$$DNCG_{T_1,T_2} \leq C_{DNCG},$$

where C_{DSG} and C_{DNCG} are predefined constants.

The degree of generalization will be extended to a set of trees. Given a set of trees $\{T_1, T_2, \ldots, T_i\}$, the resulting $LCSstr(T_1, \ldots, T_i) = Cut(T_1, S_1)$ $(= Cut(T_2, S_2) =, \ldots, = Cut(T_i, S_i)$) is valid if and only if the degrees of generalization of all the trees satisfy the above criteria.

7.3 Generic Generalization Rules

A generalization rule is to transform a set of descriptions $E_i = \{e_{i,k} | k \leq n\}$, where n is the total number of sample documents, and i is an index for document type K_i, into a more general description D_i that weakly implies the initial description. If a testing document falsifies a more general description D_i then it must falsify some specific description in E_i. Let CTX, CTX_1 and CTX_2 represent some arbitrary expressions that are augmented by additional predicates to formulate a concept description. Four generic generalization rules [18] can be described as follows:

- The *dropping condition rule*:

  $$\{CTX \,\&\, S \overset{c}{\Longrightarrow} K\} \lhd \{CTX \overset{c}{\Longrightarrow} K\},$$

  where S is an arbitrary predicate or logical expression and K is a document type. This rule states that a concept description CTX & S can be generalized by simply removing a conjunctively linked expression S.

- The *adding alternative rule*:

  $$\{CTX_1 \overset{c}{\Longrightarrow} K\} \lhd \{CTX_1 \vee CTX_2 \overset{c}{\Longrightarrow} K\}.$$

  This rule states the a concept description can be generalized by adding an alternative such as CTX_2 to it. The alternative CTX_2 is added by extending the scope of permissible values of one specific descriptor.

- *The extending reference rule*:

$$\{CTX \;\&\; [L = R_1] \overset{c}{\Longrightarrow} K\} \lhd \{CTX \;\&\; [L \;= R_2] \overset{c}{\Longrightarrow} K\},$$

  where $R_1 \subseteq R_2 \subseteq DOM(L)$ and $DOM(L)$ denote the domain of L. In this rule, L is a term, i.e. a constant, a variable, or a function, and R_1 and R_2 are internal disjunctions of values of L. The rule describes that a concept description can be generalized by enlarging the values of a constant, a variable, or a function of the description.

- The *turning constant to variable rule*:

$$\{F[a], F[b], \ldots, F[i]\} \lhd \forall v, F[v],$$

  where $F[v]$ stands for some description dependent on variable v; and $a, b, \ldots,$ and i are constants.

Before we give detailed rules for Document Type Tree generation, let's look at some *patterns* used in the *ABTE* and the *LCSstr* algorithms, which can be used for *turning constant to variable* in generalization process. Recall that the *ABTE* algorithm is used for Document Sample Tree Matching process and the *LCSstr* algorithm is employed in the Document Type Tree Discovering process. In *ABTE* algorithm, in addition to having constant nodes, whose labels and contents are specified, a pattern may contain the following marks:

- variables ($\_x$, $\_y$, etc.);
- bars (|).

These marks may appear in several places in a pattern tree (i.e., Document Sample Tree). Edges of the pattern tree are marked by bars. Leaves of the pattern tree are marked by variables preceded with an underscore. As shown in Figure 7.1, a mark-substitution (instantiation) s on the pattern pa replaces the nodes or subtrees in the data tree t according to the following ways:

- Each variable matched with a subtree in t. (Repeated variables are matched with identical subtrees.)
- Each bar is viewed as a pseudo node in pa, which is matched with part of a path (one or more pseudo nodes) from the root to a leaf of t.

Let $s(pa)$ be the resulting pattern tree after the application of mark substitution. We require that any mapping from $s(pa)$ to t maps the substituting nodes to themselves. Thus, no cost is induced by mark substitution. The distance between pa and t with respect to the substitution s is defined to be $\text{dist}(s(pa), t)$. The distance between pa and t is obtained from one of the best mark-substitutions, i.e.,

$$\text{dist}(pa, t) = \min_{s \in S}\{\text{dist}(s(pa), t)\},$$

where S is the set of all possible mark-substitutions.

In the *LCSstr* algorithm, a bar marked below a H (or V) node in a Document Type Tree matches a repeated H (or V) nodes on a path in L-S Tree of a testing document.

The following generalization rules are used to infer the Document Type Trees from a set of Document Sample Trees, T_{DS}'s.

7.4 Rules for Preprocessing Document Sample Trees

Before the generalization taking place, all the Document Sample Trees are preprocessed by using Rule 7.1 to cut the leaf node whose type is dynamic, i.e. its key term is *NULL*, Rule 7.2 to cut the leaf node which is not a basic node, and Rule 7.3 to cut the node where all its descendants are not basic nodes.

Rule 7.1 *Preprocessing Rule 1*

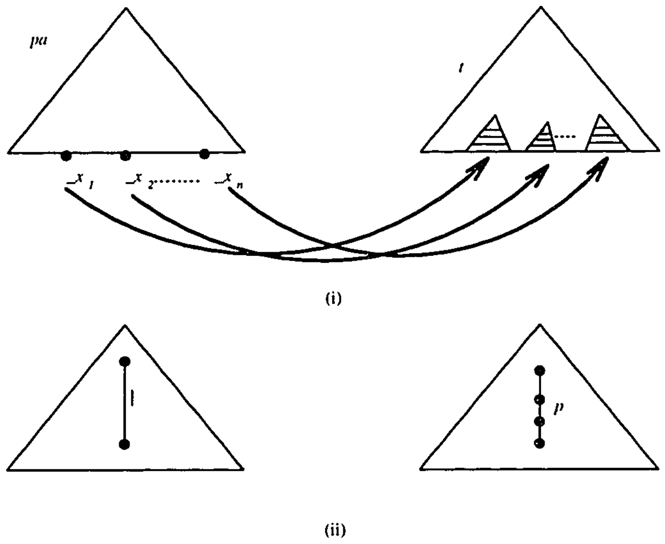


Figure 7.1 (i)Variable instantiation: The variables in pa are matched with the shaded subtrees in t. (ii) Bar instantiation: The bar is matched with the nodes (block dots) on a path p.

IF $(Contain(T_{DS}, N) \wedge IsLeafNode(N) \wedge (KeyTerm(N) = NULL))$

THEN cut N from T_{DS}

The predicate $Contain(T, N)$ returns true if T contains the node N, otherwise it returns false; the predicate $IsLeafNode(N)$ returns true if N is a leaf node, otherwise it returns false, and the function $KeyTerm(N)$ returns the value of key term for leaf node N.

Rule 7.2 *Preprocessing Rule 2*

IF $(Contain(T_{DS}, N) \wedge (IsLeafNode(N) \wedge \neg IsBasicNode(N)))$

THEN cut N from T_{DS}

The predicate $IsBasicNode(N)$ returns true if N is a basic node, otherwise it returns false.

Rule 7.3 *Preprocessing Rule 3*

IF $(Contain(T_{DS}, N)$

$\wedge(\neg IsLeafNode(N) \wedge AllDescendantsNotBasicNode(N)))$

THEN cut N from T_{DS}

The predicate $AllDescendantsNotBasicNode(N)$ returns true if all the descendants of N are not basic nodes, otherwise it returns false.

7.5 Discovering the Largest Common Substructures

In this section, we investigate a possible way of finding the Largest Common Substructures for Document Sample Trees of a document type.

Algorithm 7.1 *Creating LCSstr table.* Let $\mathbf{S}_{DS} = \{T_{DS1}, T_{DS2}, \ldots, T_{DSn}\}$, $n \geq 2$ be a set of preprocessed Document Sample Trees for a document type. A $LCSstr$ table, whose names of the rows and columns are $T_{DS_1}, T_{DS_2}, \ldots, T_{DS_n}$, consists of all possible entries $LCSstr(T_{DS_i}, T_{DS_j}) = T_{LCSstr_{i,j}}$, where $1 \leq i, j \leq n$.

For each entry $LCSstr(T_{DS_i}, T_{DS_j})$, a generalization of T_{DS_i} and T_{DS_j}, we determine the validity of the generalization, using

Rule 7.4 *Checking for Degree of Generalization Condition*

IF $((DSG_{T_{DSi},T_{DSj}} \leq C_{DSG}) \wedge (DNCG_{T_{DSi},T_{DSj}} \leq C_{DNCG}))$

THEN the generalization is valid

Both C_{DSG} and C_{DNCG} are predefined constants.

If a tree T_{LCSstr} ($= LCSstr(T_{DS_i}, T_{DS_j}) = Cut(T_{DS_i}, S_{DS_i}) = Cut(T_{DS_j}, S_{DS_j})$) is found and satisfies Rule 7.4, then S_{DS_i} and S_{DSj} are the set of cutting nodes removed from T_{DS_i} and T_{DS_j} respectively during generalization. We analyze the sets S_{DS_i} and S_{DS_j} to discover other of T_{LCSstr}'s using the following rule. (Recall that $T[i]$ stands for the subtree rooted at $t[i]$.)

Rule 7.5 *LCSstrs Discovering rule.*

IF $(\exists n_{DS_i} \geq 2, T_{DS_i}[l_i + 1], T_{DS_i}[l_i + 2], \cdots, T_{DS_i}[l_i + n_{DS_i}] \in T_{DS_i})$

$(\ Sibling(t_{DS_i}[l_i + 1], t_{DS_i}[l_i + 2], \cdots, t_{DS_i}[l_i + n_{DS_i}])$

$\wedge(1 \leq |\ \{t_{DS_i}[u] | l_i \leq u \leq (l_i + n_{DS_i}),\ t_{DS_i}[u] \in Cut(T_{DSi}, S_{DS_i})\}\ | < n_{DS_i})$

$\wedge(t_{DS_i}[l_i + n_{DS_i} + 1] = Parent(t_{DS_i}[l_i])\ )$

$\wedge(\exists n_{DS_j} \geq 2, T_{DS_j}[l_j + 1], T_{DS_j}[l_j + 2], \cdots, T_{DS_j}[l_j + n_{DS_j}], \in T_{DS_j})$

$(\ Sibling(t_{DS_j}[l_j + 1], t_{DS_j}[l_j + 2], \cdots, t_{DS_j}[l_j + n_{DS_j}])$

$\wedge(1 \leq |\ \{t_{DS_j}[u] | l_j \leq u \leq (l_j + n_{DS_j}),\ t_{DS_j}[u] \in Cut(T_{DSj}, S_{DS_j})\}\ | < n_{DS_j})$

$\wedge(t_{DS_j}[l_j + n_{DS_j} + 1] = Parent(t_{DS_j}[l_j])\ )$

$\wedge(Path(t_{Cut(T_{DSi}, S_{DS_i})})[l_i + n_{DS_i} + 1] = Path(t_{Cut(T_{DSj}, S_{DS_j})})[l_j + n_{DS_j} + 1])$

THEN find the *Longest Common Subsequences* of

$T_{DS_i}[l_i + 1], T_{DS_i}[l_i + 2], \cdots, T_{DS_i}[l_i + n_{DS_i}]$ and

$T_{DS_j}[l_j + 1], T_{DS_j}[l_j + 2], \cdots, T_{DS_j}[l_j + n_{DS_j}]$,

and discover other of $LCSstr$'s from them.

The predicate $Sibling(t_1, t_2, \cdots, t_i, \cdots, t_n)$ returns true if all the arguments t_i's are siblings and the ordering of the siblings corresponds to the ordering from the left to right in the tree. The function $Parent(N)$ returns the parent of a node N if it exists, or Null otherwise. The condition $1 \leq |\ \{t_{DS_i}[u] | l_i \leq u \leq l_i + n_{DS_i}\ , t_{DS_i}[u] \in S_{DS_i}\}\ | < n_{DS_i}$ represents that at least one but not all of the nodes in $\{t_{DS_i}[u] | l_i \leq u \leq l_i + n_{DS_i}\}$ was cut. Figure 7.2 illustrates this rule. For example, Figure 7.3 depicts a set of 11 Document Sample Trees of a MEMO document type. The DSG and $DNCG$ are defined to be 0.6 and 0.25 respectively. The $LCSstr$ trees are shown in Figure 7.4 and Figure 7.5, and their $LCSstr$ table is shown in Table 7.1. Each entry in the Table 7.1 is either blank if there is no valid $LCSstr$ to be discovered, or a set of T_{LCSstr} numbers if found, or $T_{DS\#}$ itself. Since the $LCSstr(T_1, T_2)$ is identical with $LCSstr(T_2, T_1)$, the $LCSstr$ table is diagonally symmetric.

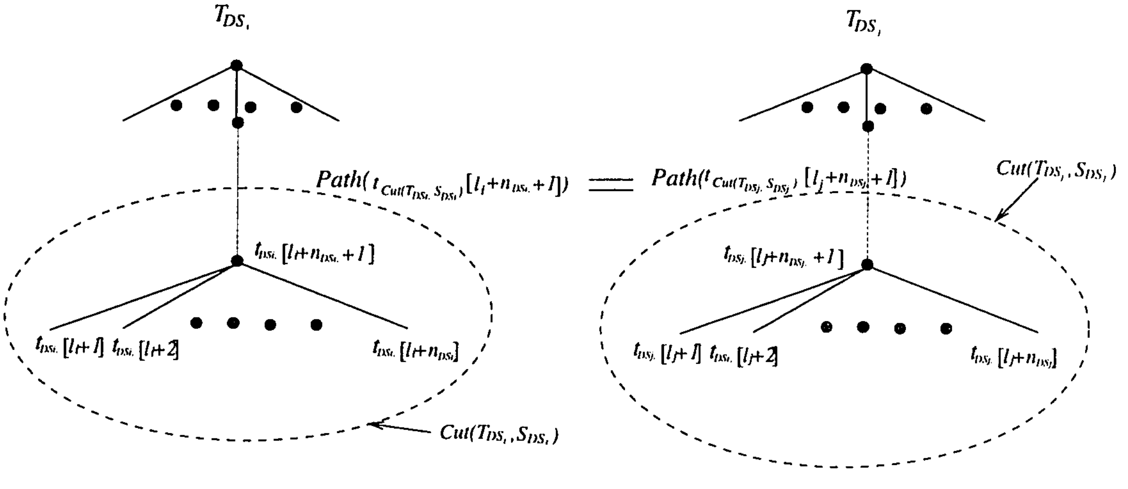

T_{DS_i}
T_{DS_j}
$Path(t_{Cut(T_{DS_i},S_{DS_i})}[l_i+n_{DS_i}+1])$ = $Path(t_{Cut(T_{DS_j},S_{DS_j})}[l_j+n_{DS_j}+1])$
$Cut(T_{DS_j},S_{DS_j})$
$t_{DS_i}[l_i+n_{DS_i}+1]$
$t_{DS_j}[l_j+n_{DS_j}+1]$
$t_{DS_i}[l_i+1]$ $t_{DS_i}[l_i+2]$
$t_{DS_i}[l_i+n_{DS_i}]$
$t_{DS_j}[l_j+1]$ $t_{DS_j}[l_j+2]$
$t_{DS_j}[l_j+n_{DS_j}]$
$Cut(T_{DS_i},S_{DS_i})$

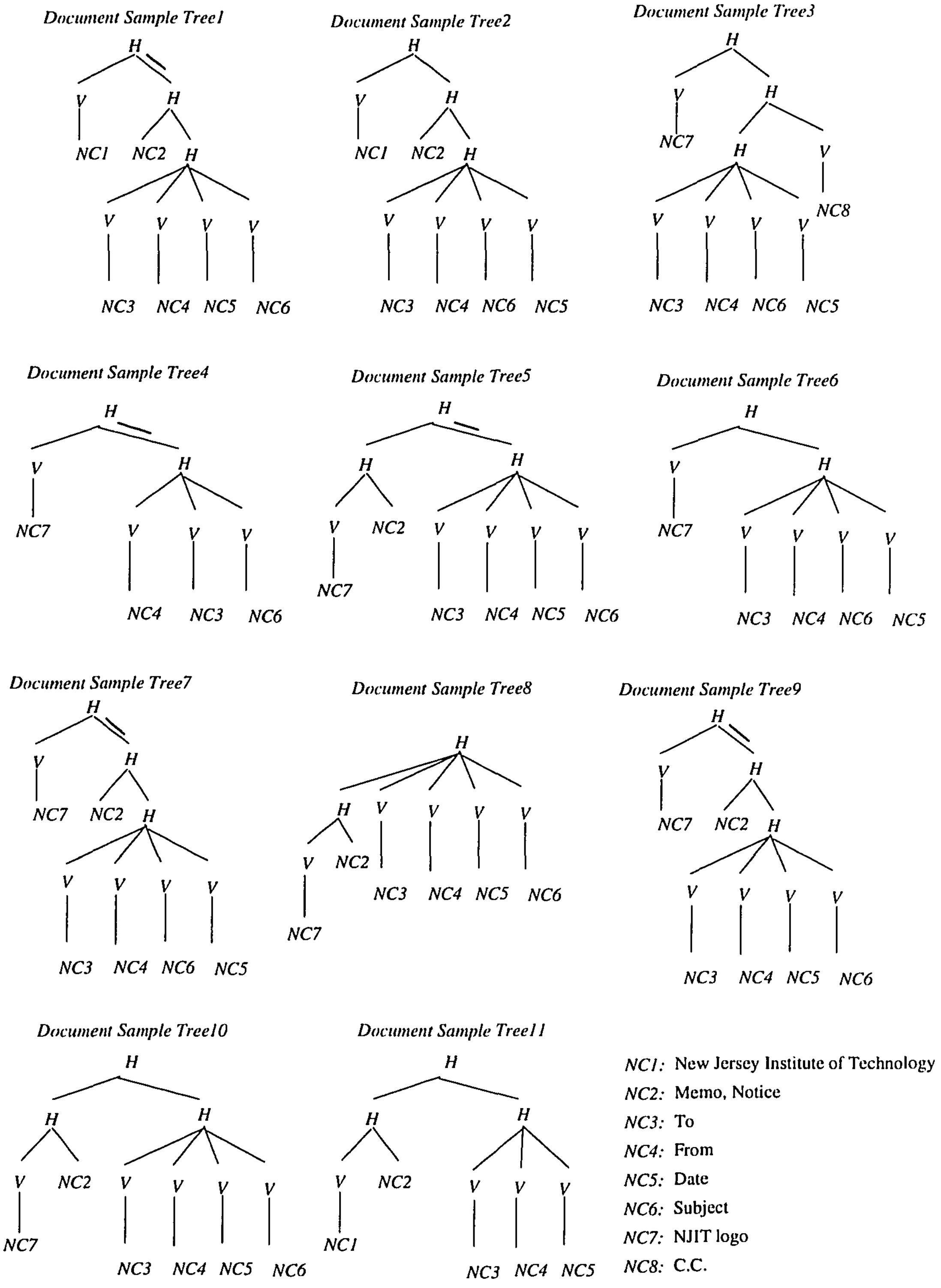
Figure 7.2 The rule of $LCSstr$'s discovering.

Table 7.1 $LCSstr$ table for Document Sample Trees in Figure 7.3.

$T_{DS}\#$	1	2	3	4	5	6	7	8	9	10	11
1	T_{DS_1}	1.1,1.2	2.1,2.2		4	9,11	6.1,6.2	7	8	4	11
2		T_{DS_2}	10		9,11	12	13	14.1,14.2	6.1,6.2	9,11	11
3			T_{DS_3}		9,11	12	15	14.1,14.2	16.1,16.2	9,11	11
4				T_{DS_4}		18.1,18.2					
5					T_{DS_5}	9,11	9,11	7	4	20	21
6						T_{DS_6}	12	14.1,14.2	9,11	9,11	11
7							T_{DS_7}	14.1,14.2	22.1,22.2	9,11	11
8								T_{DS_8}	7	7	14.1
9									T_{DS_9}	4	11
10										$T_{DS_{10}}$	21
11											$T_{DS_{11}}$

The $LCSstr$ table can be further transformed to a table, called the degree of completeness table. The degree of completeness table describes how many Document Sample Trees covered by each of $LCSstr$'s. The names of the rows are the list of Document Sample Trees and the names of the columns are the list of T_{LCSstr}. Table 7.2 shows a degree of completeness table for Table 7.1.

Obviously, T_{LCSstr} is a Document Type Tree which represents its document type if it covers all the Document Sample Trees. We can further analyze the generalization relations between each pair of $LCSstr$'s, and then update the degree of



Document Sample Tree1

Document Sample Tree2

Document Sample Tree3

Document Sample Tree4

Document Sample Tree5

Document Sample Tree6

Document Sample Tree7

Document Sample Tree8

Document Sample Tree9

Document Sample Tree10

Document Sample Tree11

Figure 7.3 Document Sample Trees of the MEMO document type.

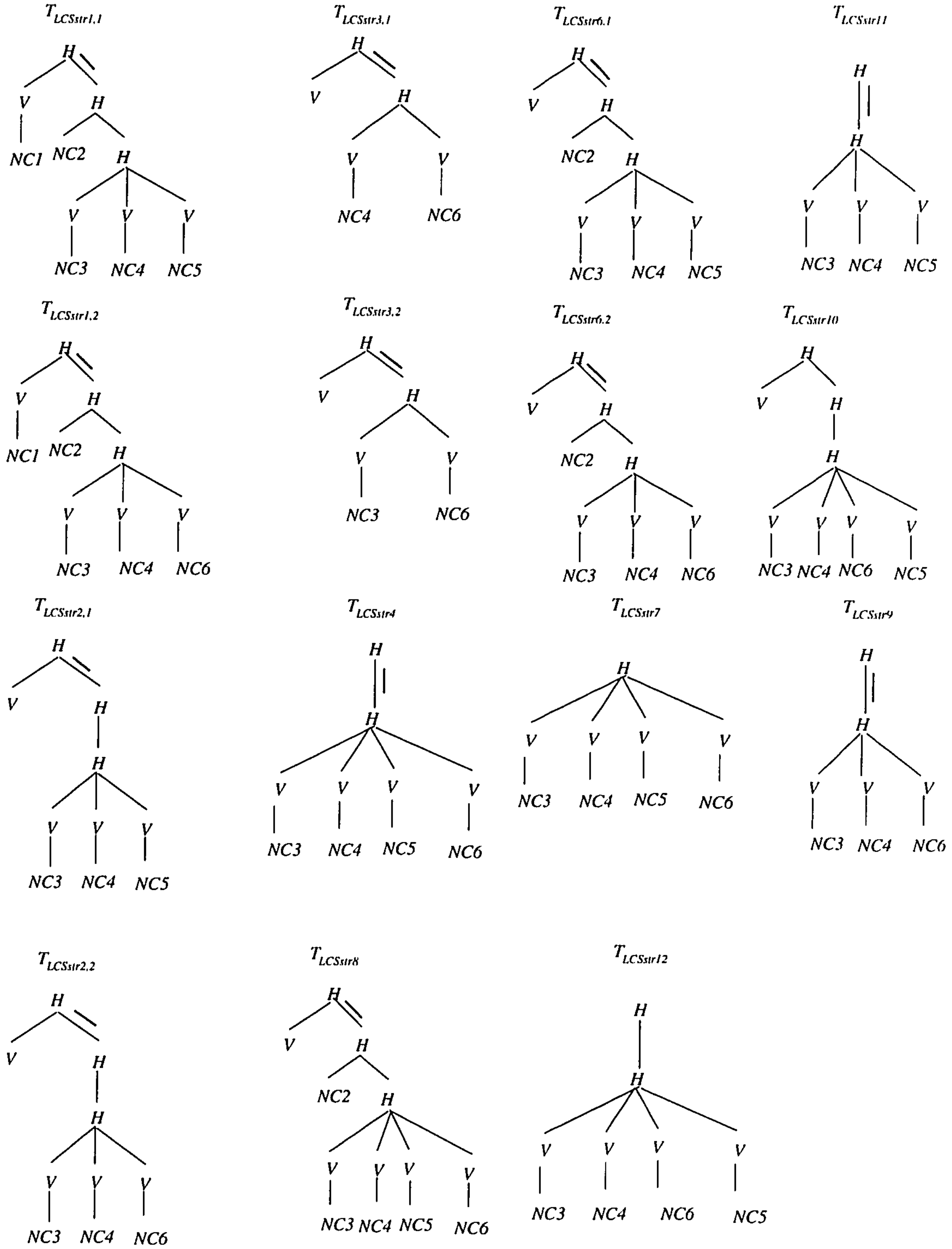

$T_{LCSstr1.1}$
H V H NC1 NC2 H V V V NC3 NC4 NC5
$T_{LCSstr3.1}$
H V H V V NC4 NC6
$T_{LCSstr6.1}$
H V H NC2 H V V V NC3 NC4 NC5
$T_{LCSstr11}$
H H V V V NC3 NC4 NC5
$T_{LCSstr1.2}$
H V H NC1 NC2 H V V V NC3 NC4 NC6
$T_{LCSstr3.2}$
H V H V V NC3 NC6
$T_{LCSstr6.2}$
H V H NC2 H V V V NC3 NC4 NC6
$T_{LCSstr10}$
H V H H V V V V NC3 NC4 NC6 NC5
$T_{LCSstr2.1}$
H V H H V V V NC3 NC4 NC5
$T_{LCSstr4}$
H H V V V V NC3 NC4 NC5 NC6
$T_{LCSstr7}$
H V V V V NC3 NC4 NC5 NC6
$T_{LCSstr9}$
H H V V V NC3 NC4 NC6
$T_{LCSstr2.2}$
H V H H V V V NC3 NC4 NC6
$T_{LCSstr8}$
H V H NC2 H V V V V NC3 NC4 NC5 NC6
$T_{LCSstr12}$
H H V V V V NC3 NC4 NC6 NC5

Figure 7.4 *LCSstr* trees for Document Sample Trees in Figure 7.3.

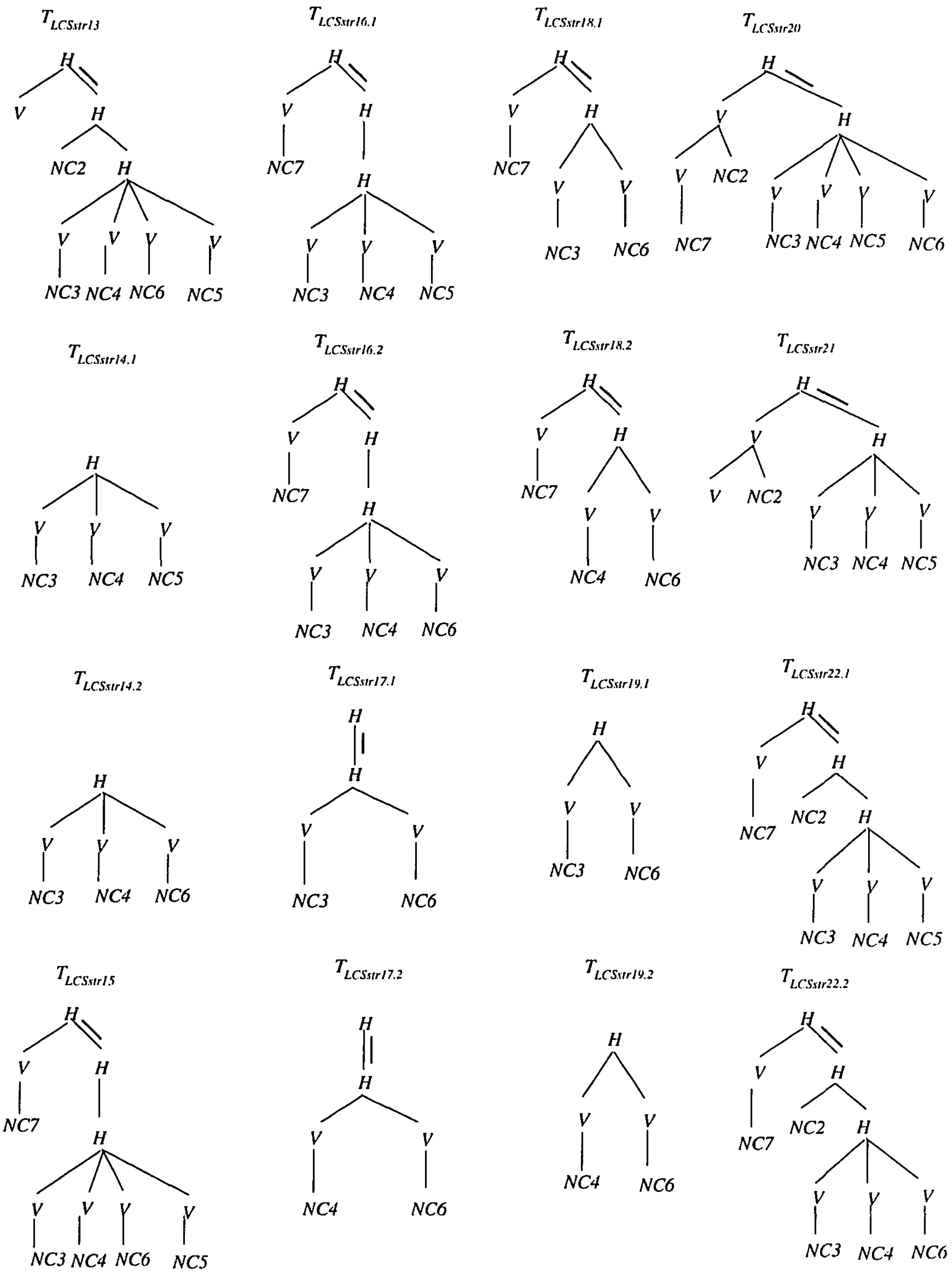


Figure 7.5 (continued from Figure 7.4) *LCSstr* trees for Document Sample Trees in Figure 7.3.

Table 7.2 Degree of completeness table for Table 7.1.

T_{LCSstr} #	T_{DS_1}	T_{DS_2}	T_{DS_3}	T_{DS_4}	T_{DS_5}	T_{DS_6}	T_{DS_7}	T_{DS_8}	T_{DS_9}	$T_{DS_{10}}$	$T_{DS_{11}}$
1.1	X	X									
1.2	X	X									
2.1	X		X								
2.2	X		X								
4	X				X				X	X	
6.1	X	X					X		X		
6.2	X	X					X		X		
7	X				X			X	X	X	
8	X								X		
9	X	X	X		X	X	X		X	X	
10		X	X								
11	X	X	X		X	X	X		X	X	X
12		X	X			X	X				
13		X					X				
14.1		X	X			X	X	X			X
14.2		X	X			X	X	X			
15			X				X				
16.1			X						X		
16.2			X						X		
18.1				X		X					
18.2				X		X					
20					X					X	
21					X					X	X
22.1							X		X		
22.2							X		X		

completeness table. In the remaining of this section, we define the generalization relation between two T_{LCSstr}'s. For finding generalization relation between T_{LCSstr}'s, all the generalization relations between each pair of T_{LCSstr}'s are first transformed to a generalization digraph, from which a modified generalization digraph can be obtained by removing redundant generalization relations from it. Then, the modified generalization digraph is used to update the degree of completeness table.

Definition 7.2 T_{LCSstr_i} can be embedded in T_{LCSstr_j} if and only if T_{LCSstr_i} is the $LCSstr$ of T_{LCSstr_i} and T_{LCSstr_j}.

Definition 7.1 T_{LCSstr_j} *can be generalized to* T_{LCSstr_i}*, denoted as* $T_{LCSstr_j} \xrightarrow{\lhd} T_{LCSstr_i}$*, if* T_{LCSstr_i} *can be embedded in* T_{LCSstr_j}*.*

In Figure 7.4, T_{LCSstr_4} can be generalized to T_{LCSstr_7}, $T_{LCSstr_{11}}$, $T_{LCSstr_{14.1}}$, and $T_{LCSstr_{14.2}}$, each of which can be embedded in T_{LCSstr_4}.

Algorithm 7.2 *Generalization Digraph.* The table of generalization relation can be represented by a set of ***directed acyclic graphs*** called the generalization digraph $G(V, E)$, where each edge $e \in E$ stands for "can be generalized to" relation, denoted as $v \xrightarrow{\lhd} u$, and vertices $v, u \in V$ stand for T_{LCSstr}'s. Then, a modified generalization digraph $G'(V, E)$ can be obtained from the generalization digraph $G(V, E)$ by using Rule 7.6 to remove redundant edges from it.

Rule 7.6 *Removing redundant edges from the generalization digraph.*

IF $(v_1 \xrightarrow{\lhd} v_2, v_2 \xrightarrow{\lhd} v_3, \ldots, v_{k-1} \xrightarrow{\lhd} v_k) \wedge (v_1 \xrightarrow{\lhd} v_k)$
THEN remove the edge $v_1 \xrightarrow{\lhd} v_k$

A generalization digraph $G = (V, E)$ can be represented by an *adjacent matrix*, where $V = \{T_{LCSstr_1}, T_{LCSstr_2}, \ldots, T_{LCSstr_n}\}$. The adjacent matrix for G is a $n \times n$ matrix A of booleans, where $A[i, j]$ is true if and only if there is an edge from vertex i to vertex j (that is, vertex i can be generalized to vertex j). Another representation of generalization digraph $G = (V, E)$ is the *adjacent list.* The adjacent list for a vertex i is a list of all vertices adjacent to i in some order.

The modified generalization digraph for the $LCSstr$ trees in Figure 7.4 and Figure 7.5 is shown in Figure 7.6 in which only direct edges are drawn without triangles. This digraph also describes the order of updating each T_{LCSstr}'s covers in the degree of completeness table. For example, the T_{DS}'s which are covered by T_{LCSstr_7} must be updated by T_{LCSstr_4} after T_{LCSstr_4} has been updated by $T_{LCSstr_{20}}$ and T_{LCSstr_8}. Applying Algorithm 7.3, a modified degree of completeness table as shown in Table 7.3 is obtained by updating the degree of completeness table (in Table 7.2).

Algorithm 7.3 *Update the degree of completeness table.*

```
/*G(V,E) = Generalization Digraph */
queue $Q$;
vertex $v$, $y$;
while ( $|E| > 0$ )
{
    for each vertex $v$ which does not have edge pointing to it
    {
        MAKENULL($Q$);
        $bf(v)$;
        for each vertex y
            mark[y] = UNVISITED;
    }
}

$bfs(v)$
{
    Vertex $x, y$;
    ENQUEUE($v, Q$);
    while (EMPTY($Q$) != TRUE)
    {
        $x$ = FRONT($Q$);
        for each vertex $y$ adjacent to $x$
        {
            if there is not any edge pointing to $x$
            {
                if (mark[$y$] == UNVISITED)
                {
```

```
                (1) in the degree of completeness table, mark the row of the
                    $T_{LCSstr_y}$ with all the trees covered by $T_{LCSstr_x}$;
                (2) ENQUEUE($x, Q$);
                (3)delete the edge $x \longrightarrow y$;
                (4)mark[$y$] = VISITED;
            }
        }
    }
    DEQUEUE($Q$);
  }
}
```

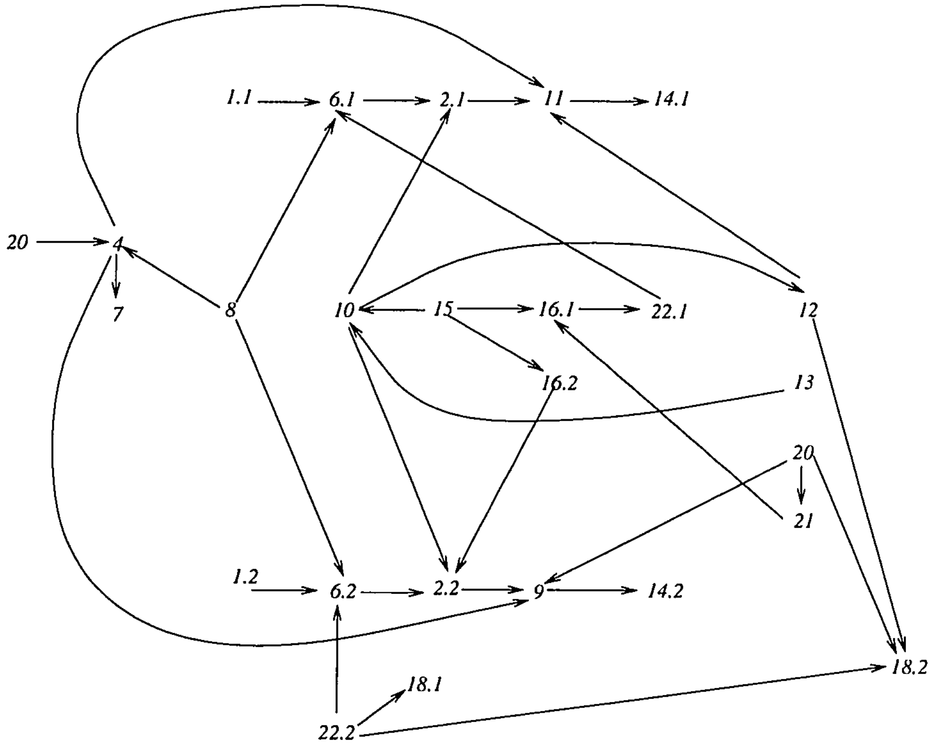


Figure 7.6 Modified generalization digraph for the *LCSstr* trees in Figure 7.4 and Figure 7.5.

Table 7.3 Modified degree of completeness table.

T_{LCSstr} #	T_{DS_1}	T_{DS_2}	T_{DS_3}	T_{DS_4}	T_{DS_5}	T_{DS_6}	T_{DS_7}	T_{DS_8}	T_{DS_9}	$T_{DS_{10}}$	$T_{DS_{11}}$
1.1	X	X									
1.2	X	X									
2.1	X	X	X		X		X		X	X	X
2.2	X	X	X				X		X		
4	X				X				X	X	
6.1	X	X	X		X		X		X	X	X
6.2	X	X					X		X		
7	X				X			X	X	X	
8	X								X		
9	X	X	X		X	X	X		X	X	
10		X	X				X				
11	X	X	X		X	X	X		X	X	X
12		X	X			X	X				
13		X					X				
14.1	X	X	X		X	X	X	X		X	X
14.2	X	X	X		X	X	X	X	X	X	
15			X				X				
16.1			X		X		X		X	X	X
16.2			X				X		X		
18.1			X	X		X	X		X		
18.2				X	X	X					
20					X					X	
21					X					X	X
22.1			X		X		X		X	X	X
22.2							X		X		

In Section 7.6, the use of this table for discovering the set of T_{LCSstr}'s, which covers all the T_{DS}'s will be investigated.

7.6 Search for Document Type Trees

Algorithm 7.4 is employed to find all the possible sets of T_{LCSstr}'s, which represent one document type.

For the MEMO document type, Algorithm 7.4 takes Table 7.3 as input to search all the possible sets of T_{LCSstr}'s which cover all the T_{DS}'s. Given a set S_{DS} of Document Sample Trees, if $S_{LCSstr} = \{T_{LCSstr,1}, T_{LCSstr,2}, \ldots, T_{LCSstr,i}\}$ is found during generalization, then it is meaningless if the number of Document Type Trees is greater than the total number of Document Sample Trees. We use the Rule 7.7 to limit the size of the set S_{LCSstr}.

Algorithm 7.4 *Search for Document Type Trees.*

```
/* $S_{DS}$ = set of Document Sample Tree */
/* $S_{LCSstr}$ = set of Largest Common Substructures */
boolean Table[$|S_{LCSstr}|$][$|S_{DS}|$];
/* array of modified degree of completeness table */
boolean Select_as_head[$|S_{LCSstr}|$];
boolean Select_as_element[$|S_{LCSstr}|$];
boolean Current_cover[$|S_{LCSstr}|$];
    for j = 1 to $|S_{DS}|$
    {
        k = find_the_seed_$T_{LCSstr}$_and_not_selected(Table);
        /* Find the $T_{LCSstr}$ which covers the most number of $T_{DS}$ and
            was not selected as the seed. */
        select_as_head[k] = True;
        for m = 1 to $|S_{DS}|$ current_cover[m] = Table[k][m];
        /* Copy the trees which are covered by $T_{LCSstr}$ into current_cover[].*/
        for l =1 to $|S_{DS}|$
        {
            search(k);
        }
    }

search(k)
{
stack S;
    MAKENULL(S);
    for i=1 to $|S_{LCSstr}|$ selected_as_element[i] = False;
    selected_as_element[k] = True;
```

```
PUSH (k, S);
while (EMPTY(S) != True)
{
    k = find_the_element_$T_{LCSstr}$_and_not_selected(Table);
    /* Find the $T_{LCSstr}$ which covers the maximum number of uncovered $T_{DS}$'s
        and was not selected as the element and seed. If there is more than one $T_{LCSstr}$
        found, select the first one in the searching order who covers maximum number
        of $T_{DS}$'s. */
    if (k == NULL )
    /* Fail to find a $T_{LCSstr}$ in this search, and recover the array of current_cover[]. */
    {
        m = TOP(S);
        recover(m, current_cover[]);
        remove the effective covers from m;
        POP(S);
        /* Remove this $T_{LCSstr}$ and do another search. */
        selected_as_element[m] = True;
    }       if ( k != NULL )
    {
        for l:=1 to $|S_{DS}|$
            if ((Table[k][l] == 1) and (current_cover[l] == 0)) current_cover[l] = 1;
        if (cover_complete(current_cover[]) == True)
        {
            output(S);
            /* A set of $T_{LCSstr}$ which completely covers all the trees in $S_{DS}$ is found.*/
            recover(current_cover[]);
            /* Reset to the coverage without k.*/
            selected_as_element[k] = True;
        }
```

```
            if (cover_complete(current_cover[]) = False)
            {
                output(Q);
                selected_as_element[k] = True;
                PUSH(k,S);
            }
        }
    }
    return;
}
```

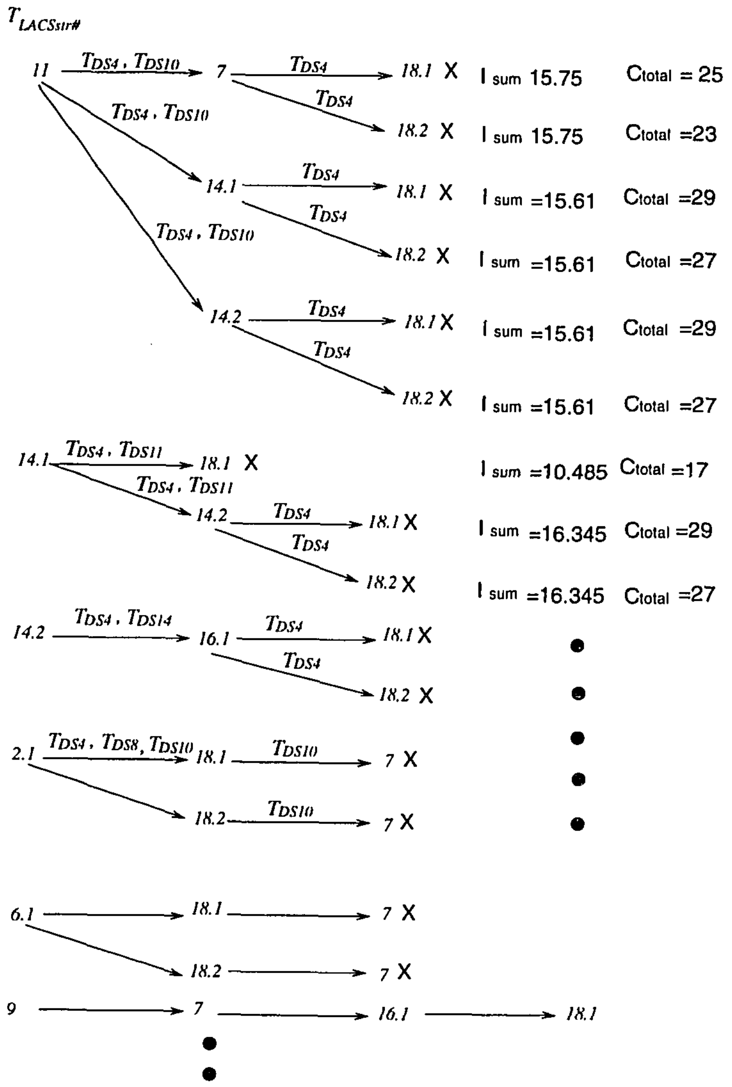
Rule 7.7 *Limit the size of final* S_{LCSstr}.

IF $|S_{LCSstr}| > C_2 * |S_{DS}|$

THEN select the most important $(C_2 * |S_{DS}|)$ trees from S_{LCSstr}

and save them into new Document Type Tree S_{DT}

where $0 < C_2 < 1$ is a predefined constant. We select the set of T_{LCSstr}'s that has a minimum member of trees and satisfies Rule 7.7 as the Document Type Trees. If there is more than one set of T_{LCSstr}'s found, we then choose the set S which has the maximum value of $(\sum_{k=1}^{k=|S|} Importance_{tree}(T_{LCSstr_k}))$, where $T_{LCSstr_k} \in S$. As shown in Figure 7.7, the search process is as follows. $T_{LCSstr_{11}}$ which covers 12 T_{DS}'s is first selected as a seed to start searching. T_{DS_4} and $T_{DS_{10}}$ are not covered so far. Then, T_{LCSstr_7} is selected to cover $T_{DS_{10}}$. Then, $T_{LCSstr_{18.1}}$ or $T_{LCSstr_{18.2}}$ is selected to cover T_{DS_4}. Now all the T_{DS}'s are covered. Two sets of candidate Document Type Trees, $\{T_{LCSstr_{11}}, T_{LCSstr_7}, T_{LCSstr_{18.1}}\}$ and $\{T_{LCSstr_{11}}, T_{LCSstr_7}, T_{LCSstr_{18.2}}\}$ are found. As the search process continues, $\{T_{LCSstr_{14.1}}, T_{LCSstr_{18.1}}\}$ will be finally chosen as the Document Type Trees because it contains only two members.



I sum : Summation of Importances of a set of T_{LCS}'s.

Ctotal: Total number of Document Sample Trees covered by a set of T_{LCS}'s.

X : End of this path.

Figure 7.7 Search process of Algorithm 7.4.

7.7 Number of Document Type Trees and Computational Complexity of Classification

The number of Document Type Trees to be discovered during generalization depends on the number of common features found. There is no Document Type Tree discovered, if all the Document Sample Trees have totally different tree structures and node contents. According to the generalization algorithm (Algorithm 7.4), each T_{LCSstr} tree covers at least two Document Sample Trees. Given two trees T_1 and T_2, the complexity of discovering largest common substructure of T_1 and T_2 is bounded by $O(|T_1| \times |T_2| \times \min(H_1, L_1) \times min(H_2, L_2))$, where H_j is the height of T_j and L_j is the number of leaf nodes in T_j [25]. This is the same as the complexity of the Tree Matching Algorithm in [39] for comparing two trees using the edit distance.

Given s_i Document Sample Trees of document type i, t_i Document Type Trees are found to cover c_i Document Sample Trees. We consider three cases as follows.

Case 1 : $t_i = 0$ and $c_i = 0$. No common feature can be found between any pair of Document Sample Trees.

Case 2 : $t_i = 1$ and $c_i = s_i$. All the Sample Document Trees are covered by one Document Type Tree.

Case 3 : $0 < t_i \leq c_i/2$ and $0 < c_i < s_i$. Only c_i Document Sample Trees are covered by t_i Document Type Trees.

Given the L-S Tree T_{test} of a testing document, the time complexity of classifying T_{test} as the document type i is as follows.

In *Case 1*, the complexity is

$$O(\sum_{k=1}^{k=s_i} |T_{test}| \times |T_{DS_k}| \times \min(H_{test}, L_{test}) \times min(H_{DS_k}, T_{DS_k})).$$

In other words, the generalization algorithm does not save any computational time because no Document Type Tree is found for this document type.

In *Case 2*, the complexity is $O(|T_{test}|\times|T_{DT}|\times\min(H_{test},L_{test})\times min(H_{DT},L_{DT}))$ where the classification system takes the most advantage from generalization.

In *Case 3*, the complexity is

$$O(\sum_{k=1}^{k=t_i}|T_{test}|\times|T_{DT_k}|\times\min(H_{test},L_{test})\times\min(H_{DT_k},L_{DT_k})+$$
$$\sum_{l=1}^{l=(s_i-c_i)}|T_{test}|\times|T_{DS_l}|\times\min(H_{test},L_{test})\times\min(H_{DS_l},L_{DS_l})).$$

7.8 Inductive Learning Process for Constructing Document Type Trees

In this section, we summarize what we have discussed throughout the Chapter 7, by describing an inductive learning process for constructing the set of Document Type Trees for a document type i which is as follows:

- Preprocess each Document Sample Tree T_{DS} in the set S_{DS} of Document Sample Trees

  1. Rule 7.1 (Preprocessing Rule 1): Remove dynamic leaves.
  2. Rule 7.2 (Preprocessing Rule 2): Cut the non-basic leaves.
  3. Rule 7.3 (Preprocessing Rule 3): Cut the nodes containing non-basic descendant nodes.
  4. Consider the relation of $LCSstr$ and $LCSreg$.

- Construct the $LCSstr$ table.

  1. Discover the T_{LCSstr}'s for each pair of T_{DS}'s in S_{DS}.
  2. Rule 7.5: Find $LCSstr$'s.
  3. Rule 7.4: Check the degree of generalization.
  4. Algorithm 7.1: Create the $LCSstr$ table.

- Create the degree of completeness table.

- Find Document Type Trees.

  1. Algorithm 7.2: Construct a Generalization Digraph.
  2. Rule 7.6: Remove redundant generalization relations.
  3. Algorithm 7.3: Update the table of degree of completeness.
  4. Algorithm 7.4: Search for Document Type Trees.
  5. Rule 7.7: Limit the number of Document Type Trees.

7.9 Finding All the Possible Largest Common Substructures

The original *LCSstr* algorithm [25] discovers the first *LCSstr* between two trees T_1 and T_2. Rule 7.5 finds some other *LCSstr*'s by applying longest common subsequence algorithm and analyzing T_1 and T_2. For the application of discovering Document Type Tree, the original *LCSstr* algorithm and Rule 7.5 is sufficient because it always finds all the possible *LCSstr*'s. But the algorithm does not consider all the cases having the same maximum size. This section describes another alternative modifying the original algorithm to find all the possible *LCSstr*'s.

Let $F = T[i..j]$ be an ordered forest containing nodes numbered from i to j in tree T, as shown in Figure 7.8. A set S of nodes in F is said to be a set of consistent subtree cuts in F if (i) $t(p) \in S$ implies that $i \leq p \leq j$, and (2) $t[p], t[q] \in S$ implies that neither one is an ancestor of the other in F. Let $Cut(F, S)$ represent the subforest of F with substree removals at all nodes in S. Let $Subtree(F)$ be the set of all possible sets of consistent subtree cuts in F. Let $fdist(F_1, F_2)$ be the distance from forest F_1 to forest F_2. The size of largest common substructures of F_1 and F_2, denoted $fsize(F_1, F_2, 0)$, is defined to be $\max\{|Cut(F_1, S_1)| + |Cut(F_1, S_2)|\}$ such that

$$fdist(Cut(F_1, S_1), Cut(F_2, S_2)) = 0$$

$$S_1 \in Substrees(F_1)$$

$$S_2 \in Substrees(F_2).$$

Let $l(i)$ denote the postorder number of the leftmost leaf of the subtree $T[i]$. If $T[i]$ is a leaf, $l(i) = i$. Let $desc(i)$ represent the set of postorder numbers of the descendants of the node $t[i]$. For example, in Figure 7.8, $l(8) = 1$ and $l(7) = 5$, $desc(8) = \{1,2,3,4,5,6,7\}$ and $desc(7) = \{5,6\}$. The $fsize(F_1, F_2, 0)$ can be represented by $fsize(l(i)..s, l(j)..t, 0)$ if $F_1 = T_1[l(i)..s]$ and $F_2 = T_2[l(j)..t]$, where $s \in desc(i)$ and $t \in desc(j)$. The size of largest common substructures of subtrees $T_1[i]$ and $T_2[j]$, which represent the substree rooted at $t_1[i]$ of T_1 and the substree rooted at $t_2[j]$ of T_2 respectively, denoted $tsize(T_1[i], T_2[j], 0)$ (or simply $tsize(i, j, 0)$), is $\max\{|Cut(T_1[i], S_1)| + |Cut(T_2[j], S_2)|\}$ such that

$$fdist(Cut(T_1[i], S_1), Cut(T_2[j], S_2)) = 0$$

$$S_1 \in Substrees(T_1[i])$$

$$S_2 \in Substrees(T_2[j]).$$

Lemma 3.3 [25] *Suppose $s \in desc(i)$ and $t \in desc(j)$. If ($l(s) \neq l(i)$ or $l(t) \neq l(j)$), then*

$$fsize(l(i)..s, l(j)..t, 0) = \mathbf{max}\begin{cases} fsize(l(i)..l(s) - 1, l(j)..t, 0), \\ fsize(l(i)..s, l(j)..l(t) - 1, 0), \\ fsize(l(i)..l(s) - 1, l(j)..l(t) - 1, 0) + tsize(s, t, 0). \end{cases}$$

Lemma 3.3 exhausts all three possible cases yielding $fsize(l(i)..s, l(j)..t, 0)$.

Case 1. If the subtree $T_1[s]$ is removed (i.e., $t_1[s] \in S_1$), the forest left in $T_1[l(i)..s]$ becomes $T_1[l(i)..l(s)-1]$ and the $fsize(l(i)..s, l(j)..t, 0) = fsize(l(i)..l(s) - 1, l(j)..t, 0)$.

Case 2. If the subtree $T_2[t]$ is removed (i.e., $t_2[t] \in S_2$), the forest left in $T_2[l(j)..t]$ becomes $T_2[l(j)..l(t)-1]$ and $fsize(l(i)..s, l(j)..t, 0) = fsize(l(i)..s, l(j)..l(t) - 1, 0)$.

Case 3. If neither $t_1[s]$ nor $t_2[t]$ is removed (i.e., $t_1[s] \notin S_1$ and $t_2[t] \notin S_2$), $t_1[s]$ maps to $t_2[t]$. In the mapping from $Cut(T_1[l(i)..s], S_1)$ to $Cut(T_2[l(j)..t], S_2), T_1[s]$ must be mapped to $T_2[t]$ to ensure the zero distance between $Cut(T_1[l(i)..s], S_1)$ and $Cut(T_2[l(j)..t], S_2)$. Accordingly, $fsize(l(i)..s, l(j)..t, 0) = fsize(l(i)..l(s) - 1, l(j)..l(t) - 1, 0) + tsize(s, t, 0)$.

In the original *LCSstr* algorithm, if there are two or more cases having the same maximum size, only the first case (in the order from *Case 1* to *Case 3*) is chosen and the rest of the cases are discarded. The modified algorithm will keep all the cases which have same maximum sizes and saves them in the mapping array of *map*[s][t][0]. The array *map*, which is a array of pointers, stores the information of selected case which has the maximum size. An array *table*[s][t][0] is created to store a series of mappings in array *map* when a substructure in $T_1[i]$ is found and matches a substructure in $T_2[j]$ during the search of *LCSstr*'s between $T_1[i]$ and $T_2[j]$. Because each entry in array *map* can point to more than one case, each entry of *table* could be a tree instead of a linked list in the original algorithm. After all the sizes of all the substructures of subtrees in T_1 and T_2 are found, we search for all the of subtree pairs $T_1[i]$ and $T_2[j]$ having maximum sizes of substructures $tsize(i, j, 0)$. Then, we discover the mappings form $T_1[i]$ to $T_2[j]$ by checking the array *table*. Since each entry in *table* is a tree structure, a stack and depth first search are devised to travel the tree structures and find all the possible *LCSstr*'s. The idea is also explained in Figure 7.9.

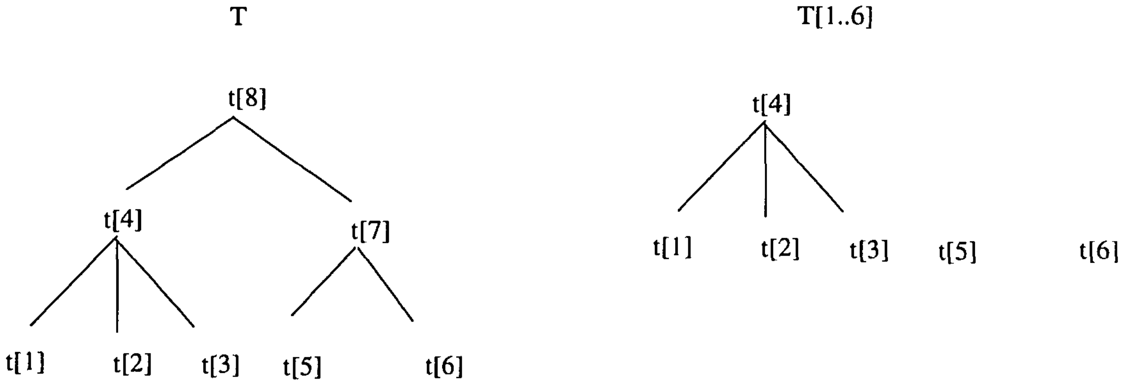
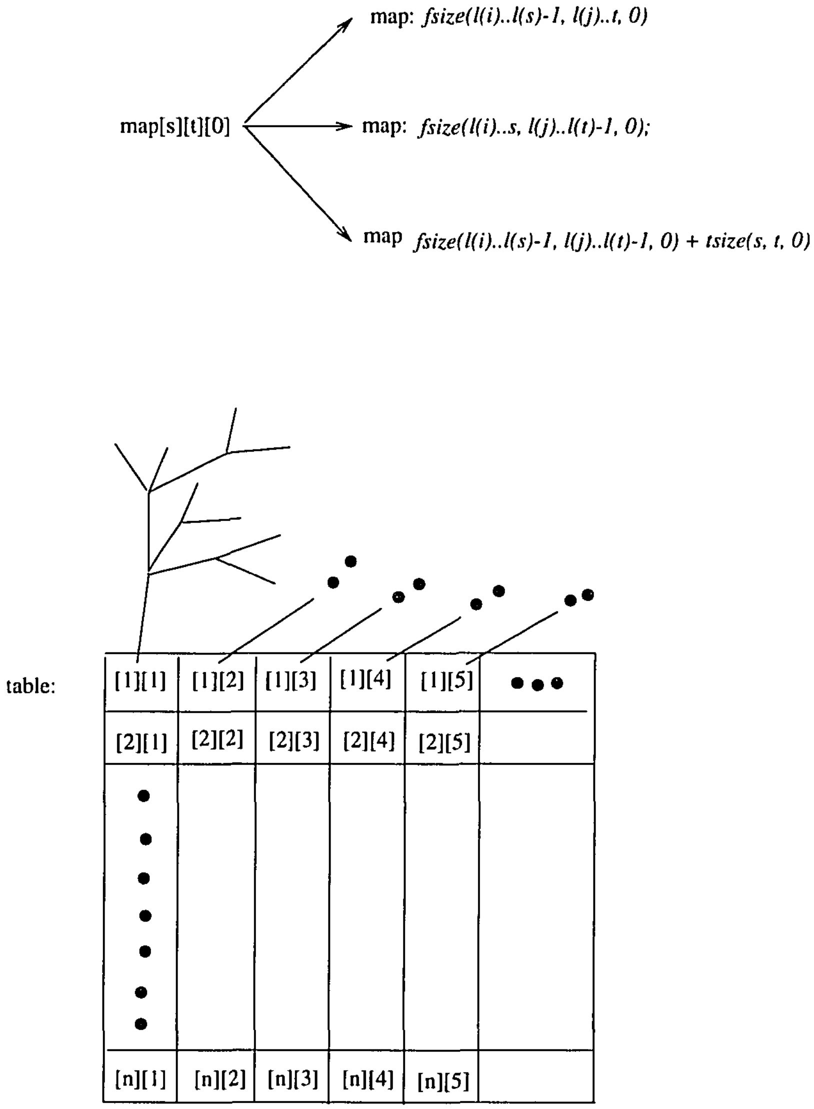


Figure 7.8 An example of induced forest.


map: fsize(l(i)..l(s)-1, l(j)..t, 0)
map[s][t][0]
map: fsize(l(i)..s, l(j)..l(t)-1, 0);
map fsize(l(i)..l(s)-1, l(j)..l(t)-1, 0) + tsize(s, t, 0)
table:

[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	●●●
[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	
● ● ● ● ● ● ●					
[n][1]	[n][2]	[n][3]	[n][4]	[n][5]	

Figure 7.9 A data structure of *map* and *table*.

CHAPTER 8
DOCUMENT CLASSIFICATION

In the preceding chapters, we presented the generation of the Document Sample Trees and Document Type Trees. In this chapter we will discuss the document classification.

Given a testing document D, it is first transferred to be L-S Tree. Because we only enforce the completeness condition during inductive learning process, it is possible that two different document types could have some identical Document Type Trees. The consistency condition can be satisfied by associating each Document Type Tree with a weight based on Zipf's law [31]. If a Document Type Tree occurs in m types, its weight is assigned as $\log_2[(M/m)]$, where M is the total number of types.

The first stage is Document Type Tree discovering. We try to discover each Document Type Tree from a given L-S Tree by applying the $LCSstr$ algorithm. After discovering process, each document type obtains a raw score, which is equal to the sum of the weights of the Document Type Trees occurring in the L-S Tree. The raw score of a type is normalized by dividing the score by the total weight of all the Document Type Trees and then multipling it by 100. This either succeeds to find the best fitting document type candidates, or fails to find any one. For the second case, since it discovers no Document Type Tree from the L-S Tree, D must be a new document type or a new format of an existing document type. The KAT (Knowledge Acquisition Tool) will be activated to update the structural knowledge base. In the first case, the Document Sample Trees belonging to document type candidates will match against the L-S Tree to find the exact document format. If the Document Sample Trees of all the possible candidates of Document Type Trees fail to match the L-S Tree of the testing document, the classification system will learn the node contents (i.e. the key terms, attribute, *etc*) of the testing document through user interaction by activating KAT. Figure 8.1 shows the document classification process.

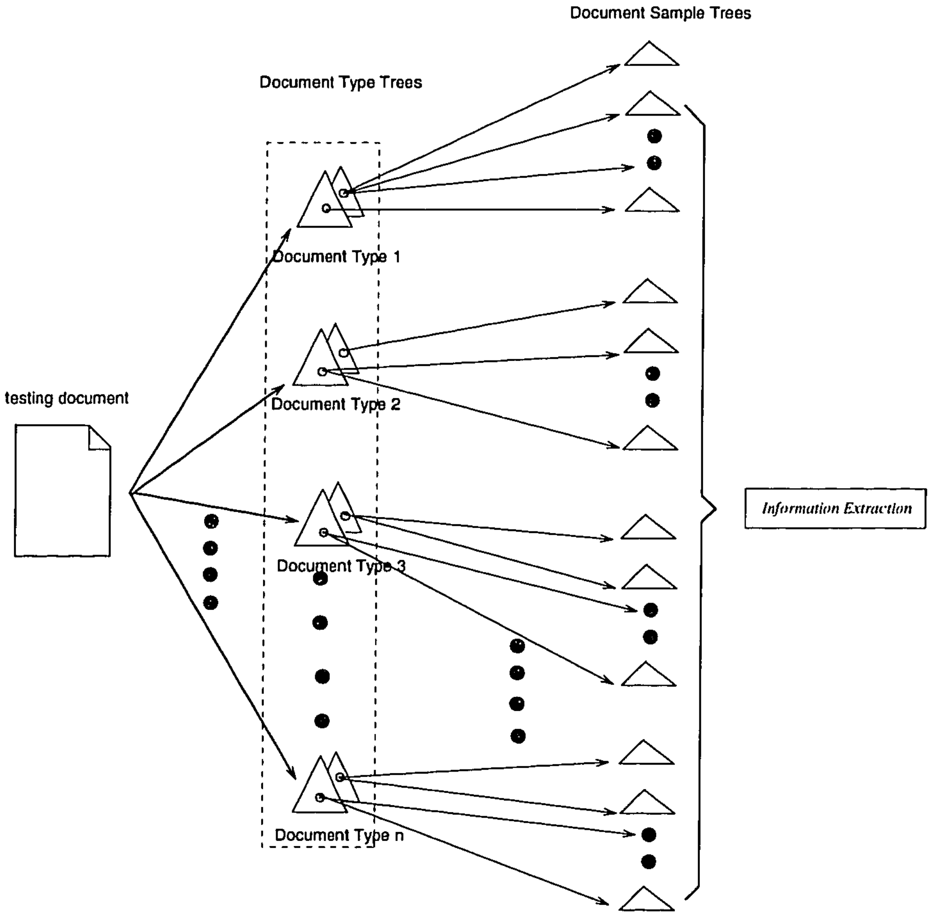


Figure 8.1 Document classification process.

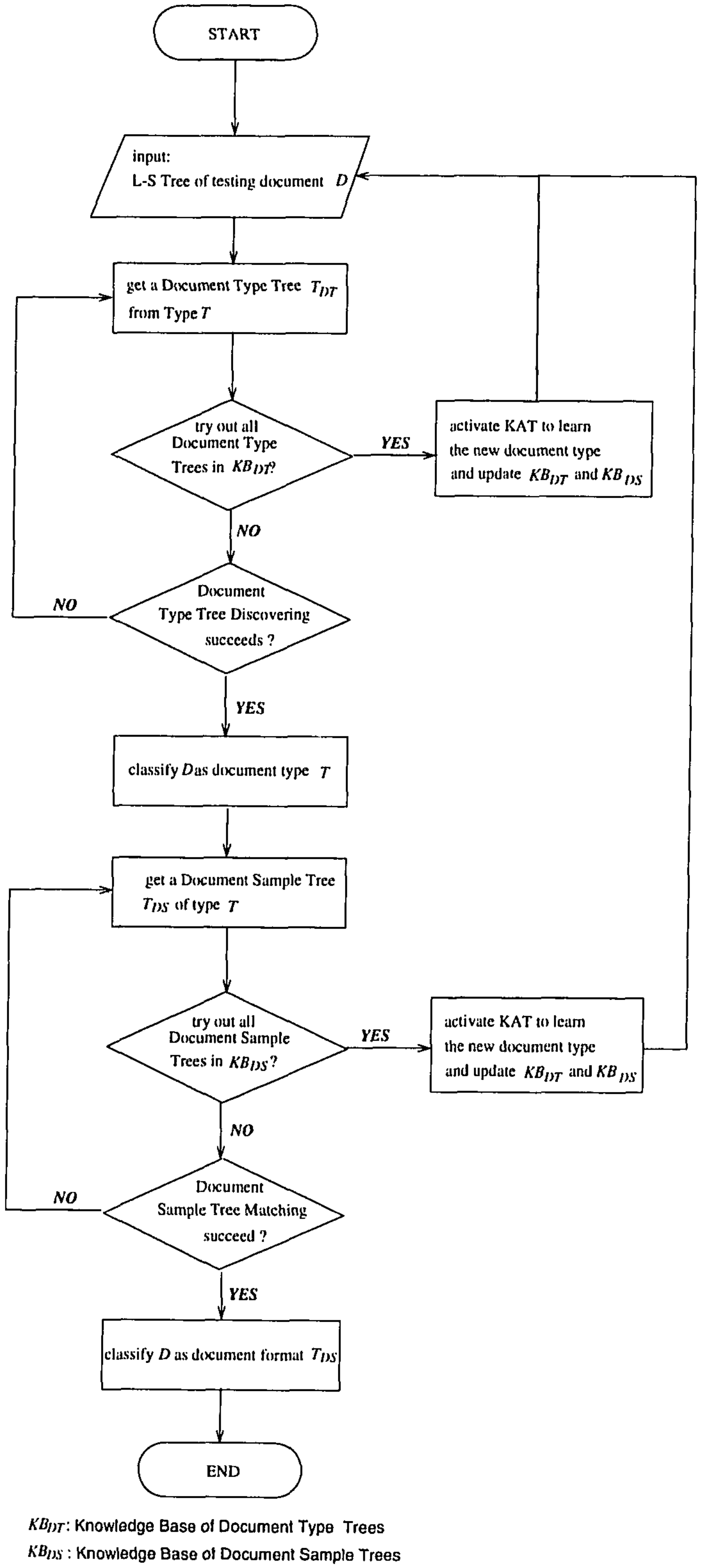

START
input:
L-S Tree of testing document D
get a Document Type Tree T_{DT}
from Type T
try out all
Document Type
Trees in KB_{DT}?
YES
activate KAT to learn
the new document type
and update KB_{DT} and KB_{DS}
NO
Document
Type Tree Discovering
succeeds ?
NO
YES
classify D as document type T
get a Document Sample Tree
T_{DS} of type T
try out all
Document Sample
Trees in KB_{DS}?
YES
activate KAT to learn
the new document type
and update KB_{DT} and KB_{DS}
NO
Document
Sample Tree Matching
succeed ?
NO
YES
classify D as document format T_{DS}
END
KB_{DT}: Knowledge Base of Document Type Trees
KB_{DS} : Knowledge Base of Document Sample Trees

Figure 8.2 Document classification algorithm.

CHAPTER 9
EXPERIMENTAL RESULTS AND CONCLUSION

This dissertation presents the design of a knowledge based system in TEXPROS [32] for classifying office documents. The layout structure and conceptual analysis of documents are used to identify the testing document. A novel inductive learning technique is presented, and is employed to train the system and build up the structural knowledge base (the Document Sample Trees and Document Type Trees). A knowledge Acquisition Tool is devised to perform the inductive learning from L-S Trees of document samples and then generate the Document Sample Tree and Document Type Tree bases. The Document Type Trees allow that a small set of trees (rather than a large pool of Document Sample Trees) is possibly used to identify the type of a document during document classification process. A testing document is classified if a Document Type Tree is discovered as a substructure of the L-S Tree of the testing document, then we match the L-S Tree with the Document Sample Trees of the classified type to find the format of the teting document. Our empirical study shows that the document recognition rate is very promising by using this tool.

Forty different document samples for each of eight document types were selected. In total, three hundred and thirty different document samples were selected and preclassified into eight different document types. Fifteen sample documents out of forty document samples of each type were used for training the classification system in the learning stage. Figure 9.1 depicts some the Document Type Trees discovered during the knowledge acquisition process. After the system has been trained, another twenty five document examples are employed to test the classification process. The document types include letter, memo, journal of IP&M (Information Processing & Management), PAMI (Pattern Analysis and Machine Intelligence), E. M. (Journal of Electronic Materials), COMM. (IEEE Transactions

Table 9.1 Experimental result 1 of document type classification.

	letter	memo	IP&M	PAMI	E. M.	COMM.	COMPUTER	call for papers	unknown
letter	90%								10%
memo		90%							10%
IP&M			100%						
PAMI				100%					
E. M.					100%				
COMM.						100%			
COMPUTER							100%		
call for papers								80%	20%

on Communication), COMPUTER (IEEE Transactions on Computer) , and call-for-papers. The experimental result is represented by the *precision rate* defined as:

$$precision\ rate = \frac{M}{N} \times 100\%,$$

where M is the number of documents of some type classified successfully and N is the total number of documents of that type being tested.

The result is shown in Table 9.1. From Table 5, 10% of the letter, 10% of the memo, and 20% of the call-for-papers are classified as unknown document types because, in learning process, the sample documents couldn't cover all the possible document formats. The journal document type basically has fixed document format and its recognition rate is not proportional to the number of training samples. The rest of the documents were classified 100%.

By increasing the number of training samples of each document type to 20, the experiment shows that recognition rates of letters and memos are raised up to 96% and 94% as shown in Table 9.2.

The time needed to generate Document Type Trees for document type *memo*, *letter* and *Journal* in the first learning process is listed in Table 9.3.

Table 9.2 Experimental result 2 of document type classification.

	letter	memo	IP&M	PAMI	E. M.	COMM.	COMPUTER	call for papers	unknown
letter	96%								4%
memo		94%							6%
IP&M			100%						
PAMI				100%					
E. M.					100%				
COMM.						100%			
COMPUTER							100%		
call for papers								80%	20%

Table 9.3 Experimental result of document type learning time.

document type	time(seconds)
letter	40
memo	35
Journal	30

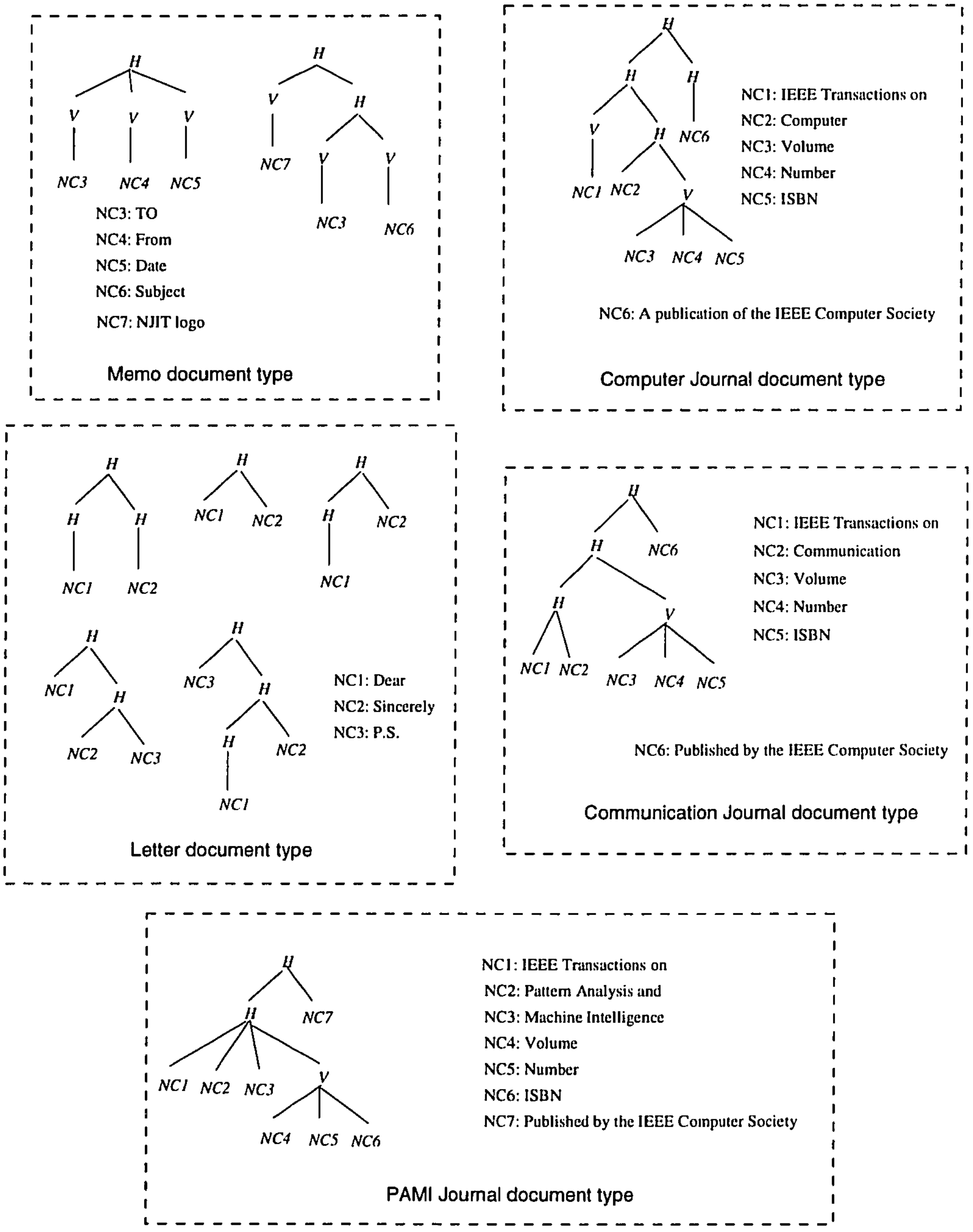


Figure 9.1 Document Type Trees.

CHAPTER 10
FUTURE RESEARCH

We would like to conclude this dissertation writing with a note describing some of the research issues which remains to be investigated.

In this dissertation, we demonstrated that the Document Sample Trees can be obtained during the learning stage of the process of classifying documents, which are represented by L-S Trees. We then proposed that the Document Sample Trees of a type are generalized to a fewer Document Type Trees. This allows that, in the classifying stage, given a L-S Tree of a document to be classified, we are first finding the best possible match between the L-S Tree and a Document Type Tree from a pool of Document Type Trees of various types in the base, instead of Document Sample Trees of various types. This speeds up the process of classifying documents. However, the success of optimizing the speed of classifying documents depends upon how smaller number of the Document Type Trees per type we can get. The generalization rules, which are used to generate Document Type Trees from a large pool of Document Sample Trees of a type, employs three criteria such as the importance of a tree, the degree of generalization, and the degree of completeness. These criteria represent the user's preference. However, it is our conjecture that there are more preference criteria, such as semantic importance of node contents, that can be used to speed up the process of discovering $LCSstr$'s.

Throughout the discussion in this dissertation, we had established with experimental results that the proposed scheme is operable. However, there is a need to formalize the concepts of Document Sample Trees and Document Type Trees, which represent *concisely* and *completely* the significant characteristics and features of the classifying documents. The formalization of these concepts could allow us to investigate the properties of these concepts for representing documents; to examine the relationship between the Document Sample Trees and the Document Type Trees such that a fewest Document Type Trees for representing each type can be theoretically obtained; to partition the Trees into sets of trees of various document types based on their representing characteristics and features without examining, if possible, the structural differences among the trees; to recognize that there are repetitive tree structures regardless of their document types, without going through the process of matching trees; to recognize the possibility of having two identical Document Type Trees of different document types, and so forth. Above all, the formalized concept of the Document Type Trees allows us to prove that we always can classify correctly documents of their representing types.

Given a specific application domain, how do we determine the document type hierarchy for representing a large collection of documents? Specifically, how do we represent a type of documents? In TEXPROS, we use the concept of frame template consisting of various attributes for describing the common and distinct characteristics and features of documents of different types. However, for a large collection of documents of numerous types, the use of attributes for describing the common and distinct characteristics and features of documents of different types becomes inadequate and ineffective way for representing the document types. Either we have to use a large list of attributes or there is only a few (possibly, one or two only) common attributes between any two frame templates. The former introduces the problem of increasing the size of the Document Sample Trees and therefore the size of the Document Type Trees. The latter leads to that the Document Type Trees will not be as fewer as we want to have. Because of these, our approach will be less effective.

Finally, what is the well-defined sets of Document Sample (or Type) Trees of each document type? That is, for each document type, what is the smallest set of Document Sample Trees (and therefore the Document Type Trees) that we need to represent all the possible documents of the type? How do we discover the additional

Document Sample (or Type) Trees from the existing Document Sample Trees and Document Type Trees? Such a knowledge discovery for the Document Sample or Type Trees may optimize the process of classifying documents.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, pp. 189–192, Addison-Wesley, Reading, Massachusetts, 1983.

2. R. Bareiss, *Examplar-Based Knowledge Acquisition*, Academic Press, New York, 1989.

3. B. G. Buchanan, D. K. Barstow, R. Bechtel, J. Bennett, W. Clancey, C. Kulikowski, T. M. Mitchell, , and D. A. Walterman, "Construct an Expert System," Building Expert System edited by F. Hayes-Roth, D. A. Waterman and D. B. Lenat, Reading, Massachusetts, Addison-Wesley, pp. 127-167, 1983.

4. J. G. Carbonell, R. S. Michalski, and T. M. Mitchell, "An Overview of Machine Learning ," Machine Learning - An Artificial Intelligence Approach, edited by R. S Michalski, J. G. Carbonell, and T. M. Mitchell, Morgan Kaufmann, Palo Alto, California, pp. 3-24, 1983.

5. W. J. Clancey, "Heuristic Classification," *Artificial Intelligence*, vol. 27, pp. 289–350, 1985.

6. R. D. Coyne, M. A. Rosenman, A. D. Radford, M. Balachandran, and J. S. Gero, *Knowledge-Based Design Systems*, Addison Wesley, Reading, Massachusetts, 1990.

7. A. Dengel and G. Barth, "ANASTASIL - A Hybrid Knowledge-Based System for Document Layout Analysis," in *Proceedings of International Joint Conference of Artificial Intelligence*, vol. 2, pp. 1254–1259, 1989.

8. H. Eirund and K. Kreplin, "Knowledge Based Document Classification Supporting Integrated Intelligent Document Handling," in *Proceedings on Office Information Systems*, pp. 189–196, 1988.

9. F. Esposito, D. Malerba, G. Semeraro, E. Annese, and G. Scafuro, "An Experimental Page Layout Recognition System for Office Document Automatic Classification," in *Proceedings on the 10th IEEE Conference on Pattern Recognition*, Atlantic City, New Jersey, pp. 557–562, 1989.

10. J. L. Fisher, S. C. Hinds, and D. P. D'amato, "A Rule-Based System for Document Image Recognition," in *Proceedings of the 10th IEEE International Conference on Pattern Recognition*, Atlantic City, New Jersey, pp. 567–572, June 1990.

11. T. R. Gruber, *The Knowledge Acquisition of Strategic Knowledge*, Academic Press, New York, 2nd ed., 1989.

12. X. Hao, J. T. L. Wang, M. Bieber, and P. A. Ng, "A Tool for Classifying Office Document," in *Proceedings of the 5th IEEE International Conference on Tools with Artificial Intelligence*, pp. 427–439, November 8-11 1993.

13. X. Hao, J. T. L. Wang, and P. A. Ng, "Nested Segmentation: A Approach for Layout Analysis in Document Classification," in *Proceedings of the Second IAPR Conference on Document Analysis and Recognition*, Tsukuba Science City, Japan, pp. 319–322, October 1993.

14. F. Hayes-Roth and J. McDermott, "Knowledge Acquisition Structural from Descriptions," in *the 5th International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, pp. 356–362, 1977.

15. S. J. Hong, "Developing Classification Rules from Examples," in *Tutorial for the Int. Conference on Artificial Intelligence for Applications*, Orlando, Florida, pp. 1–37, March 1993.

16. E. Lutz, H. V. Kleist-Retzow, and K. Hoernig, "MAFIA - An Active Mail-Filter-Agent for an Intelligent Document Processing Support," Multi-User Interface and Applications, eds., S. Gibbs and A. A. Verrijn-Stuart, Elsevier, Science Publishers, North Holland, pp. 16-32, 1990.

17. R. S. Michalski, "Discovering Classification Rules Using Variable-Valued Logic System VL1," in *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Standford, California, pp. 162–172, August 1973.

18. R. S. Michalski, "A Theory and Methodology of Inductive Learning," *Artificial Intelligence*, vol. 20, pp. 111–161, 1983.

19. R. S. Michalski, "Pattern Recognition as Rule-Guided Inductive Inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 2, no. 4, pp. 349–361, July 1980.

20. R. S. Michalski and R. E. Stepp, "Learning From Observation: Conceptual Clustering," Machine Learning - An Artificial Intelligence Approach, edited by R. S. Michalski and J. G. Carbonell and T. M. Mitchell, Morgan Kaufmann, Palo Alto, California, pp. 331-363, 1983.

21. B. Pagurek, N. Dawes, G. Bourassa, G. Evans, and P. Smithera, "Letter Pattern Recognition," in *Proceedings of the 6th Conference on Artificial Intelligence for Applications*, Santa Barbara, California, pp. 313–319, 1990.

22. Z. Pawlak, "Rough Classification," *Int. J. Man-Machine Studies*, vol. 20, pp. 469–483, 1984.

23. B. W. Porter, R. Bareiss, and R. Holte, "Concept Learning and Heuristic Classification in Weak Theory Domains," *Artificial Intelligence*, vol. 45, pp. 229–263, 1990.

24. J. Schmdit and W. Putz, "Knowledge Acquisition and Representation for Document Structure Recognition," in *Proceedings of the 9th Conference on Artificial Intelligence for Applications : the CAROL Project*, Orlando, Florida, pp. 177–181, March 1993.

25. D. Shasha, J. T. L. Wang, and K. Zhang, "On Discovering the Largest Approximately Common Substructures of Two Trees," Submitted for publication.

26. F. Shih, S. S. Chen, D. C. Hung, and P. A. Ng, "A Document Segmentation Classification and Recognition System," in *Proceedings of the Second IEEE International Conference on Systems Integration*, Morristown, New Jersey, pp. 258–267, June 15-18 1992.

27. Y. Y. Tang, C. D. Yan, and C. Y. Suen, "Document Processing for Automatic Knowledge Acquisition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 3-21, 1994.

28. Y. Tsuji, "Document Image Analysis for Generating Syntactic Structure Description," in *Proceedings of the 9th IEEE Conference on Pattern Recognition*, Rome, Italy, pp. 744–747, June 1988.

29. S. Tsujimoto and H. Asada, "Understanding Multi-articled Document," in *Proceedings of the 10th IEEE Conference on Pattern Recognition*, Atlantic City, New Jersey, pp. 551–556, 1990.

30. J. T. L. Wang, K. Jeong, K. Zhang, and D. Shasha, "Reference Manual for ABTE-A Tool for Approximate Tree Pattern Matching," technical report, Courant Institute of Mathematical Science, 1991.

31. J. T. L. Wang, T. G. Marr, and D. Shasha, "Discovering Active Motifs in Sets of Related Protein Sequences and Using Them for Classification," *Nucleic Acids Research*, vol. 22, no. 14, pp. 2769–2775, May 12 1994.

32. J. T. L. Wang and P. A. Ng, "TEXPROS: An Intelligent Document Processing System," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 2, pp. 171–196, June 1992.

33. J. T. L. Wang, K. Zhang, K. Jeong, and D. Shasha, "A System for Approximate Tree Matching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 4, pp. 559–571, August 1994.

34. C. S. Wei, "Inductive Learning and Knowledge Representation in Document Classification," Ph.D. Proposal, New Jersey Institute of Technology, Newark, New Jersey, 1994.

35. C. S. Wei, Q. Liu, J. T. L. Wang, and P. A. Ng, "Knowledge Discovering for Document Classification Using Tree Matching in TEXPROS," submitted to *Information Sciences: An International Journal*.

36. C. S. Wei, J. T. L. Wang, and P. A. Ng, "Inductive Learning and Knowledge Representation for Document Classification: the TEXPROS Approach," in *Proceedings of the Third IEEE International Conference on Systems Integration*, São Paulo City, Brazil, pp. 1166–1175, August 15-19 1994.

37. K. Y. Wong, R. G. Gasey, and F. M. Wahl, "Document Analysis System," *IBM J. Res. Develop*, vol. 6, no. 6, pp. 642–656, 1982.

38. R. Yasdi, "Learning Classification Rules from Database in the Context of Knowledge Acquisition and Representation," *IEEE Transactions of Knowledge Data Engineering*, vol. 3, no. 3, pp. 293–306, September 1991.

39. K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems," *SIAM J Computing*, vol. 18, no. 6, pp. 1245–1262, December 1989.