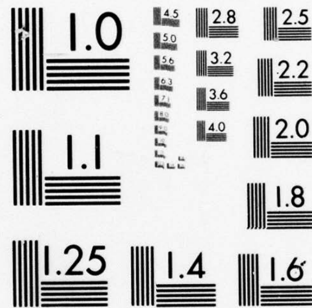
AD-A052 916 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/4
DISCRETE-PATTERN MATCHING ALGORITHMS AND DATA STRUCTURES FOR CY--ETC(U)
MAR 78 J B ISETT
UNCLASSIFIED AFIT/GCS/MA/78M-2 NL

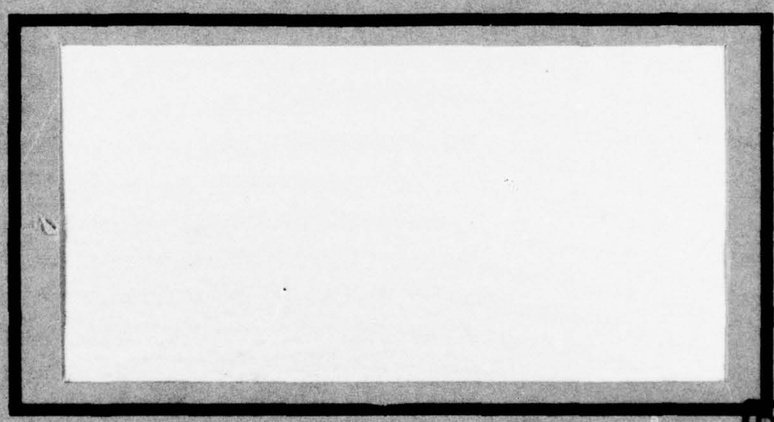
1 OF 2
AD
A052916



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A052916

AD No.
DDC FILE COPY



DDC
RECEIVED
APR 20 1978
A

UNITED STATES AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

AD A052916

AD No.
DDC FILE COPY

(1)

(6)

DISCRETE-PATTERN MATCHING
ALGORITHMS AND DATA STRUCTURES
FOR CYBER 74.

(9) Master's THESIS,

(10) John Barton Isett
Captain USAF

(14) AFIT/GCS/MA/78M-2

(11) Mar 78

(12) 102 p.

DDC
RECEIVED
APR 20 1978
A

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

012 225

Preface

This paper presents an investigation into the string pattern matching process and in particular, the realization of the process through use of various data structures and algorithms on the CDC CYBER 74 computer. The presentation presumes the reader is familiar with standard computer terminology and has a basic understanding of computer operation. A knowledge of finite state machines would be helpful during the discussion of alternate/successor linked list and finite state automata data structures but is not a prerequisite to their understanding.

As result of this work much basic insight into the pattern matching process has been developed and some interesting conclusions have been reached. The work done in evaluation of the various implementations was time consuming but enjoyable and forced a methodical approach in constructing and comparing the many test programs. This approach I will be able to apply in future work and is something that only experience can teach. To this end I wish to acknowledge the guidance and interest of my thesis advisor, Captain George Orr.

I would also like to express my gratitude for the assistance and patience of my wife, Karen, during the pursuit of a Masters degree.

John B. Isett

White Section
Buff Section
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

Contents

	Page
Preface	ii
List of Figures	v
List of Tables	vii
Abstract	viii
I. Introduction	1
The Pattern Matching Process	1
Pattern Matching Applications	2
Thesis Objective	5
Thesis Approach	6
Data Structures and Associated Algorithms	6
Simple List	6
Indexed Simple List	8
Indexed Linked List	13
Indexed Binary Tree	13
Indexed Alternate/Successor Linked List	19
Finite State Automata Linked List	21
II. CYBER 74 Implementations	29
Test Program Design	29
Unpacked Approaches	35
EXEC1A (Simple List)	35
EXEC2A (Indexed Simple List)	35
EXEC3A (Indexed Binary Tree)	36
EXEC4A (Alternate/Successor Linked List)	37
Packed Approaches	39
EXEC1B (Simple List)	40
EXEC2B (Indexed Simple List)	41
EXEC3B (Indexed Binary Tree)	41
EXEC4B (Alternate/Successor Linked List)	41
EXEC3C (Binary Tree, Packed Input)	44
EXEC4C (Alt/Suc Linked List, Packed Input)	48
Special Approaches	48
EXEC5B (Special Alt/Suc)	48
EXEC6A (f.s.a.)	50
III. Testing Procedures	54

IV. Results and Conclusions 60

Results 60
Conclusions 72

V. Recommendations 74

Bibliography 77

Vita 78

Appendix A: CYBER 74 Character Set With Display Codes A-1

Appendix B: Numerical Results of Test Program Executions B-1

List of Figures

Figure		Page
1	The Pattern Matching Process	2
2	Simple List Data Structure	7
3	Search with Simple List Data Structure	9
4	Indexed Simple List Data Structure	11
5	Search with Indexed Simple List	12
6	Indexed Linked List Data Structure	14
7	Search with Indexed Linked List	15
8	Indexed Binary Tree Data Structure	17
9	Search with Binary Tree	18
10	Indexed Alternate/Successor Linked List Data Structure	20
11	Search with Alternate/Successor Linked List . . .	22
12	Functions to Implement Finite State Automata Match Algorithm	24
13	Search with Finite State Automata Data Structure .	25
14	Packed Simple List Data Structure	27
15	Bubble Chart for Test Program Design	30
16	Structure Chart for Test Program Design	32
17	Programs Developed to Test Pattern Matching Algorithms	34
18	Binary Tree as Would be Constrcted by EXEC3A . . .	38
19	Packed Simple List Data Structure of EXEC1B . . .	41
20	Packed Binary Tree Data Structure of EXEC3B . . .	43
21	Sample Data Structure Construction Steps for EXEC4B	45

22 Pattern Matching as Done in EXEC3C and EXEC4C . . 47

23 Concept of Search as Used in EXEC5B 49

24 Sample Finite State Automata Data Structure Built by EXEC6A 52

25 Files Created for Evaluation Testing 57

26 Performance Measures 59

27 Bar Graph of Averaged Search Times and Averaged Data Structure Sizes 65

List of Tables

Table		Page
I	Rankings- Search Time	61
II	Rankings- Size of Data Structure	63
III	Rankings- Number of Match Checks to Find Patterns in Text	67
IV	Rankings- Number of Comparisons Made During Search	69
V	Rankings- Time Required to Construct Data Structure	71

Abstract

A description of the discrete-pattern matching process is presented with the key elements described. Six data structure approaches and related search algorithms are presented. These include a simple list, an indexed simple list, a linked list, a binary tree, an alternate/successor linked list, and a finite state automata.

Twelve programs were coded to implement five out of the six data structures/algorithms (linked list not used) using packed and unpacked approaches on a CYBER 74. Runs were made with twelve different data files using Fortran, English, and random text. The effect of the number of patterns in the data structure and the expected incidence in the text were included. Results indicate a "best" data structure/algorithm may be chosen from three implementations: an unpacked finite state automata approach, an unpacked alternate/successor linked list approach, and a packed version of the latter.

DISCRETE-PATTERN MATCHING ALGORITHMS AND DATA STRUCTURES FOR CYBER 74

I. Introduction

This thesis will investigate several different algorithms and data structures used to implement the pattern matching process with discrete-patterns. These algorithms and data structures were implemented on a Control Data Corporation (CDC) CYBER 74 computer.

A "discrete-pattern" is, in the context of this paper, a finite string of characters. For example, a single word, this line of text, or for that matter, this entire page may be considered a discrete-pattern since each is a finite string of characters. But, for the remainder of this paper, rather than referring to a "discrete-pattern" each time, the term "pattern" will be used instead.

The Pattern Matching Process

The concept of the pattern matching process is illustrated in figure 1. The two key elements which will be studied in this paper are the matching algorithm and the data structure. Also of some interest will be the type of input string and what influence it may have on the overall process.

Briefly, the pattern matching process is performed by the matching (or search) algorithm. This algorithm "searches" the input subject string for occurrences of patterns which are stored in the data structure. The results of the search

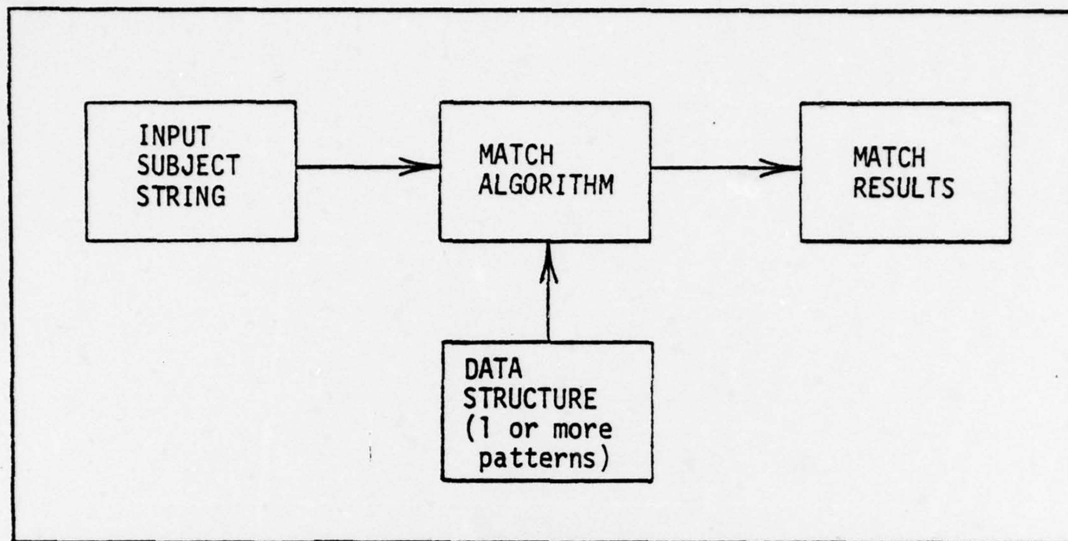


Figure 1. The Pattern Matching Process

are output as the end product of the match process.

The exact organization of the data structure can vary from one application to another, and so, with it, the match algorithm varies also. This thesis will attempt to identify efficient combinations of data structure/matching algorithm for use on the CYBER 74.

Pattern Matching Applications

Pattern matching is an integral concept in many different computer applications programs. In fact, for some uses the matching algorithm is the speed limiting, and hence, efficiency determining portion of the application. For example, a language compiler during its lexical analysis is essentially a pattern matching algorithm; it searches an input subject string (source program) for occurrences of specific patterns (such as keywords and variable names), and outputs an analyzed program for subsequent use in the remaining phases.

The pattern matching process is applied in the latter phases also but to a lesser degree. In any event, the speed of the language compiler is closely related to the speed of its pattern matching portion.

Macro-processors (or preprocessors as they are often called) also rely on the pattern matching process. A macro-processor allows a user to specify his own syntax for a standard language, say, like Fortran. That is, the user "tells" the macro-processor what syntax constructs to look for (patterns) and what is to be done as the result of a pattern match. As a very simple example, suppose the user would like the occurrence of the string "QUIT" in the source coding to mean halt processing and close all files. He would therefore identify to the macro-processor the pattern "QUIT" and the coding necessary to perform the stop and close files operations. (This is done within the language of the macro-processor.) Thus, when a source program is submitted to the macro-processor, it is searched for occurrences of the pattern "QUIT" and at each such occurrence the associated coding would replace the string "QUIT". This updated (preprocessed) code is then sent to the regular language compiler which generates the final object code.

However, the previous example has perhaps over simplified the sometimes complex problem of pattern definition and matching. For instance, what if the user would like a pattern "IF#THEN" to match all occurrences of an "IF" follwed by a "THEN" where "#" represents an arbitrary number of interceding

characters. How is this pattern stored in the data structure? And how is the match algorithm to determine a success or failure?

Identical situations exist in the string and list manipulating languages like SNOBOL and LISP. Much work has been done in the study of such patterns and their representations (Ref 6). Even faster versions of SNOBOL (such as SPITBOL) have been developed, all of which owe much of their speedier operation to the design of the pattern data structure and associated matching algorithm.

Another user of the pattern matching process is the familiar text editor. Its use of the matching process is obvious. For example, when using the text editor, one is often scanning text (e.g. source code for a program) for occurrence of a string of characters and then replacing this string with another or even deleting it.

Bibliographic search is an amplified text editor scanning problem. In this case, a large amount of input text, perhaps an entire book, is searched for occurrences of key words (stored in a data structure) in order to identify references to these keywords.

Thus, the pattern matching process can be seen as an essential part of several computer applications programs:

1. Compilers
2. Macro-processors
3. String and list manipulating languages
4. Text editors

But, there are other, perhaps not so obvious, uses. These are within the operating system of the computer. For example, core allocation can use pattern matching to match a requested amount of memory (input subject string) against available blocks (patterns) in a list of available memory (data structure). Another example, file management algorithms can locate files by matching the queried name (subject string) against the names of current files (patterns) in the list of file names (data structure).

So, it is evident that the pattern matching process is an important function of computer operations, both from the user's viewpoint (applications programs) and from the analyst's viewpoint (operating system).

Thesis Objective

The purpose of this thesis is to investigate the performance of various data structures and associated pattern matching algorithms within the hardware constraints of the CDC CYBER 74 computer. Specifically, the investigation has been limited to pattern matching as described for bibliographic search and text editing. That is, the discrete-pattern is simply a string of characters, and therefore, the effect of substitution characters (such as the "#" mentioned before) and other advanced concepts will not be evaluated. Such topics cannot adequately be studied until the simpler cases are well understood. Thus, this thesis will provide a firm foundation for future studies into the implementation of such complex pattern representations on the CYBER 74. (See the recommendations chapter.)

Thesis Approach

Therefore, with this focus established, the thesis must first identify the various data structures available for use in the pattern matching process and the effect these structures have on the matching algorithm. Then, these data structures and algorithms must be implemented on the CYBER 74 and their relative performances evaluated. Key features of the CYBER 74 hardware will be identified and used during this evaluation, hopefully providing insight into more complex future implementations.

Data Structures and Associated Algorithms

There are many different data structures which may be used in the pattern matching process. With each unique data structure there is a corresponding match algorithm which interfaces with the structure. Some data structure designs will store one character per word of computer storage; others will store multiple characters. Each requires different handling by the match algorithm.

Simple List. Perhaps the least complicated data structure to work with is shown in figure 2. In this example, four patterns are stored in the data structure. Notice that each character of each pattern occupies a single word of storage. But, also notice the other entries in the data structure. These integers are the values of pattern length (LE_i) and pattern number (PN_i) associated with each pattern, i, in the structure. This structure is a simple list.

Pattern #	Pattern
1	HE
2	SHE
3	HAT
4	THEY

Data Structure

Location #	Contents	
1	2	LE_1
2	H	
3	E	
4	1	PN_1
5	3	LE_2
6	S	
7	H	
8	E	
9	2	PN_2
10	3	LE_3
11	H	
12	A	
13	T	
14	3	PN_3
15	4	LE_4
16	T	
17	H	
18	E	
19	Y	
20	4	PN_4

LE- Length in characters
PN- Pattern number

Figure 2. Simple List Data Structure

The pattern number represents that piece of information which is uniquely associated with each pattern. It could easily be a pointer to an associated routine or other function. The pattern length, however, may be considered an unnecessary but nice to have piece of data. Its value is included since it is known at the time the pattern is stored and therefore requires no overhead in calculation. Its convenience lies in its use as a "stepping" value so that one may search the structure by adding lengths. (This is opposed to a search made by examining each storage word for occurrence of a pattern number (stored in distinctive form such as a negative value) which would then indicate the beginning or end of a pattern.)

These two pieces of information, pattern number and pattern length, will be associated with each pattern in all of the following examples.

The pattern matching algorithm associated with this simple list will be equally simple, though not terribly efficient. That is, the algorithm begins its search with the first character of an input subject string and does not move to the next character until all patterns in the data structure are compared against all possible matches beginning with that character. It is easier shown than said, as is illustrated in figure 3.

Indexed Simple List. A modification to the simple list is to incorporate an index to point to (hàsh into) the location of the first occurrence of a pattern based on some index

	Algorithm	S	H	E	P	Results
1.	Begins search with - - - -	▲				
2.	Tries - - - - - - - - - - - - -	H				no match
3.	Tries - - - - - - - - - - - - -	S				match
4.	Continues - - - - - - - - - -		H			match
5.	Continues - - - - - - - - - -			E		match, PN_2
6.	Tries - - - - - - - - - - - - -	H				no match
7.	Tries - - - - - - - - - - - - -	T				no match
8.	Reaches end of data structure					
9.	Moves to next input char -		▲			
10.	Tries - - - - - - - - - - - - -		H			match
11.	Continues - - - - - - - - - -			E		match, PN_1
12.	Tries - - - - - - - - - - - - -		S			no match
13.	Tries - - - - - - - - - - - - -		H			match
14.	Continues - - - - - - - - - -			A		no match
15.	Tries - - - - - - - - - - - - -		T			no match
16.	Reaches end of data structure					
17.	Moves to next input char -			▲		
18.	Tries - - - - - - - - - - - - -			H		no match
19.	Tries - - - - - - - - - - - - -			S		no match
20.	Tries - - - - - - - - - - - - -			H		no match
21.	Tries - - - - - - - - - - - - -			T		no match
22.	Reaches end of data structure					
23.	Moves to next input char -				▲	
24.	Tries - - - - - - - - - - - - -				H	no match
25.	Tries - - - - - - - - - - - - -				S	no match
26.	Tries - - - - - - - - - - - - -				H	no match
27.	Tries - - - - - - - - - - - - -				T	no match
28.	Reaches end of data structure					
29.	Moves to next input char -					▲ DONE

(Columns S H E P are headed "Subject String".)

Figure 3. Search with Simple List Data Structure

value. A possible choice for index value might be to use first character lexical values. Figure 4 shows such a data structure and associated index. Note in the figure that the second pattern beginning with "H" is not indexed.

A match algorithm using this structure will, one would think, require fewer comparisons than the algorithm using a simple list. This assumption is borne out when figure 5 is compared to figure 3. Figure 5 represents the search steps using an indexed simple list for the same input subject string as figure 3. Notice that in the example the total number of steps has decreased from twenty-nine to twenty. This is a result of using an index to enter the data structure. Specifically, in figure 5, the construct "INDEX( )" refers to the search algorithm checking the index value of a particular character. For example, in step 2, the index value of S, INDEX(S), is equal to 5 as shown in figure 4. (This use of INDEX( ) will be consistent for the remainder of this paper.)

However, though the number of steps in the search has decreased, data structure size has been increased by the addition of the index table. It should also be noted that in the worst case, when all patterns begin with the same character, this structure will be no better than a simple list. In such a case, a better choice of indexing value might be to hash into the list based on the first two or three characters. In any event, the choice of a hash function is often very data dependent, and extensive studies

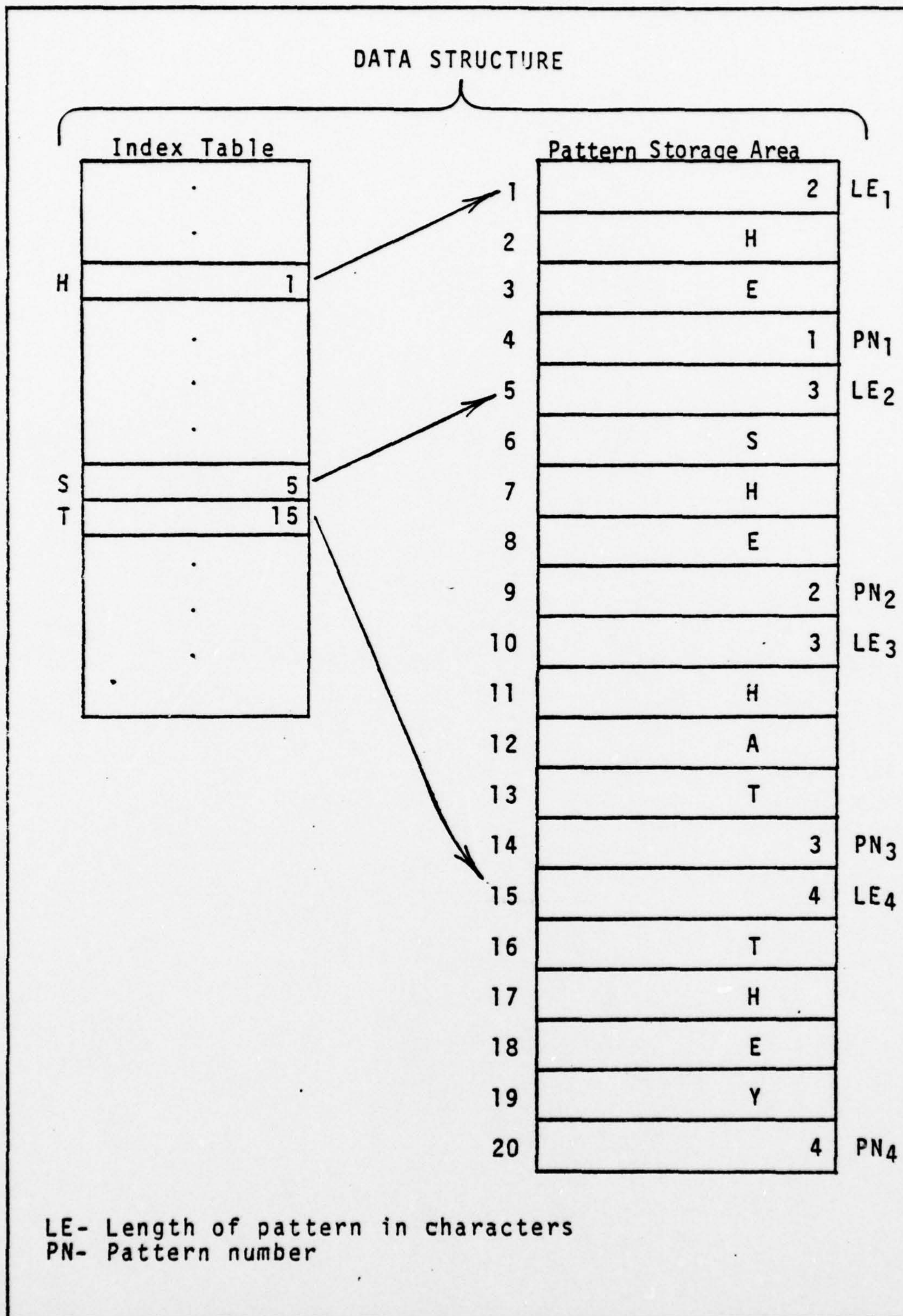


Figure 4. Indexed Simple List Data Structure.

	Algorithm	S	H	E	P	Results
1.	Begins search with - - - -	▲				
2.	INDEX(S)=5, Tries - - - - -	S				match
3.	Continues - - - - - - - - - -		H			match
4.	Continues - - - - - - - - - -			E		match, PN_2
5.	Tries - - - - - - - - - - - -	H				no match
6.	Tries - - - - - - - - - - - -	T				no match
7.	Reaches end of data structure					
8.	Moves to next input char - -		▲			
9.	INDEX(H)=1, Tries - - - - -		H			match
10.	Continues - - - - - - - - - -			E		match, PN_1
11.	Tries - - - - - - - - - - - -		S			no match
12.	Tries - - - - - - - - - - - -		H			match
13.	Continues - - - - - - - - - -			A		no match
14.	Tries - - - - - - - - - - - -		T			no match
15.	Reaches end of data structure					
16.	Moves to next input char -			▲		
17.	INDEX(E)=0, Fails on index					
18.	Moves to next input char -				▲	
19.	INDEX(P)=0, Fails on index					
20.	Moves to next input char -					▲ DONE

(Column headers S H E P appear under the heading "Subject String".)

Figure 5. Search with Indexed Simple List

into optimum selections have been documented by others, so this paper will not become involved in this area (Ref 10: 315-358).

Indexed Linked List. As the next logical "improvement" to the indexed list one might choose to link patterns of the same index value. Figure 6 shows how this data structure would appear; with each pattern, i, there is an associated link (LI_i).

Now, the search using this data structure is improved since only patterns with the same index value will be compared. Figure 7 shows this relationship when compared to the last example in figure 5 (seventeen steps to twenty steps).

Again, as in the indexed simple list, if all patterns were to have the same index value, then the search would be no better than a simple list. On the other hand, given a larger and more dispersed set of patterns the linked list approach would show an even more significant improvement.

Indexed Binary Tree. With the choice of a better index (hash) value, patterns beginning with the same character could be found uniquely, or with few sharing the same index value. However, with this approach the index table could grow disproportionately large when compared to the pattern storage area, or the hashing function could become overly complex. An alternative improvement to the indexed linked list is an indexed sorted linked list, of which the binary tree is a possible choice.

In the binary tree, there are two links associated with each pattern, a high link and a low link. As each new

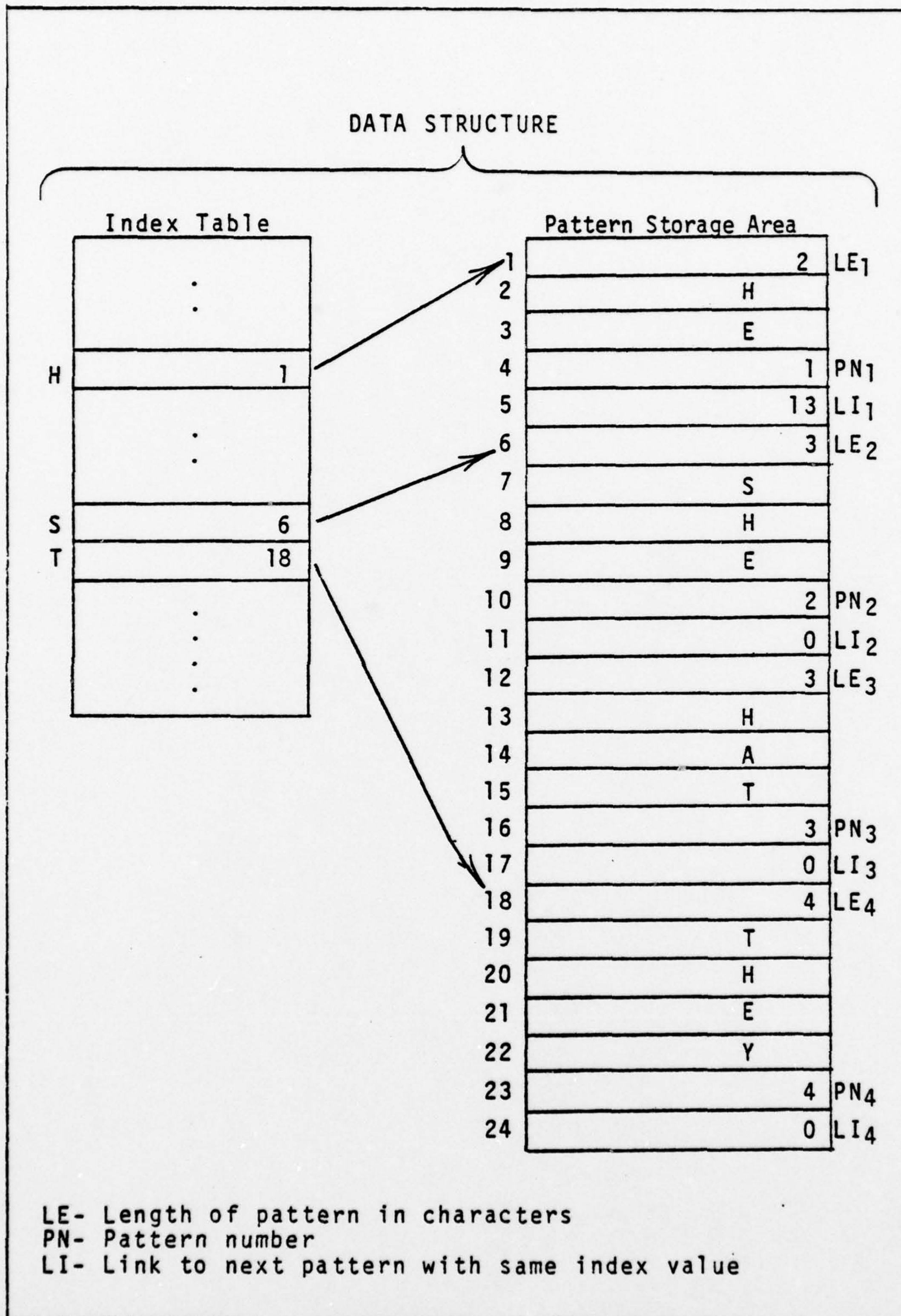


Figure 6. Indexed Linked List Data Structure

	Algorithm	Subject String: S	H	E	P	Results
1.	Begins search with - - - -	▲				
2.	INDEX(S)=6, Tries - - - - -	S				match
3.	Continues - - - - - - - - -		H			match
4.	Continues - - - - - - - - -			E		match, PN_2
5.	LI_2=0, Fails on link					
6.	Moves to next input char -		▲			
7.	INDEX(H)=1, Tries - - - - -		H			match
8.	Continues - - - - - - - - -			E		match, PN_1
9.	LI_1=13, Follow link					
10.	Tries		H			match
11.	Continues - - - - - - - - -			A		no match
12.	LI_3=0, Fails on link					
13.	Moves to next input char -			▲		
14.	INDEX(E)=0, Fails on index					
15.	Moves to next input char -				▲	
16.	INDEX(P)=0, Fails on index					
17.	Moves to next input char -					▲ DONE

Figure 7. Search with Indexed Linked List.

pattern is added to the data structure, its index value is first calculated. If no pattern exists with that index value, then the new pattern is entered immediately into the data structure. However, if a pattern already exists with that index value, then the new pattern and existing pattern are compared to see if the new pattern should be added out the high or low link of the old pattern. This comparison may be based on second characters, or some combination of characters. As in all indexed files, any attempt to optimize this hash function can be very data dependent.

Figure 8 shows a binary tree data structure for the same patterns as in previous examples. Because only two patterns share the same index value, "HE" and "HAT", the true worth of the binary tree is not clearly demonstrated. Its value will become more evident in later work. Note the additional link for each pattern for a total of two per pattern, a low link (LL_i) and a high link (HL_i).

A search using an indexed binary tree will require a little extra work to determine which link to follow. This overhead, on the average, should not cause the matching algorithm to perform less efficiently, and, as can be seen in the example in figure 9, required one less step than in the indexed linked list. However, these comparative results are hardly conclusive, but on the average, they can be expected.

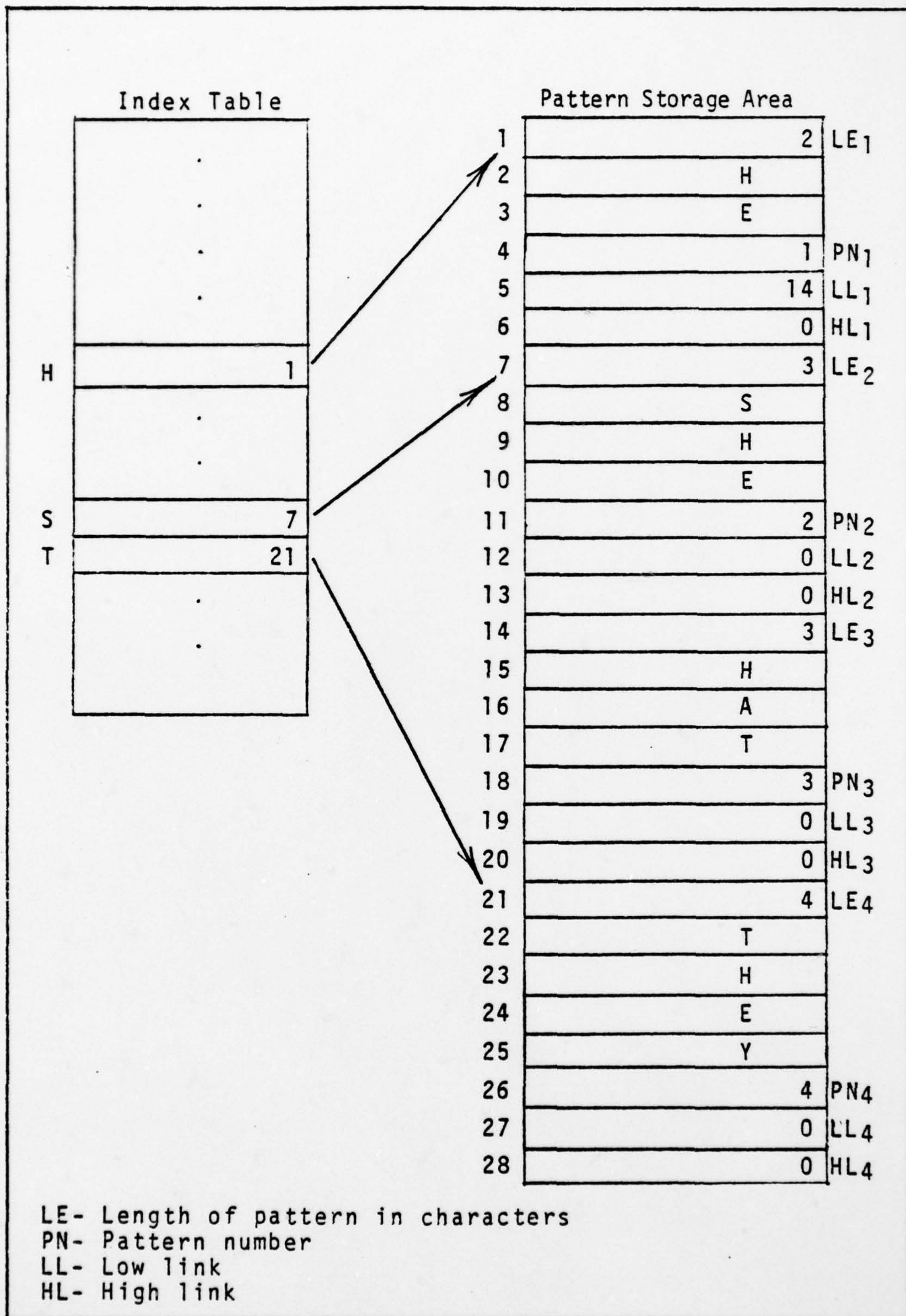


Figure 8. Indexed Binary Tree Data Structure

	Algorithm	Subject String S	H	E	P	Results
1.	Begins search with - - - -	▲				
2.	INDEX(S)=7, Tries - - - -	S				match
3.	Continues - - - - - - - - -		H			match
4.	Continues - - - - - - - - -			E		match, PN_2
5.	Compares second characters					check high link
6.	HL_2=0, Fails on link					
7.	Moves to next input char -		▲			
8.	INDEX(H)=1, Tries - - - -		H			match
9.	Continues - - - - - - - - -			E		match, PN_1
10.	Compares second characters					check high link
11.	HL_1=0, Fails on link					
12.	Moves to next input char -			▲		
13.	INDEX(E)=0, Fails on index					
14.	Moves to next input char -				▲	
15.	INDEX(P)=0, Fails on index					
16.	Moves to next input char -					▲ DONE

Figure 9. Search with Binary Tree

Indexed Alternate/Successor Linked List. In the preceding data structures each pattern has been stored in its entirety; there has been no way for similar patterns to share storage of duplicate characters. For example, if the patterns "HE" and "HAT" were stored, a minimum of five words were required. But if the "H" of the two words could share the same storage location, then the characters of the two patterns could be stored in four words. This is the concept behind the alternate/successor linked list. The study of this data structure was motivated by the similar structure used within SNOBOL (Ref 6).

However, in building such a structure there is considerable overhead involved in maintaining character relationships, e.g., in the last example, the "H" belongs to two patterns, "HE" and "HAT". This overhead is diminished as the proportion of duplicate leading characters is increased.

Specifically, associated overhead can be seen in figure 10. With each character there is an alternate and successor field to identify the character relationships. There is also the necessity to include a pattern number field for each character since at any character there may be a possible pattern number. Therefore, the size of the data structure has increased quite a bit when compared to the previous examples.

However, with increase in size, the search algorithm for the pattern matching process has become less complicated. Figure 11 shows how the algorithm procedes. When compared

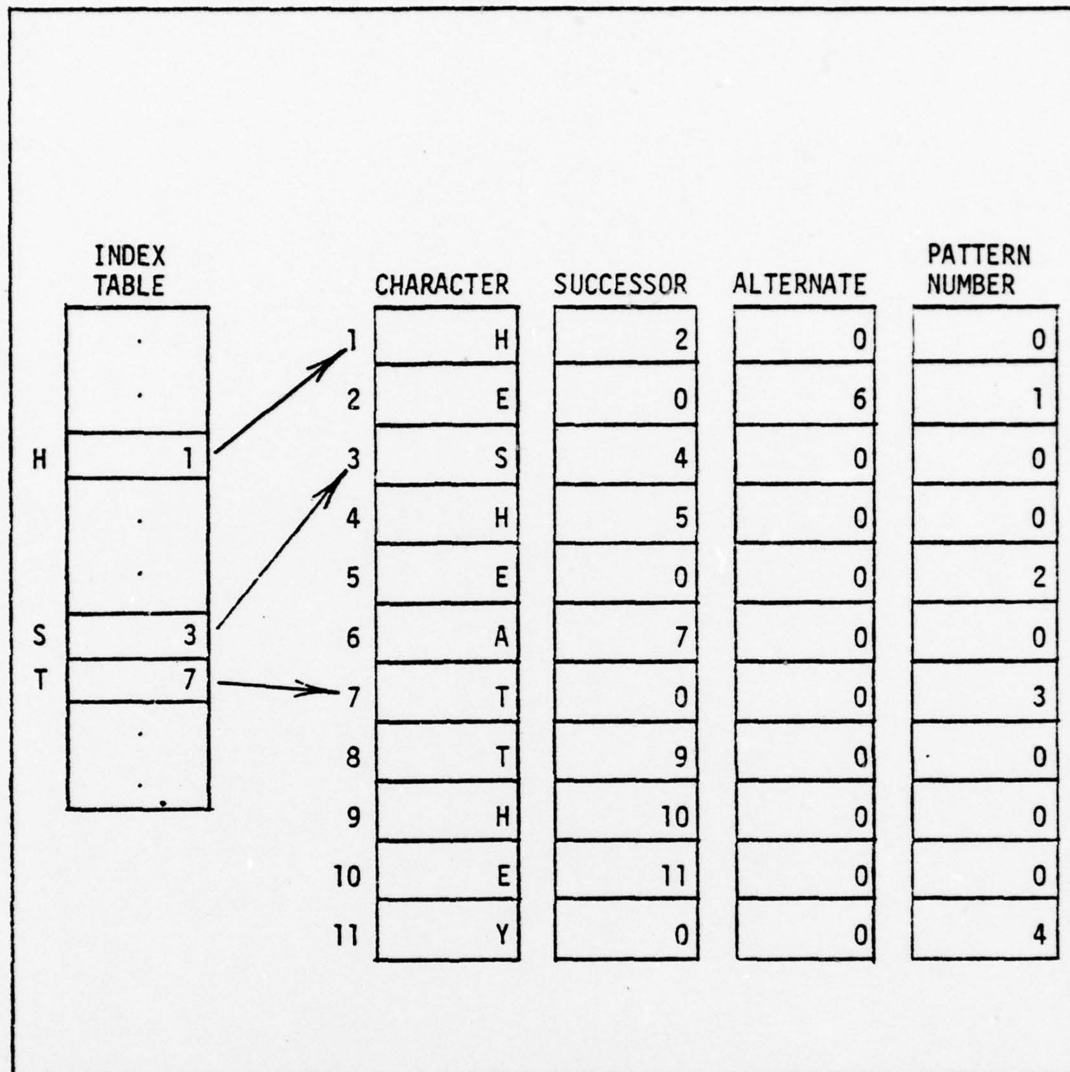


Figure 10. Indexed Alternate/Successor Linked List Data Structure

to the binary search, there are fewer steps in the processing of the same input subject string.

Therefore, of the five structures thus far discussed, the alternate/successor approach appears to allow the most rapid pattern matching search. But, is there room for improvement? The answer is yes.

Finite State Automata Linked List. Refer to figure 11. In steps 3 and 4, an "H" and an "E" are matched to find the pattern "SHE" in "SHEP". Then in steps 7 and 8, an "H" and an "E" are again matched to find "HE" in "SHEP". A more efficient search would be to find not only "SHE" but "HE" at the same time; in effect the search would become "no back-up". That is, as the algorithm examines each character of the input subject string it never backs up. Such a search can be achieved through the design and construction of the data structure.

The data structure herein described which allows a no back-up search will be called a finite state automata (f.s.a.) linked list. Its concept is taken from an article by Alfred Aho and Margaret Corasick (Ref 1). Briefly, in a finite state automata, at any given time, one is aware of only two things: (1) the current state, and (2) the current input character.

An actual implementation of an f.s.a. data structure will be illustrated later. But in figure 12 there is an example of the three "functions" required to realize the f.s.a. approach in the pattern matching algorithm. These

Algorithm	S	H	E	P	Results
1. Begins search with - - - -	▲				
2. INDEX(S)=3, Tries - - - - -	S				match
3. SUCCESSOR(3)=4, Tries - - -		H			match
4. SUCCESSOR(4)=5, Tries - - -			E		match, PN_2
5. SUCCESSOR(5)=0, Fails on successor link					
6. Moves to next input char -		▲			
7. INDEX(H)=1, Tries - - - - -		H			match
8. SUCCESSOR(1)=2, Tries			E		match, PN_1
9. SUCCESSOR(2)=0, Fails on successor link					
10. Moves to next input char -			▲		
11. INDEX(E)=0, Fails on index					
12. Moves to next input char -				▲	
13. INDEX(P)=0, Fails on index					
14. Moves to next input char -					▲ DONE

(Columns S H E P are headed "Subject String".)

Figure 11. Search with Alternate/Successor Linked List

functions are the GOTO, FAIL, and OUPTUT functions.

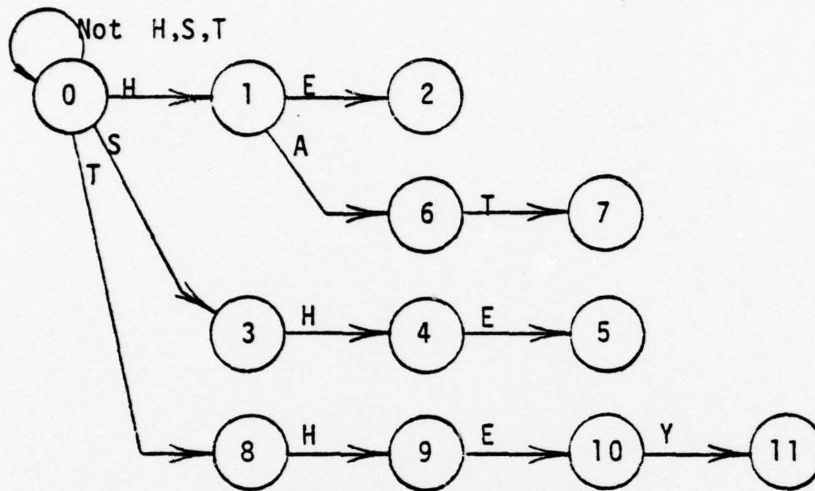
The GOTO function determines which state to "go to" given a current state and input character. For example, in figure 12 (a), GOTO(0,H) equals 1; that is, given state 0 and input character "H", the next state to enter is 1.

The second function, FAIL, is used to determine which state to enter, given that the GOTO function has failed. (The GOTO function fails when it is not defined for a given character and state.) For example, GOTO(4,X) fails, and FAIL(4) equals 1; therefore, state 1 is the state to enter from state 4 if the input character is "X". (Notice that this is also the case for any input character other than an "E" in state 4.)

The OUTPUT function is referenced as each state is entered to determine if a pattern match has occurred. As shown in figure 12 (c), OUTPUT may signal that one or more, or no patterns have been matched given a current state. For example, OUTPUT(6) is the null set, but OUTPUT(5) is the set containing both "SHE" and "HE". Therefore, when state 6 is entered there is no pattern match, but when state 5 is entered both patterns "SHE" and "HE" have been matched.

Figure 13 shows the relatively simple search using the same input subject string and patterns as in previous examples. However, preparing the data structure and determining the three functions is much more complicated and costly as will be shown latter.

Thus, it appears that the pattern matching algorithms



(a) GOTO function

State	0	1	2	3	4	5	6	7	8	9	10	11
FAIL(State)	0	0	0	0	1	0	0	8	0	1	0	0

(b) FAIL function

State	OUTPUT(State)
2	{HE}
5	{SHE,HE}
7	{HAT}
10	{HE}
11	{THEY}

(c) OUTPUT function

Figure 12. Functions to Implement Finite State Automata Algorithm

	Algorithm	S	H	E	P		Results
1.	Begins search with - - - -	▲					
2.	INDEX(S)=3						OUTPUT(3)= null
3.	Moves to next character - -		▲				
4.	GOTO(3,H)=4						OUTPUT(4)= null
5.	Moves to next character - -			▲			
6.	GOTO(4,E)=5						OUTPUT(5)= SHE,HE
7.	Moves to next character - -				▲		
8.	GOTO(5,P)=fail						
9.	FAIL(5)=0						
10.	Moves to next character - -					▲	DONE

(Column group S H E P is headed "Subject String".)

Figure 13. Search with Finite State Automata Data Structure

associated with the data structures so far discussed may be ranked from fastest to slowest:

1. Search using finite state automata linked list.
2. Search using indexed alternate/successor linked list.
3. Search using indexed binary tree.
4. Search using indexed linked list.
5. Search using indexed simple list.
6. Search using simple list.

However, the ranking of the size of the data structures is also in the same order, from largest to smallest. At first, one might suggest packing the patterns (placing more than one character per computer word of storage) as a solution to this problem. And as shown in figure 14, a packed simple list does indeed save space over the unpacked version, eight words compared to twenty. (Recall figure 2.) But now each packed character must be unpacked in order to perform comparisons during the matching process.

The question, then, is can a savings in space be made without an unacceptable sacrifice in speed? In fact, can the speed of the matching algorithm actually be improved by taking advantage of the fewer accesses which must be made to the data structure? For example, only two words must be retrieved to obtain the pattern "HAT" and related values in the packed example of figure 14, whereas, five words must be retrieved in the unpacked version of figure 2. Another thought, perhaps the match algotithm can perform multi-character comparisons without unpacking the pattern and

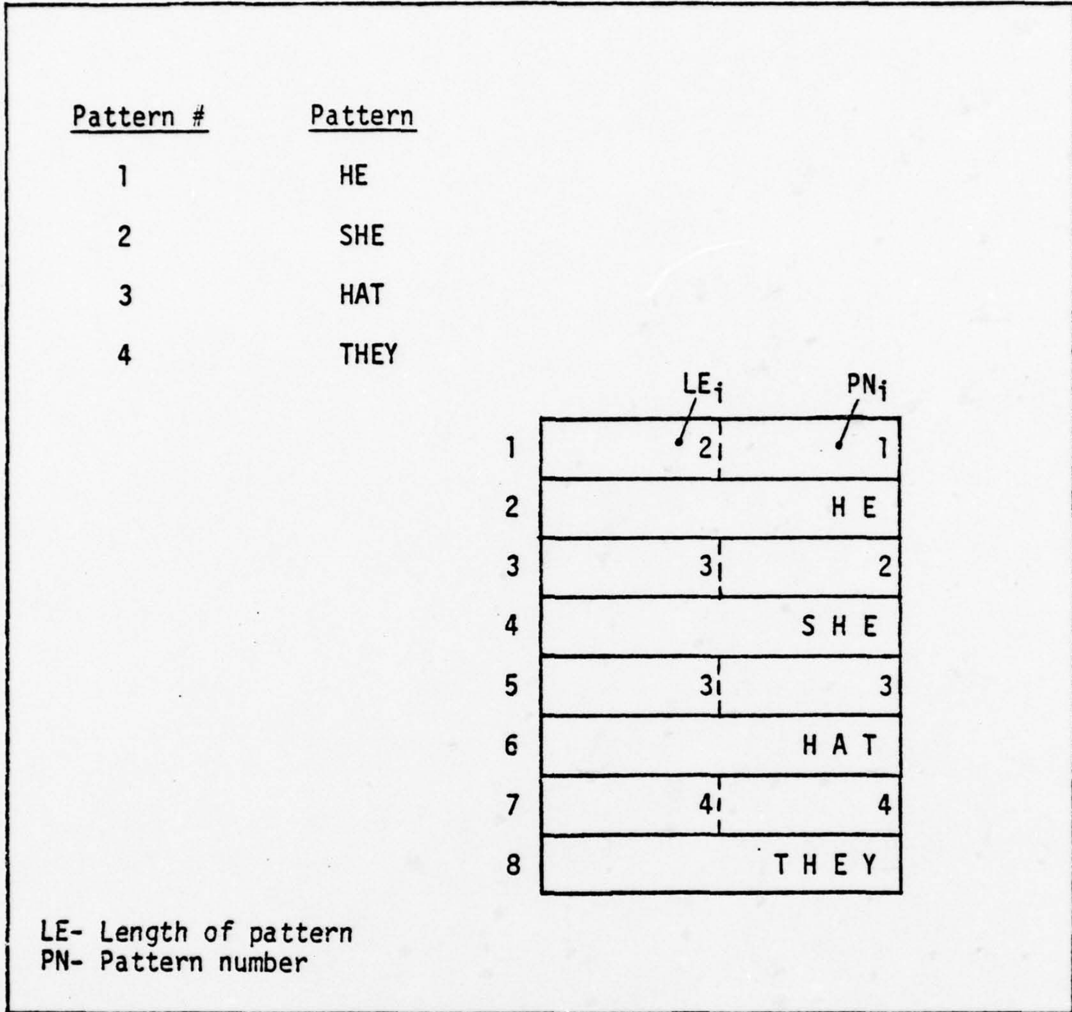


Figure 14. Packed Simple List Data Structure

thereby, too, increase speed.

It is in the interest of resolving these questions and ideas that the thesis work was conducted. In the following chapter the various approaches used during the thesis will be discussed. In particular, machine dependent (CYBER 74) features will be identified and an attempt to quantify expected results will be made. In Chapter III the test procedures will be explained, and in Chapter IV, a discussion of the results and their implications will be made.

II. CYBER 74 Implementations

In this chapter, the many programs that were actually implemented on the CDC CYBER 74 will be presented. There were a total of twelve different configurations based on type of data structure, type of subject input (packed or unpacked), and whether the data structure itself was packed or unpacked.

Test Program Design

The first step in constructing the programs was to develop a design around which all programs could be built. It was necessary that this design allow each matching algorithm (that portion of the program which searched the subject string for pattern matches) to be timed separately from input and output constraints. It was also decided that the data structure should be able to be modified independently of the search routine, and that the data structure should be "hidden" from the search routine as much as possible. The "bubble chart" (Ref 4) of the resulting design is shown in figure 15.

The afferent (input) portion of the program consists of the modules which access the input file and present to the central transforms (processing modules) an input record. This input record, in theory, can consist of any number of characters, either packed or unpacked. Now, this record may be a pattern which is to be added to or deleted form the data structure, or it may be a subject string (text) which is to

Figure 15. Bubble Chart for Test Program Design

be searched for pattern matches. It is the function of the central transform "BUILD INFO RECORD" to determine which of these possibilities the input record fills and modify the record as necessary to produce "Pattern Info" or "Text Info" as appropriate. If the record is a pattern, it is sent to the efferent (output) module which handles modifications to the data structure along with the associated command indicating the pattern is to be added to or deleted from the data structure. On the other hand, if the input record is text, then it is passed to the other central transform "SEARCH FOR MATCH" which does just that, i.e., processes the input record looking for pattern matches. Match information is then relayed to the "OUTPUT MATCH INFO" module which will identify the results of the matching algorithm.

The corresponding structure chart is shown in figure 16. Here, the actual program elements and the data that flow between them are pictured. As implied in this chart, the executive module controls the overall sequencing and hence execution of the other modules.

The module "GETREC" and subordinate modules represent the single afferent branch of the bubble chart, inputting information to the executive. The two central transforms are represented by "BINFO" and "SEARCH". Their functions are those as described for the corresponding "bubbles". Notice that "SEARCH" accesses the data structure through the module "GETPAT". In this manner the data structure is effectively hidden and the coding for "SEARCH" will not

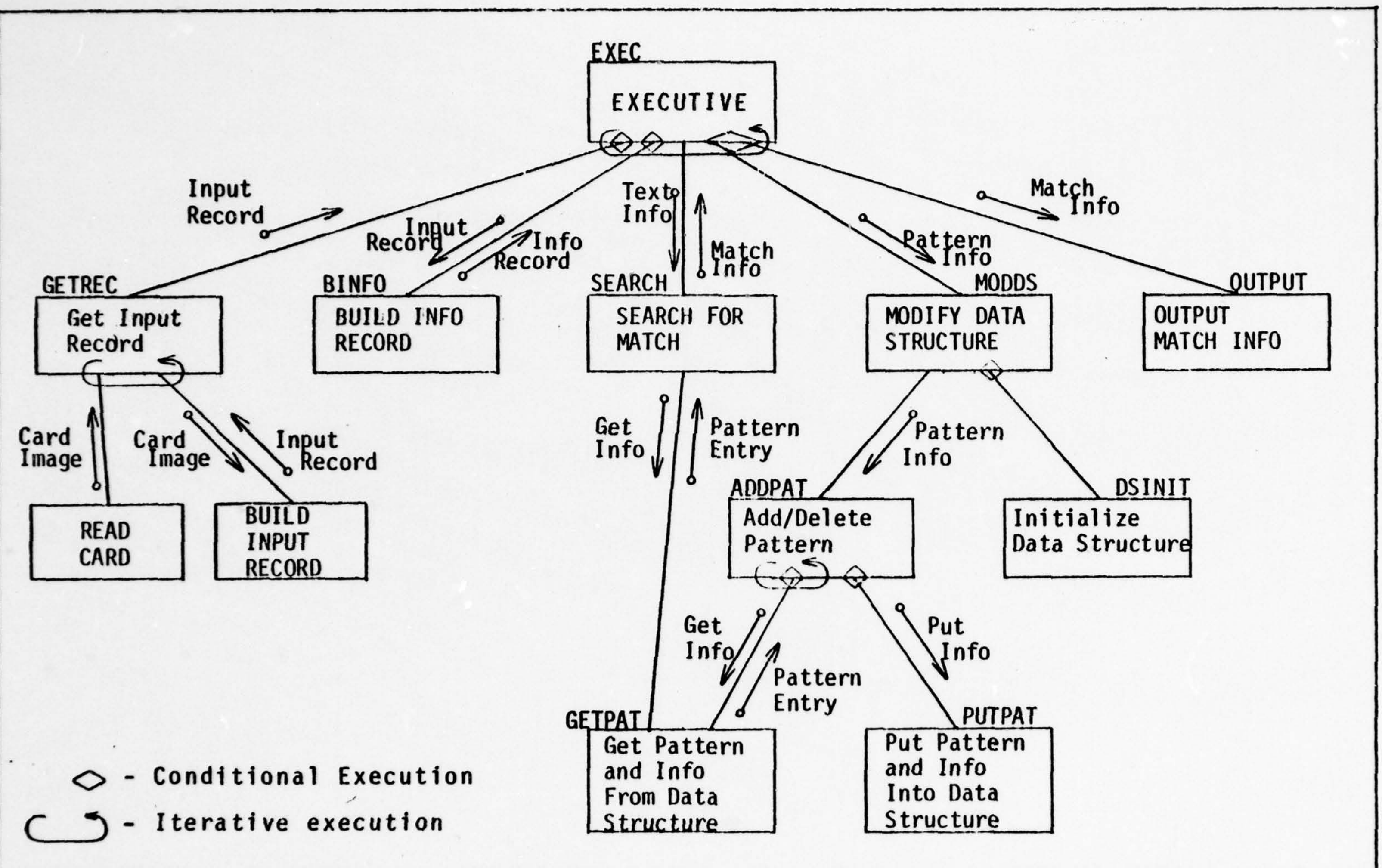


Figure 16. Structure Chart for Test Program Design

become involved in actual data structure manipulation. (Albeit, the search module is highly related to the design of the data structure.)

The remaining efferent branches of the bubble chart are fulfilled by the modules "MODDS" and "PUTOUT". The subordinate modules to "MODDS" represent those functions that may be necessary when modifying the data structure--initialize, add, or delete. Again, the data structure is hidden from "MODDS" also, through "GETPAT" plus the module "PUTPAT". (It sould be mentioned that within the context of this thesis, a deleted pattern will not necessarily free any storage space. In any event, pattern deletions and associated "garbage collection" procedures will not be considered during performance evaluation.)

Thus, the initial design criteria are established by the structure chart in figure 16. All programs were coded within this design using CDC Fortran IV Extended. Some departures from the general design did occur and will be mentioned during the description of the implementation. Essentially, only the modules "SEARCH", "GETPAT", and "PUTPAT" varied from one program to another.

In the remainder of this chapter, each program will be uniquely identified by its six character name, e.g., EXEC1A. The fifth character will always be a number and refers to the type of data structure implemented. The sixth character will be the letter "A", "B", or "C" referring to the type of input and data structure used. Figure 17 lists these program names

Program Name	Type of Data Structure	Input Text	Data Structure
EXEC1A	Simple List	Unpacked	Unpacked
EXEC2A	Indexed Simple List	Unpacked	Unpacked
EXEC3A	Indexed Binary Tree	Unpacked	Unpacked
EXEC4A	Alternate/Successor Linked List	Unpacked	Unpacked
EXEC6A	Finite State Automata	Unpacked	Unpacked
EXEC1B	Simple List	Unpacked	Packed
EXEC2B	Indexed Simple List	Unpacked	Packed
EXEC3B	Indexed Binary Tree	Unpacked	Packed
EXEC4B	Alternate/Successor Linked List	Unpacked	Packed
EXEC5B*	Alternate/Successor Linked List	Unpacked	Packed
EXEC3C	Indexed Binary Tree	Packed	Packed
EXEC4C	Alternate/Successor Linked List	Packed	Packed

*- Data structure for EXEC5B is same as EXEC4B; however, the two differ in method of search. (See text.)

Figure 17. Programs Developed to Test Pattern Matching Algorithms

and identifies their characteristics.

Unpacked Approaches

EXEC1A (Simple List). The first program developed, EXEC1A, used a simple list data structure, and both the input subject text and the data structure were unpacked. This approach was chosen as the first implementation for several reasons:

1. It was simple,

2. A simple program was needed to validate the initial software design,

3. It would provide a base upon which other programs could be built and compared, and

4. Its searching logic closely approximates an intuitive approach.

The resulting data structure of EXEC1A was identical to the one pictured in figure 2 of the last chapter. Each pattern was entered into the data structure, one character per word, right justified, with zero fill. Thus, each six bit character was essentially an integer value (0 - 63). (Appendix A shows the CDC display code for legal characters.) With each pattern there were also two words storing the values for pattern number and pattern length.

The search algorithm was coded to perform the matching function as shown in figure 3. Since the input text was also unpacked (right justified, zero fill) character comparisons were simple checks for integer equality. Due to the simplicity of this method, there were no difficulties in implementation.

EXEC2A (Indexed Simple List). As an improvement to the simple list, EXEC2A, with its indexed list, was the logical choice to implement next. The index value for a given pattern was chosen to be the display code value of the first character plus one. Thus, the index values ranged from one to sixty-four, and so, the index table was established as sixty-four words long. Its function then, was as illustrated in figure 4, to point to the first occurrence of a pattern with a given first character.

The search algorithm coding for EXEC2A was almost identical to EXEC1A. The only change was to reference the index table when looking for a possible first character match for each new input character.

EXEC3A (Indexed Binary Tree). Referring to Chapter I, an indexed linked list followed an indexed list in order of expected efficiency. However, EXEC3A implements an indexed binary tree, skipping the linked list. This choice was an arbitrary omission based on an interest in other implementations. Due to time constraints, one implementation was chosen to be omitted from the thesis; the indexed linked list was the one eliminated.

The indexed binary tree data structure of EXEC3A was similar to the one pictured in figure 8. As in the other implementations, each pattern character was stored right justified, zero filled, in a word of storage. High and low pointer values were established based on comparison of second character display code values. That is, if a new pattern were

entered in the data structure and its second character were greater than or equal to the second character of an existing pattern, then the new pattern would be added out the high pointer. Otherwise, it would be added out the low pointer. If the new pattern were only a single character, then it was placed at the head of the tree and tied to the index table regardless of existing patterns. Figure 18 shows a set of patterns and the binary tree that would result if the data structure building algorithm of EXEC3A were applied. (A note to the reader--the figure is not meant to imply a packed data structure, just that with each pattern there are associated high and low pointers.)

The choice of second character to determine high or low order was perhaps not the best, since, if there were many patterns all with the same first and second character, then the binary tree would become merely a linked list. However, the expected input during the test was not to be of this form and therefore, would not require a "better" hash function.

The search algorithm for EXEC3A was coded to perform the pattern matching algorithm using the unpacked binary tree data structure and unpacked input text.

EXEC4A (Alternate/Successor Linked List). EXEC4A was developed to use the alternate/successor linked list data structure described in Chapter I. As mentioned then, this structure is much more complex than the others so far developed; two pointer words and one pattern number word are needed for each character stored in the data structure. Because of

Patterns and the order entered into data structure are:

1. HIGH
2. HOT
3. HEM
4. HIDE
5. HAM
6. HOPE
7. HAT

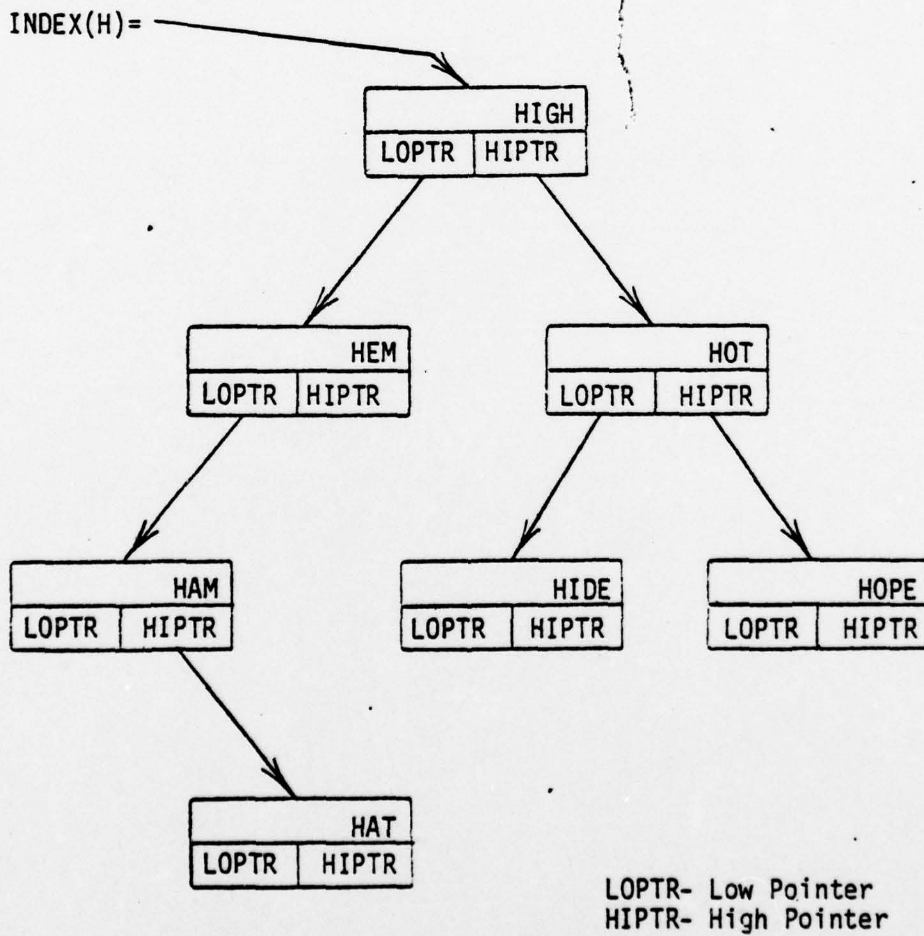


Figure 18. Binary Tree as Would be Constructed by EXEC3A.

this "extra" information, slight departures were made from the initial program design of figure 16.

In particular, the module "ADDPAT" was given direct access to the data structure rather than working solely through the data hiding modules "GETPAT" and "PUTPAT". As a matter of fact, "GETPAT" and "PUTPAT" too, were changed for EXEC4A. Their new functions were to manipulate entries corresponding to single characters rather than to work with entire patterns.

The search routine, however, presented no departures from the original design. In fact, the complexity of constructing the data structure actually had an inverse effect on the search module, decreasing the difficulty of its coding.

Packed Approaches

After EXEC4A was implemented, rather than proceeding to the more complex finite state automata data structure, it was decided that more advantage should be made of the CDC hardware given the then completed programs, EXEC1A, EXEC2A, EXEC3A, and EXEC4A. The obvious idea was to make use of the large (sixty bit) CDC computer word, that is, to pack information. In this way, up to ten characters could be placed in a single word, thereby reducing data structure storage requirements plus, it was hoped, decrease search time through fewer accesses to the data structure.

Another hardware concept identified as nice to test, would be to make more efficient use of the CPU registers

during the search process. However, this would most certainly require machine level coding (COMPASS), and it was decided that this avenue would not be pursued at this time.

EXEC1B (Simple List). In making use of a packed data structure, EXEC1A was the first program to be modified producing the new program, EXEC1B. Figure 19 is the resulting data structure produced by EXEC1B. When this is compared to the unpacked version in figure 2, one can see that indeed, much space can be saved, especially when dealing with long patterns such as number five, "LONGER THAN TEN". Notice also, that the length of the pattern in both words (LW_i) and characters (LC_i) along with the pattern number are packed into a single word.

The pattern is packed left justified, and the fill is unimportant. In this way a particular character may be selected by a simple left circular shift and mask operation based on a desired character position. For example, in pattern number four, if one wished to examine character position three, the Fortran Extended statement would be:

```
CHAR=SHIFT(PATTERN,CHARPOS*6).AND.(.NOT.MASK(54))
```

where PATTERN is the pattern "THEY", and CHARPOS has the value three. After execution of this statement, "CHAR" would contain the single value representing the letter "E", which is the desired answer.

The search algorithm then, was written to perform this unpacking operation as each character of a pattern was

Pattern #	Pattern
1	HE
2	SHE
3	HAT
4	THEY
5	LONGER THAN TEN

Word	LW_i	LC_i	PN_i
1	1	2	1
2	H E		
3	1	3	2
4	S H E		
5	1	3	3
6	H A T		
7	1	4	4
8	T H E Y		
9	2	15	5
10	L O N G E R ƀ T H A		
11	N ƀ T E N		

LW- Length of pattern in words
LC- Length of pattern in characters
PN- Pattern number

Figure 19. Packed Simple List Data Structure of EXEC1B.

needed. Otherwise, its operation was identical to that of the algorithm in EXEC1A.

EXEC2B (Indexed Simple List). The next program to incorporate a packed data structure was EXEC2B. Like EXEC1B, EXEC2B was a redesign of its unpacked counterpart. The resulting data structure was identical to that of EXEC1B with the addition, of course, of the index table. The index table could itself have been packed, say, by placing four index values per word. However, it was decided not to do this since the overhead in obtaining a packed index value would not be worth the minimal savings in space.

There were no problems in implementing EXEC2B.

EXEC3B (Binary Tree). EXEC3B is the packed data structure version of EXEC3A. As in EXEC2B, the index table was not packed, and each pattern had its related information packed into a single word. Figure 20 shows the packed structure of EXEC3B. One can see that each pattern is stored left justified as in the other packed implementations.

Therefore, the resulting change to the search module of EXEC3A in order to work for EXEC3B was the same shift and mask operation described for EXEC1B.

EXEC4B (Alternate/Successor Linked List). The changes thus far described to achieve packed data structures have been relatively easy to implement, though not trivial. However, achieving a packed version of the alternate/successor linked list was not nearly so simple. The first question that came to mind was just what do you pack, or

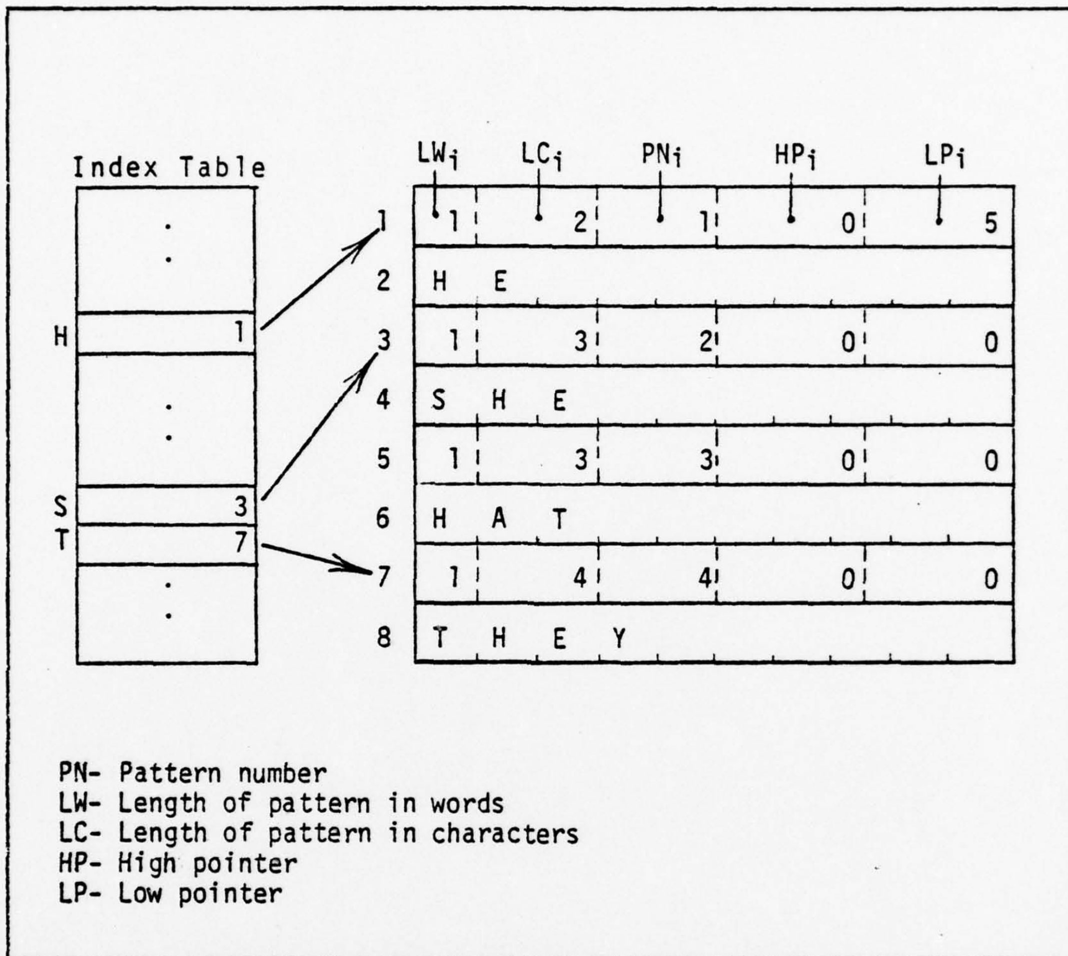


Figure 20. Packed Binary Tree Data Structure of EXEC3B

more appropriately, what can be packed?

The answer to this question was an involved approach that allowed as much of a pattern to be packed as possible. Specifically, shared leading characters of two or more patterns were packed in a word and the remaining unique characters of each pattern packed in separate words. The packed portions were linked by successor and alternate links as in EXEC4A. Figure 21 shows step by step how the data structure would be built by EXEC4B for the given patterns.

One can see that the data structure can quickly become complicated. Nonetheless, the extra time spent in building the data structure was expected to provide decreased time in the actual pattern matching algorithm.

So, at this time all unpacked pattern matching programs had been converted to packed operation. However, the packed patterns, once retrieved from the data structure were still being unpacked for comparisons during the pattern matching search. With up to ten characters available for a single comparison, it was reasoned that perhaps multi-character comparisons could be achieved using packed input text and thereby, speed up the search. This was the motivation behind creating EXEC3C and EXEC4C.

EXEC3C (Binary Tree, Packed Input). The data structure building and accessing modules for EXEC3C were used as coded from EXEC3B. (Figure 20 shows the data structure.) However, the search module was very much changed.

In EXEC3C the concept was to bring packed text into the

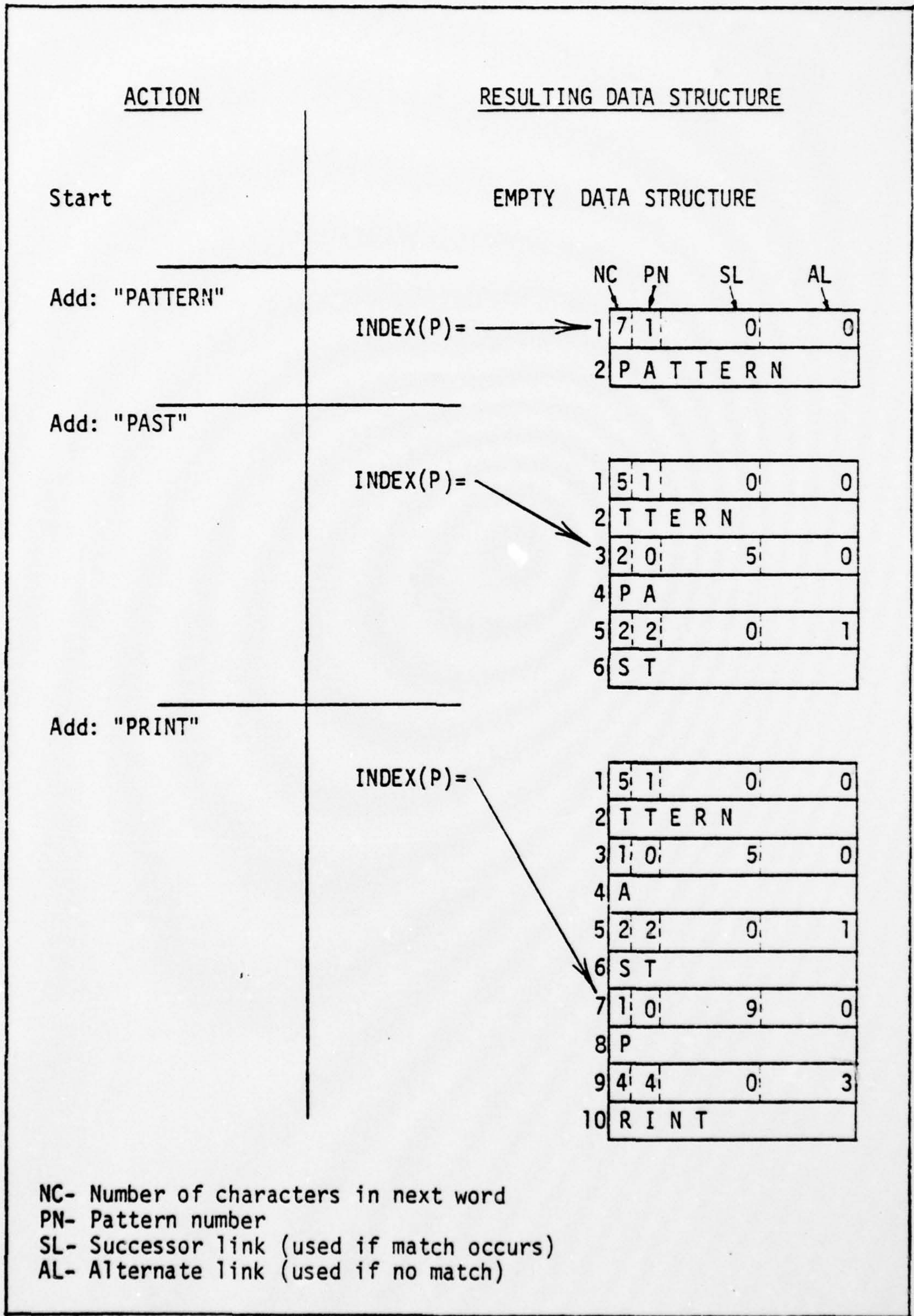


Figure 21. Sample Data Structure Construction Steps for EXEC4B

search module and search for pattern matches using direct comparisons between the text and packed patterns; there was to be no unpacking. In this way, up to ten characters could be tested for equality by a single comparison. Also, 10,000 characters of input text in previous implementations required a 10,000 word buffer if brought in all at once, but in EXEC3C (and EXEC4C) the same number of characters could be brought in in a 1,000 word buffer--a ten to one savings in space. Equivalently, given the same size buffer area, EXEC3C need refill it only a tenth as often as for the other programs, another possible increase in search performance.

The search module was coded with these ideas in mind--multi-character comparisons with packed input text and patterns. The approach works fine when the pattern occurs within the text in the same relative position as it is stored in the data structure, i.e., left justified within a word of storage. Figure 22 (a) shows this concept. However, if the pattern occurs across word boundaries, then either the text must be unpacked and repacked to form an occurrence as in (a) or the pattern must be "broken-up" like in the text, and therefore, two comparisons must be made. This latter alternative was chosen for EXEC3C and is shown conceptually in figure 22 (b).

Specifically, the search module for EXEC3C was implemented requiring three times as much code as the longest search module of the previous programs.

Packed Text: HERE IS A | PATTERN IN | A STRING.

Step 1. Get packed pattern from data structure . PATTERN

Step 2. MASK word two of input text to get. . . . PATTERN

Step 3. Compare the two to get complete match

(a) Occurrence at Beginning of Text Word

Packed Text: HERE A PAT | TERN CROSS | ES A WORD.

Step 1. SHIFT and MASK word one of text to get. . . PAT

Step 2. MASK pattern to get. . . . . PAT

Step 3. Compare the two to get partial pattern match

Step 4. MASK (no SHIFT) word two of text to get . . TERN

Step 5. SHIFT and MASK pattern to get . . . . TERN

Step 6. Compare the two to complete pattern match

(b) Occurrence Across Word Boundary

Figure 22. Pattern Matching as done in EXEC3C and EXEC4C.

EXEC4C (Alt/Suc Linked List, Packed Input). EXEC4C was implemented essentially be taking EXEC4B and replacing its search module with one coded similarly to EXEC3C. Therefore, the data structure of EXEC4C is identical to that shown in figure 21, and its search algorithm performed as described for EXEC3C and as illustrated in figure 22.

Special Approaches

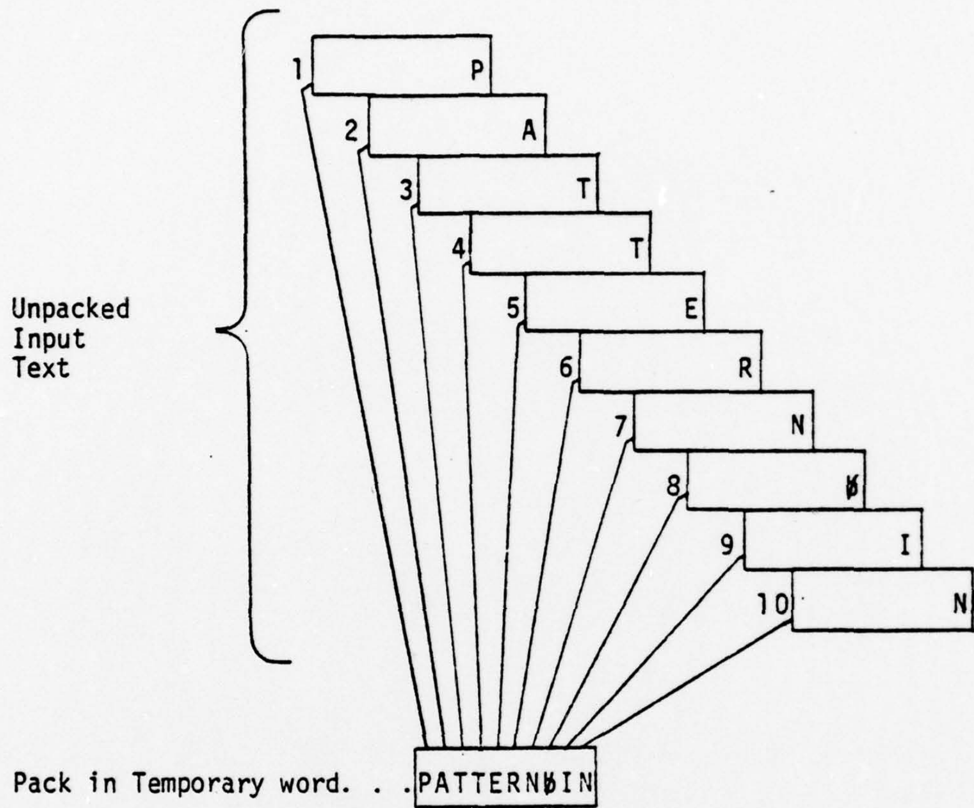
So, with the completion of EXEC4C there were ten programs with which to enter testing. However, preliminary analysis of the check-out runs of these programs suggested another approach might be taken resulting in a new program, EXEC5B.

EXEC5B (Special Alt/Suc). EXEC5B was a modification to EXEC4B. It was the result of two observations made during the debug runs of the previously completed programs:

1. EXEC4B appeared to be producing satisfactory search times despite its possibly complex data structure.

2. EXEC3C and EXEC4C did not seem to be doing so well with the packed input text and multi-character comparisons.

So, EXEC5B was designed making use of the data structure of EXEC4B (same as EXEC4C) and combining the unpacked text approach with the ability to perform multi-character comparisons which motivated the creation of EXEC3C and EXEC4C in the first place. The concept of this approach is shown in figure 23. The unpacked input text is packed into a single word, and patterns are compared against this packed word. If the pattern match continues beyond the ten packed characters, the comparisons are then made character by character as in



Now comparisons may be made against this packed word.

Step 1. Get pattern from data structure . . . . PATTERN

Step 2. MASK temporary word to get . . . . . . PATTERN

Step 3. Compare to get complete pattern match.

Figure 23. Concept of Search as Used in EXEC5B

EXEC4B. On the other hand, if there are no pattern matches, then new input characters are shifted into the temporary word from the right and comparisons are made against this new temporary word.

With the completion of EXEC5B there were eleven machine dependent pattern matching programs implemented. The final program to be built was designed using the algorithmic description of the finite state automata approach presented by Aho and Corasick (Ref 1).

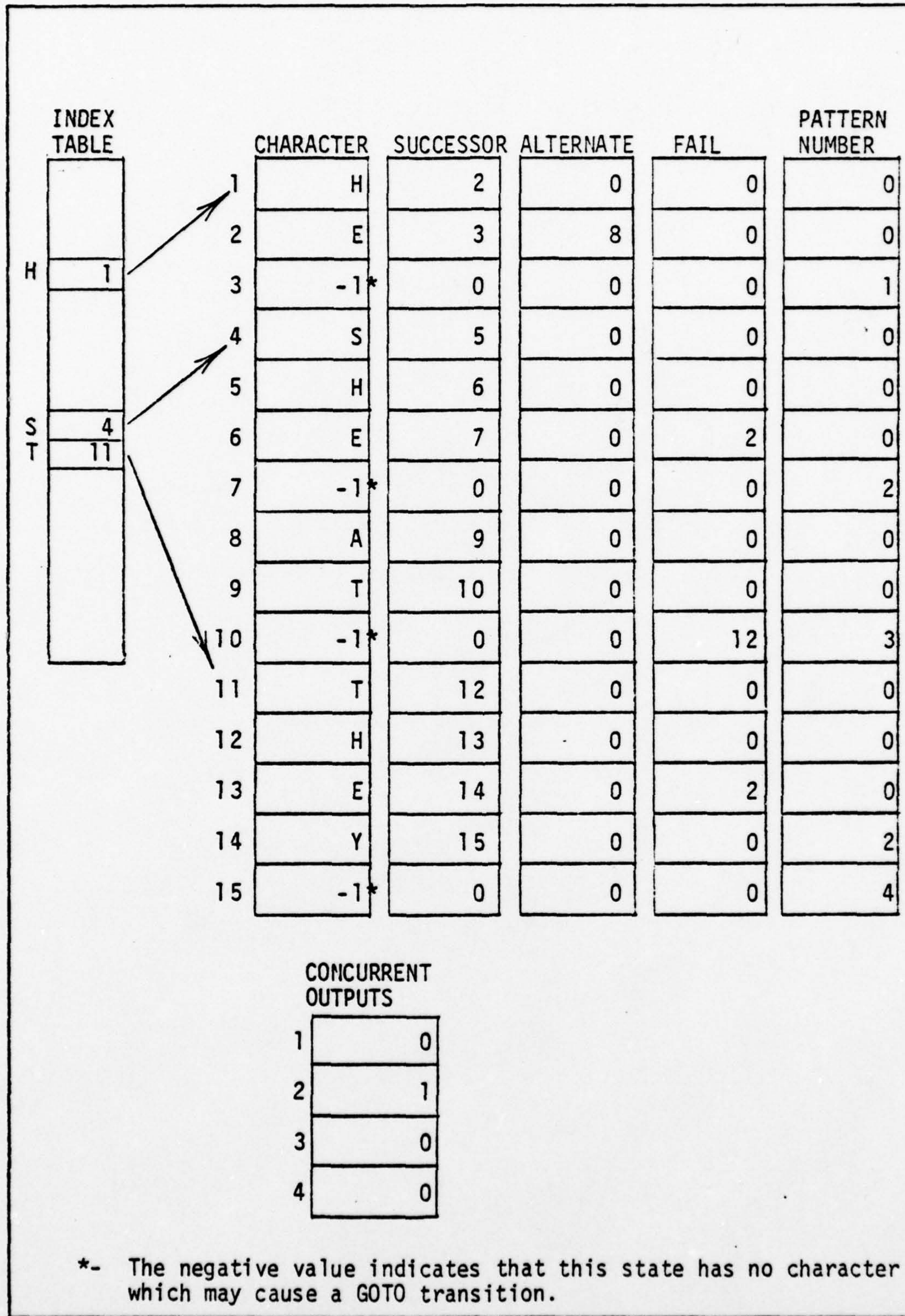
EXEC6A (f.s.a.). The finite state automata approach of EXEC6A was achieved using a modified version of the data structure in EXEC4A--the linked list. And the search algorithm was coded to perform the three functions, GOTO, FAIL, and OUTPUT as described in Chapter I.

Construction of the data structure, from a simplistic viewpoint, was a two step process. First, patterns were entered into the data structure using slightly modified routines from EXEC4A. Then, a special module was invoked which calculated the remaining values to be stored in the data structure; these were the FAIL and OUTPUT values for each state.

Figure 24 shows the finite state automata data structure that would be constructed using the algorithm of EXEC6A. (This may be compared to figure 10 which shows the equivalent alternate/successor linked list data structure.) In figure 24 each row corresponds to a state. The character associated with that row is the character which, if matched, would cause

the search algorithm to GOTO the state in SUCCESSOR of that row. For example, if the search algorithm is in state (row) five and it has an "H" then, it would GOTO to state six since SUCCESSOR(5) equals six. Then, while in state six, if the search algorithm did not have an "E" then, it would FAIL to state two where it would look for "E" again, which it does not have, and would finally FAIL to state zero. In this last example a weakness in EXEC6A is illustrated; the data structure does not eliminate possible redundant failure transitions. Authors Aho and Corasick, however, discuss in their article (Ref 1) how such a "deterministic" finite state automata may be generated. However, it was believed that this refinement would net minor benefits in search time, so its implementation is offered as a recommendation for further study.

The OUTPUT function of the search algorithm operated by signalling a match had occurred if a state were entered and the corresponding pattern number was not zero. Thus, if state seven were entered, then pattern number two would be signalled as matched. Also, CONCURRENT OUTPUTS (figure 24,too) identifies if another pattern is matched at the same time as another. For example, if the pattern "SHE" is matched so, also, is the pattern "HE". That is why the CONCURRENT OUTPUTS entry for pattern number two is equal to one. Thus, when pattern number two is signalled as found, so is pattern number one.



INDEX TABLE

	INDEX TABLE
H	1
S	4
T	11

	CHARACTER	SUCCESSOR	ALTERNATE	FAIL	PATTERN NUMBER
1	H	2	0	0	0
2	E	3	8	0	0
3	-1*	0	0	0	1
4	S	5	0	0	0
5	H	6	0	0	0
6	E	7	0	2	0
7	-1*	0	0	0	2
8	A	9	0	0	0
9	T	10	0	0	0
10	-1*	0	0	12	3
11	T	12	0	0	0
12	H	13	0	0	0
13	E	14	0	2	0
14	Y	15	0	0	2
15	-1*	0	0	0	4

	CONCURRENT OUTPUTS
1	0
2	1
3	0
4	0

*- The negative value indicates that this state has no character which may cause a GOTO transition.

Figure 24. Sample of Finite State Automata Data Structure of EXEC6A

So, with implementation of EXEC6A the twelve machine dependent pattern matching programs listed in figure 17 were complete and ready for evaluation testing.

But before proceeding to the next chapter, it should be mentioned that during the research for this thesis another algorithm was found (Ref 2). Its design revolves around a data structure that is built specifically for a single pattern. Hence, it could not easily accommodate the multi-pattern data structures of this thesis. However, as will be reiterated in the recommendations chapter, this algorithm should be investigated and compared to the results of this work.

III. Testing Procedures

In this chapter the criteria used to evaluate the performance of the pattern matching programs will be discussed, and how the statistics were gathered will be described.

First, one should recall the pattern matching problem. It is, given a set of patterns (in a data structure) to find all occurrences of these patterns in a given input subject string. For all practical purposes this input string may be considered text, but this "text" could well be source code and the pattern matcher a compiler, as described in Chapter I.

However, the efficiency of the pattern matching program can become very data dependent. That is, if the input text were Fortran source code, one might expect a higher incidence of patterns beginning with a particular character than say, might be found in regular English text. (As an example, many programmers tend to choose variable names that begin with the same letter, e.g., INAME, IADDR, ISSAN, etc.) Such a situation would lessen the value of an indexed data structure based on first characters as has been done in all the programs of this thesis. Another thought, how does the number of patterns affect the speed of the search from one approach to another? Also, does the expected frequency of occurrence (incidence) have any effect? With these ideas in mind, a test plan was developed.

It was decided that test data should include both English text and some programming language code--Fortran was chosen. And in order to minimize inherent peculiarities attributable to construction of English or Fortran text, it was also decided that a text of random combinations of characters should be used also. Therefore, three text files were chosen to be used during testing: English text, Fortran source code, and random text.

Next, suitable choices for patterns had to be made. It was decided that the following pattern combinations should be made:

1. Large number of patterns occurring often in the input text. (LH- large number, high incidence)

2. Small number of patterns occurring often in the input text. (SH- small number, high incidence)

3. Large number of patterns occurring rarely in the input text. (LL- large number, low incidence)

4. Small number of patterns occurring rarely in the input text. (SL- small number, low incidence)

Therefore, the three input texts were analyzed and frequency counts made on occurrences of individual words and particular combinations of characters. From the results of this work, the twelve pattern files to be used were constructed. During testing each of these pattern files was combined with the appropriate text file, English, Fortran, or random, to form twelve distinct test files. The original fifteen files (three text and twelve pattern) are described

in figure 25.

At this time it was necessary to identify the performance measures that would be needed in later evaluation. The single obvious measure was the time required for the search module to complete the pattern matching process. After all, this is what was to actually determine the "fastest" algorithm. However, it would be equally important to have some idea as to why one algorithm was faster than another. So, other measures were identified. These included the number of times the data structure was accessed during the search and the number of words returned. These measures would indicate the amount of work involved in communicating with the data structure. Also included were the total number of equality checks made and the number of successful ones which determined a pattern match. These measures would provide a relative efficiency for the various search algorithms.

Another proposed measure was the total number of comparisons made during the search which would give an idea of the general "overhead" processing. (This last measure was chosen because of the relatively large amount of CPU time required for a comparison operation.) Two other measures chosen were the amount of time spent in constructing the data structure and the size of the completed structure. These last two would be an especially important factor in a dynamic pattern structure as will be discussed later.

In all, nine different performance evaluation measures were chosen and the appropriate statistics gathering state-

TEXT FILES

File Name	
FORTRAN-	237 lines of Fortran source code totalling 22,201 characters. (Trailing blanks at end of line not counted.)
ENGLISH-	266 lines of English text totalling 15,015 characters. (Trailing blanks at end of each line not counted.)
RANDOM-	200 lines of random text totalling 15,762 characters, made up of 22 unique characters.

PATTERN FILES

File Name	Number of Patterns	Total Number of Characters In Patterns	Number of Unique First Characters In Patterns
LHFOR	50	297	20
SHFOR	10	34	10
LLFOR	50	361	19
SLFOR	10	72	6
LHENG	50	312	19
SHENG	10	26	7
LLENG	50	373	20
SLENG	10	101	9
LHRAN	50	182	21
SHRAN	10	14	8
LLRAN	50	199	21
SLRAN	10	31	9

LH- Large High incidence　　LL- Large Low incidence
SH- Small High incidence　　SL- Small Low incidence

Figure 25. Files Created for Evaluation Testing

ments were placed in each of the twelve programs. These nine measures are described in figure 26. Also listed in figure 26 are two calculable measures. One provides an average number of match comparisons made to find a pattern. The other gives an indication of the average amount of space required to store a pattern. Both these measures were expected to vary widely depending on type of input text and pattern choice.

The final test plan then, was to execute each of the twelve programs with each of the twelve different data files resulting in a total of 144 executions. In the next chapter the results of these runs will be discussed.

Search Time-	That amount of time spent searching input text for pattern matches.*
Construct Time-	That amount of time spent building the data structure.*
Structure Size-	Minimum number of words to contain data structure.
Words Returned-	Total number of words obtained form data structure during search.
Match Checks-	Total number of equality checks made while comparing patterns to text. (Can be character or word comparisons depending on program)
Check Successes-	Total number of successful equality checks which is by no means equal to number of patterns found.
Patterns Found-	Total number of patterns found during search.
Search Comparisons-	Total number of comparisons made during the search exclusive of those made implicitly within iterative DO LOOP's and intrinsic functions such as MINO.

Other Calculable Measures

Match Checks ÷ Patterns Found-	Average number of comparisons to find a pattern.
Structure Size ÷ Number of Patterns-	Average number of words required per pattern.

*- According to CDC these times are accurate to .01 seconds.

Figure 26. Performance Measures

IV. Results and Conclusions

Results

The numerical results of the testing are presented in Appendix B. The tables there show the actual figures returned from each execution of the twelve programs using the twelve data files. They are presented in the appendix since a relative evaluation is more important than actual comparison of hundreds of numbers. In fact, Tables I, II, III, IV, and V present just such rankings of the results. In these tables a "1" identifies the algorithm(s) which returned the "best" values for a given test, and a "12" identifies the algorithm which returned the worst results. Algorithms which returned identical values are given identical rankings.

Perhaps the most important measure of performance of a pattern matching algorithm is the amount of time it requires to search through the input text. Table I presents the rankings of the search times for the twelve programs and twelve input files (actual times in Appendix B). It is interesting to note that no one algorithm performed best for all input cases. For example, EXEC6A (f.s.a.) did "best" during the searches involving a large number of patterns (there were fifty patterns) in the data structure, but for a small number of patterns (ten) EXEC4B (alt/suc linked list) was best. Also interesting, it appears that the incidence of patterns had little influence on the comparative rankings.

TABLE I

Search Time Rankings

PROGRAM NAME	LARGE NUMBR OF PATTERNS WITH HIGH INCIDENCE (LH)			LARGE NUMBR OF PATTERNS WITH LOW INCIDENCE (LL)			SMALL NUMBR OF PATTERNS WITH HIGH INCIDENCE (SH)			SMALL NUMBR OF PATTERNS WITH LOW INCIDENCE (SL)		
	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN
EXEC1A	12	12	11	12	12	11	11	11	11	12	12	11
EXEC2A	10	10	9	10	10	9	9	9	9	10	10	9
EXEC3A	6	6	4	6	5	3	3	3	4	3	5	3
EXEC4A	3	2	2	4	3	2	5	2	1	8	3	2
EXEC6A	1	1	1	1	1	1	6	5	7	5	4	6
EXEC1B	11	11	12	11	11	12	12	12	12	11	11	12
EXEC2B	9	9	10	9	9	10	10	10	10	9	9	10
EXEC3B	7	7	6	7	7	5	7	7	6	6	7	7
EXEC4B	2	3	3	2	2	4	1	1	2	1	1	1
EXEC5B	4	4	5	3	4	6	2	4	3	2	2	4
EXEC3C	8	8	8	8	8	7	8	8	8	7	8	8
EXEC4C	5	5	7	5	6	8	4	6	5	4	6	5

However, no matter the number of patterns or incidence, several algorithms are clearly just plain slow. EXEC1A (simple list) and EXEC1B (packed simple list) were the "winners" in this category, chalking up times like seventy seconds compared to less than two seconds for the fastest algorithm given the same input. EXEC2A and EXEC2B did somewhat better using an indexed simple list data structure, turning in times one-third to one-half those of EXEC1A and EXEC1B.

Of the other eight algorithms/data structures, the times were all very close; the difference between number "1" and number "8" not quite a factor of 2. And in all cases, the difference between number "1" and number "2" was never greater than two to three, and often less.

Therefore, the job of choosing a single "best" algorithm based on search time alone is difficult to do. However, looking at Table I, the choice would probably be made between EXEC4A (alt/suc), EXEC6A (f.s.a.), and EXEC4B (packed alt/suc). Close together, but not in contention for first place, are EXEC3A (binary tree), EXEC5B (alt/suc, special search), and EXEC4C (alt/suc, packed input). And it appears that EXEC3B (packed binary tree) and EXEC3C (binary tree, packed input) are out of the running (along with EXEC1A, EXEC2A, EXEC1B, and EXEC2B).

Another measure which may influence the final choice is the size of the data structure. Table II presents the relative rankings for the various algorithms. An interesting comparison may be made between these and the rankings for

TABLE II

Size of Data Structure Rankings

PROGRAM NAME	LARGE NUMBR OF PATTERNS WITH HIGH INCIDENCE (LH)			LARGE NUMBR OF PATTERNS WITH LOW INCIDENCE (LL)			SMALL NUMBR OF PATTERNS WITH HIGH INCIDENCE (SH)			SMALL NUMBR OF PATTERNS WITH LOW INCIDENCE (SL)		
	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN
EXEC1A	8	8	8	8	8	8	2	2	2	5	8	2
EXEC2A	9	9	9	9	9	9	9	9	9	9	9	9
EXEC3A	10	10	10	10	10	10	10	10	11	11	11	11
EXEC4A	11	11	11	11	11	11	11	11	10	11	11	11
EXEC6A	12	12	12	12	12	12	12	12	12	12	12	12
EXEC1B	1	1	1	1	1	1	1	1	1	1	1	1
EXEC2B	2	2	2	2	2	2	3	3	3	2	2	3
EXEC3B	2	2	2	2	2	2	3	3	3	2	2	3
EXEC4B	5	5	5	5	5	5	3	6	3	6	5	6
EXEC5B	5	5	5	5	5	5	3	6	3	6	5	6
EXEC3C	2	2	2	2	2	2	3	3	3	2	2	3
EXEC4C	5	5	5	5	5	5	3	6	3	6	5	6

for search time. That is, the algorithm with the faster time has the larger data structure (as was predicted). For example, one of the slowest programs, EXEC1B (packed simple list) had the smallest data structure, while EXEC6A (f.s.a.), one of the fastest, had the largest data structure.

Figure 27 illustrates this relationship of search time to data structure size. In preparing this chart only figures for the top six programs were used, and the actual values shown are averages. That is, each time and size "bar" represents the average of six executions--low and high incidence runs for each Fortran, English, and random text file. Therefore, the relative relationships illustrated reflect typical expected values. In any event, it is clear that for EXEC3A, EXEC4A, and EXEC6A as the size of the structure increased the search time decreased. This, on the average, is true for all cases--the larger the structure, the faster the search.

However, notice some of the packed versions required less storage space and were still faster than unpacked programs; this is an important, almost contrary to anticipated, observation. For example, EXEC4B (packed alt/suc) and EXEC3A (binary tree) fit such a situation. EXEC4B was always faster than EXEC3A and always required less storage. But this speed difference is attributable to the difference in the design of the algorithms/data structures not the packed/unpacked approach. In fact, comparing again Table I and Table II, one can see that between programs of the same design (EXEC1A and EXEC1B, EXEC2A and EXEC2B, etc.) packing had little effect

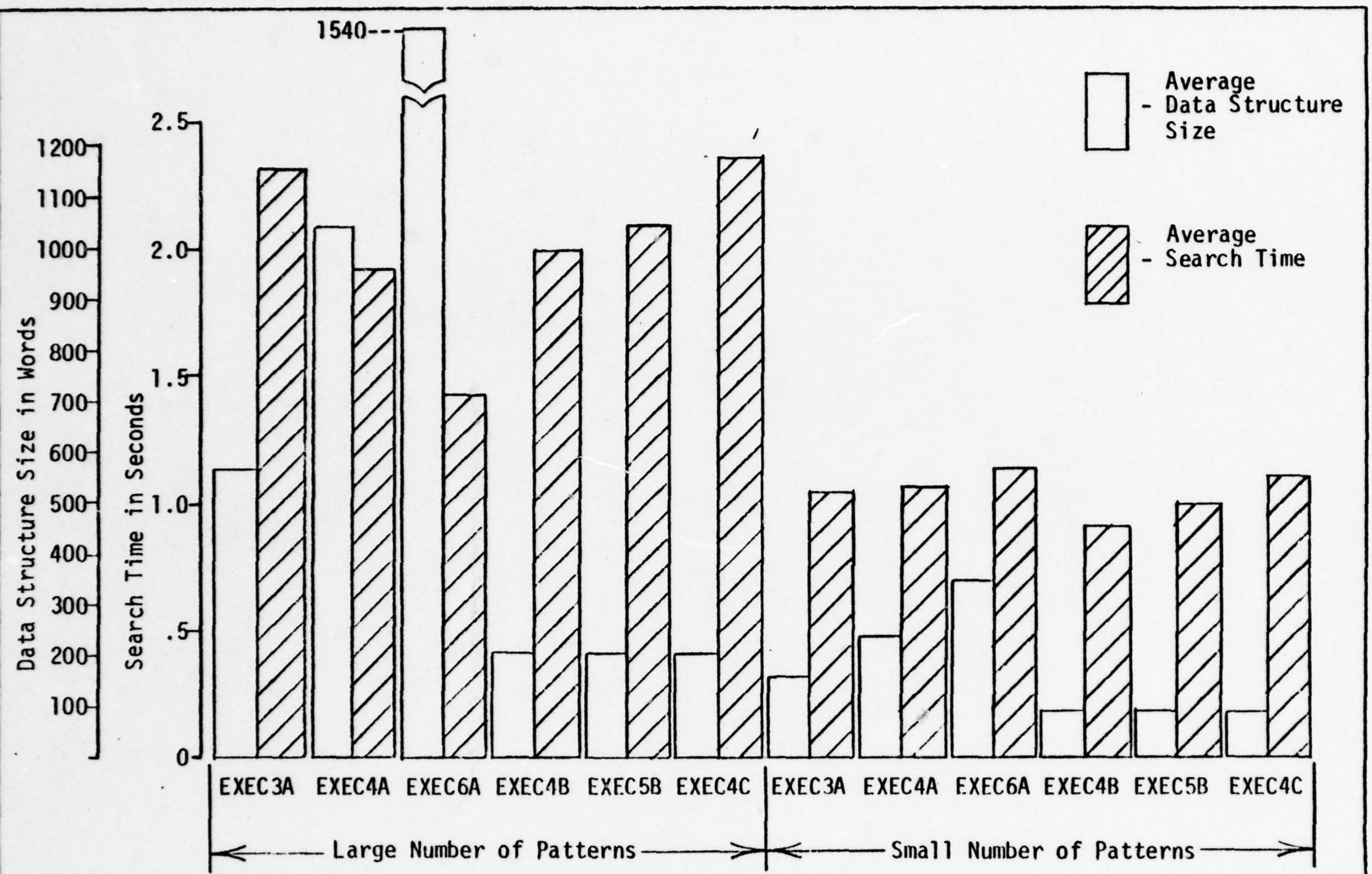


Figure 27. Bar Graph of Averaged Search Times and Averaged Data Structure Sizes

on the overall rankings for search time. (Figure 27 shows this limited difference between programs EXEC4A and EXEC4B.) However, it is important to note that packing did not always increase search time and, in some cases, even allowed a decrease in time.

The idea of design brings up another measure which may be used to evaluate overall efficiency of a given program. This is the measure reflecting the number of equality checks required to find patterns in the input text. The relative rankings of this measure are shown in Table III. (A "1", the best, required fewest checks.) One would expect these rankings to consistent with those of search time; that is, the algorithm making the fewer checks should be faster than another making more. However, with the exception of EXEC1A, EXEC2A, EXEC1B, and EXEC2B, this is not necessarily the situation.

For example, EXEC3B and EXEC3C, which pretty much held down positions seven and eight in search time, rank well in the number of match checks made. And of the original fast trio (EXEC4A, EXEC6A, and EXEC4B) only EXEC6A (f.s.a.) shows a consistently high ranking. Why then, this disparity between search time and the number of match checks? That is, why do the faster programs not necessarily make the fewer match checks? Table IV may give some idea of the answer to this question.

TABLE III

Rankings- Number of Match Checks
Made to Find Patterns In Text

PROGRAM NAME	LARGE NUMBR OF PATTERNS WITH HIGH INCIDENCE (LH)			LARGE NUMBR OF PATTERNS WITH LOW INCIDENCE (LL)			SMALL NUMBR OF PATTERNS WITH HIGH INCIDENCE (SH)			SMALL NUMBR OF PATTERNS WITH LOW INCIDENCE (SL)		
	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN
EXEC1A	11	11	11	11	11	11	11	11	11	11	11	11
EXEC2A	9	9	9	9	9	9	9	9	9	9	9	9
EXEC3A	5	4	1	6	4	1	3	2	1	6	3	1
EXEC4A	2	3	4	3	3	5	3	4	1	5	5	4
EXEC6A	1	1	3	1	1	4	1	1	1	4	2	3
EXEC1B	11	11	11	11	11	11	11	11	11	11	11	11
EXEC2B	9	9	9	9	9	9	9	9	9	9	9	9
EXEC3B	6	4	1	6	4	1	3	2	1	6	3	1
EXEC4B	8	8	8	8	8	8	8	8	8	8	7	8
EXEC5B	5	7	6	5	7	6	2	6	5	3	8	6
EXEC3C	2	2	5	2	2	3	3	5	7	2	1	5
EXEC4C	4	6	7	4	6	7	3	6	6	1	6	7

Table IV gives the rankings for the total number of comparisons made during the search. Recall that this measure was chosen to provide some indication of the overall amount of processing involved in the search process. And as can be seen in Table IV, EXEC3B (packed binary tree) does appear to have the highest amount of equality testing of the eight fastest programs. On the other hand, EXEC3C (binary tree, packed input) does not fair as badly, especially in the large pattern structures. This is a contraindicative result when compared to search time ranking for EXEC3C which places it a solid "8". The observation of this result caused this author to restudy the coding for EXEC3C. It was found that three comparisons were not being counted as they should. Therefore, because of the close agreement with search time rankings for the other programs, it will be assumed that had these comparisons been counted the correct relationships would be reflected. Another reason for this claim is that the processing for EXEC3C was actually more involved than EXEC4C and should compare similarly as EXEC3B compares to EXEC4B, a ratio of almost two to one.

One final measure which must be discussed before presenting the conclusions reflects the amount of time spent building the data structure. It is true that this time is trivial when compared to search time (numbers in Appendix B). However, if one is speaking of a dynamic data structure this can become an important consideration. The rankings of this measure are shown in Table V.

TABLE IV

Rankings- Number of Comparisons Made During Search

PROGRAM NAME	LARGE NUMBR OF PATTERNS WITH HIGH INCIDENCE (LH)			LARGE NUMBR OF PATTERNS WITH LOW INCIDENCE (LL)			SMALL NUMBR OF PATTERNS WITH HIGH INCIDENCE (SH)			SMALL NUMBR OF PATTERNS WITH LOW INCIDENCE (SL)		
	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN
EXEC1A	11	11	11	11	11	11	11	11	11	11	11	11
EXEC2A	9	9	9	9	9	9	9	9	9	9	9	9
EXEC3A	7	7	7	7	7	7	5	4	3	6	5	6
EXEC4A	1	1	1	3	2	2	4	2	1	3	3	3
EXEC6A	2	3	5	1	3	4	8	8	8	8	8	8
EXEC1B	12	12	12	12	12	12	12	12	12	12	12	12
EXEC2B	10	10	10	10	10	10	10	10	10	10	10	10
EXEC3B	8	8	8	8	8	8	7	6	5	7	7	7
EXEC4B	4	4	3	4	4	3	2	3	4	2	2	2
EXEC5B	5	6	6	6	6	6	1	1	2	1	1	1
EXEC3C	3	2	2	2	1	1	6	5	6	5	6	5
EXEC4C	6	5	4	5	5	5	3	7	7	4	4	4

From Table V one can see that there does not appear to be a clear-cut "winner". However, it appears there are some clear-cut "losers". For example, EXEC1A, EXEC1B, EXEC2A, EXEC2B, and EXEC6A seem to fill up the last five rankings; though admittedly, there are some exceptions for EXEC2A and EXEC2B. These exceptions are due to the small number of patterns in the data structure and the limited conflicts in index value. That is, for all programs (except EXEC6A) the time required to build the data structure is basically a combination of first, verifying that a pattern does not already exist in the structure, and then, second, adding the new pattern to the structure. Therefore, because EXEC2A and EXEC2B are so uncomplicated (using an indexed simple list), they were able to store a small number of patterns very quickly. (Note, this advantage disappears if several patterns share the same index value.)

Now, EXEC6A was just mentioned as a special case in its construction of the data structure. This is because after all patterns have been stored in the data structure, then the fail values for each state must be calculated. (Recall the finite state automata data structure discussed in Chapters I and II.) This "extra" time is approximately the time difference for EXEC4A and EXEC6A to construct the same data structure. On the average, EXEC6A required three times as much time; and sometimes as much as six or seven times.

Then, in light of a dynamic data structure, EXEC6A will not fair well in construction time when compared to the other

TABLE V

Rankings- Time Required To Construct Data Structure

PROGRAM NAME	LARGE NUMBR OF PATTERNS WITH HIGH INCIDENCE (LH)			LARGE NUMBR OF PATTERNS WITH LOW INCIDENCE (LL)			SMALL NUMBR OF PATTERNS WITH HIGH INCIDENCE (SH)			SMALL NUMBR OF PATTERNS WITH LOW INCIDENCE (SL)		
	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN	FOR	ENG	RAN
EXEC1A	12	12	12	12	12	12	10	7	8	11	12	10
EXEC2A	9	10	10	11	9	10	4	10	11	2	6	8
EXEC3A	1	5	2	3	6	7	7	1	2	5	4	4
EXEC4A	5	1	1	2	4	3	2	6	4	5	11	2
EXEC6A	10	9	8	9	11	9	12	12	12	12	10	12
EXEC1B	11	11	11	10	10	11	8	7	10	7	6	11
EXEC2B	8	8	9	8	8	8	11	2	2	1	2	4
EXEC3B	3	2	4	5	2	5	4	2	4	9	8	6
EXEC4B	6	3	6	3	1	1	8	4	8	2	1	6
EXEC5B	4	3	5	6	6	2	2	5	6	9	9	1
EXEC3C	2	6	2	1	2	4	1	11	6	7	3	8
EXEC4C	6	6	7	7	4	6	4	7	1	2	4	2

methods, excluding, of course, the slowest programs.

Conclusions

It is difficult, if not impossible, to pick a single overall "best" algorithm. But the choice does point to a group of three, EXEC4A, EXEC6A, and EXEC4B. Not surprisingly, all three are based on the alternate/successor linked list and hence, limit "back-up". EXEC4A uses a straight-forward unpacked approach; EXEC4B uses a packed version of the same, cutting space requirements by a factor up to four and more; and EXEC6A uses an unpacked and more complicated finite state automata version requiring the largest data structure and most time to construct.

From these three programs the final choice can be based on expected input and answers somewhat, the questions presented in Chapter III. That is, if the input is to involve a large static pattern structure, then EXEC6A is the best choice. However, if the pattern structure is small, or especially if it is dynamic, then the best choice is EXEC4A or EXEC4B, and the choice between them should be based on the space available for the data structure.

Equally important to identify are the worst programs. Clearly, the simple list and even the indexed simple list (EXEC1A, EXEC2A, EXEC1B, and EXEC2B) are unacceptable, regardless of their relative simplicity. The binary tree may have given better results if a better indexing method were used (EXEC3A, EXEC3B, and EXEC3C), but a subjective evaluation points to an indexed linked list (not implemented)

as a probable better choice.

A "double-sided" conclusion can be made regarding those programs designed to take advantage of the CYBER 74 large storage word. On one hand, it has been shown that shrinking the size of the data structure by packing does not adversely affect the speed of the search algorithm, and in some cases, the search can be faster. On the other hand, attempting to use packed input and a packed data structure to perform packed (up to ten character) comparisons did not fare so well.

There are several reasons why this packed comparison approach did not perform as was hoped. For one, the overhead in keeping track of search position in the input text and within each pattern overshadowed the benefits given by multi-character comparisons. However, these results can in part, be blamed on the inefficiency of using the high level language, Fortran. In the following recommendations chapter a proposal to further investigate multi-character comparisons will be presented.

V. Recommendations

It is in the spirit of resolving the many questions raised during this thesis that the following recommendations are made.

1. As mentioned earlier in Chapter I, one user of the pattern matching algorithm is the preprocessor. Mortran is one such example, and it is recommended that one of the "best" versions should be implemented within the Mortran processor. It should be a packed design since space is critical. The probable choice is EXEC4B (packed alternate/ successor data structure).

2. A possibility for improving the relative worth of an indexed simple list (EXEC2A and EXEC2B) might be to investigate better hashing (indexing) techniques rather than first character as was used.

3. Also in this idea of a better hash technique, perhaps a linked list data structure should be implemented, which, as noted before, this author feels would serve as well or possibly better than a binary tree, especially for a small number of patterns in the structure.

4. A fast pattern matching algorithm (Ref 2) was mentioned in Chapter I. It would be interesting to see how well the claimed performance of this algorithm would fair when using many patterns in the data structure. The design as presented in the reference is for a single scan through

the input text with a single pattern.

5. EXEC6A might be modified to achieve the "deterministic" finite state automata as described by authors Aho and Corasick (Ref 1). At the same time a packed version might be considered. This approach might make EXEC6A look better both from a search time viewpoint and from the amount of storage space required.

6. Other improvements to search time should be considered. For example, heuristic futility checks, such as "is there enough text left to match this pattern?" might be implemented. In this line of thought, an article by Malcolm Harrison (Ref 8) may be used to investigate an interesting process which uses "hashing signatures" to identify patterns and subject strings. In a way, this hashing technique applies a probabilistic futility check, i.e., "what is the chance that this pattern is contained in this line of text?"

7. Better advantage may be taken of the CYBER 74 hardware through use of Compass (CDC assembly) language coding. Perhaps this is easier said than done, but some ideas that come to mind are, one, use a single register to hold ten characters of input text during the search to eliminate repeated memory references, and, two, use a single register from which to unpack a single memory location rather than referencing the same repeatedly.

So, these above seven recommendations are offered as continuation to the work begun in this thesis. In particular,

Compass coding may offer unique data structures and algorithms for the CDC machine which will outperform some of the very sophisticated general purpose approaches. And from the beginning, the purpose of this thesis has been to provide a foundation upon which such future investigation may be built.

Bibliography

1. Aho, Alfred V. and Corasick, M.J. "Efficient String Matching: An Aid to Bibliographic Search," Communications of the ACM, 14: 777-779 (December 1971).

2. Boyer, R. S. and Moore, J. S. A Fast String Searching Algorithm, Technical Report. Menlo Park, CA: Stanford Research Institute, March 1976. (AD A022028)

3. Clark, Nathan B. Analysis and Modification of the Mortran 2.0 Preprocessor. MS Thesis. Wright-Patterson AFB, Ohio: AFIT, March 1977.

4. Constantine, L. L. and Yourdin, E. Structured Design. New York: Yourdin, Inc, 1975.

5. Control Data Corporation. Fortran Extended Version 4 Reference Manual. California: Control Data Corporation, 1976.

6. Gimpel, J. F. Algorithms in Snobol4. New York: Wiley and Sons, 1976.

7. Gimpel, J. F. "A theory of Discrete Patterns and Their Implementation in Snobol4," Communications of the ACM, 16: 91-100 (February 1973).

8. Harrison, Malcolm C. "Implementation of the Substring Test by Hashing," Communications of the ACM, 14: 777-779 (December 1971).

9. McIlroy, M. D. "Programming Techniques," Communications of the ACM, 10: 420-424 (July 1967).

10. Pfaltz, John L. Computer Data Structures. New York: McGraw-Hill Book Co, 1977.

Vita

John Barton Isett was born on 21 December, 1950 in Elmhurst, Illinois. From Illinois his family moved to Jackson, Mississippi in 1964 and there he graduated Callaway High School in 1969. He attended Mississippi State University and received a three year ROTC scholarship. He graduated in May 1973 with a Bachelor of Science degree in Mathematics and Computer Science and at the same time was commissioned in the USAF. He entered active duty in September 1973 at Keesler AFB, Mississippi. He worked there first, for two years, as a Computer Assisted Instruction (CAI) coursewriter, and then, for one year, as a classroom instructor for the WWMCCS Computer Programmer Branch. He entered the School of Engineering, Air Force Institute of Technology as a Computer Systems student in September, 1976.

APPENDIX A

CYBER 74 Character Set With Display Codes

Character	Display Code	Character	Display Code
:	00	5	40
A	01	6	41
B	02	7	42
C	03	8	43
D	04	9	44
E	05	+	45
F	06	-	46
G	07	*	47
H	10	/	50
I	11	(	51
J	12	)	52
K	13	$	53
L	14	=	54
M	15	ƀ (blank)	55
N	16	,	56
O	17	.	57
P	20	#	60
Q	21	[	61
R	22	]	62
S	23	%	63
%	24	"	64
U	25	_	65
V	26	!	66
W	27	&	67
X	30	'	70
Y	31	?	71
Z	32	<	72
0	33	>	73
1	34	@	74
2	35	\	75
3	36	^	76
4	37	;	77

APPENDIX B

Numerical Results of Test Program Executions Using Data Files Described in Figure 25

DATA FILE: LHFOR . (50 PATTERNS, Fortran TEXT, High INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	70.145	.0959	402	1,110,082	7,815,016	1,159,657	52,738	3131	3,504,435	370.3	8.0
EXEC2A	26.869	.0390	466	455,838	3,404,926	483,212	40,385	3131	1,076,054	154.3	9.3
EXEC3A	2.674	.0100	566	35,696	296,048	40,317	17,612	3131	253,249	12.8	11.3
EXEC4A	2.117	.0159	980	51,609	206,436	29,408	9,610	3131	157,318	9.4	19.6
EXEC6A	1.643	.0430	1487	31,323	9,122	22,342	8,228	3131	158,401	7.1	29.7
EXEC1B	65.402	.0600	105	1,110,082	2,331,173	1,159,657	52,743	3131	3,553,984	370.3	2.1
EXEC2B	26.138	.0309	169	455,838	925,777	483,212	40,385	3131	1,137,961	154.3	3.4
EXEC3B	3.008	.0120	169	35,696	64,100	40,317	17,612	3131	285,560	12.8	3.4
EXEC4B	2.099	.0160	202	40,889	81,778	42,045	21,947	3131	164,894	13.4	4.0
EXEC5B	2.202	.0139	202	40,885	81,770	12,255 Words 24,344 Chars	8,048 Words 8,418 Chars	3131	166,390	11.7	4.0
EXEC3C	3.145	.0139	169	35,682	64,082	29,589 Words	3,754 Words	3131	159,141	9.4	3.4
EXEC4C	2.424	.0109	202	41,715	83,430	32,939 Words	11,977 Words	3131	168,203	10.5	4.0

DATA FILE: LHENG . (50 PATTERNS, English TEXT, High INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	48.023	.0809	414	750,786	5,465,778	779,436	30,727	2043	2,355,480	381.4	8.3
EXEC2A	19.234	.0430	478	315,000	2,592,503	328,628	22,360	2043	726,512	160.8	9.6
EXEC3A	1.869	.0149	587	24,684	217,134	22,267	6,291	2043	158,484	10.9	11.6
EXEC4A	1.499	.0110	1152	35,804	143,216	20,789	5,134	2043	105,964	10.2	23.6
EXEC6A	1.013	.0420	1697	21,418	6,398	15,775	4,410	2043	109,526	7.7	33.9
EXEC1B	44.809	.0530	108	750,786	1,621,702	779,436	30,729	2043	2,384,120	381.4	2.2
EXEC2B	17.552	.0360	172	315,000	661.460	328,628	22,360	2043	763,521	160.8	3.4
EXEC3B	1.911	.0129	172	24,684	46,442	22,267	6,291	2043	174,356	10.9	3.4
EXEC4B	1.542	.0130	210	29,976	59,952	29,451	13,497	2043	112,789	14.4	4.2
EXEC5B	1,619	.0130	210	29,976	59,952	8,255 Words 18,158 Chars	5,945 Words 4,474 Chars	2043	120,040	12.9	4.2
EXEC3C	2.179	.0150	172	24,668	46,401	20,279 Words	2,258 Words	2043	108,158	9.9	3.4
EXEC4C	1.787	.0150	210	30,733	61,466	24,703 Words	7,951 Words	2043	119,789	12.0	4.2

DATA FILE: LHRAN . (50 PATTERNS, Random TEXT, High INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	44.069	.0679	277	788,135	3,578,152	821,634	40,229	6,694	2,487,475	122.6	5.5
EXEC2A	24.548	.0400	341	445,815	2,463,774	463,552	27,555	6,694	1,011,647	69.2	6.8
EXEC3A	2.276	.0120	441	32,955	238,245	24,903	1,744	6,694	206,247	3.7	8.8
EXEC4A	1.918	.0109	652	47,217	188,868	31,455	1,720	6,694	135,818	4.7	13.0
EXEC6A	1.625	.0280	1083	28,345	12,583	31,212	1,711	6,694	175,637	4.6	21.7
EXEC1B	45.662	.0610	100	788,135	1,576,270	821,634	40,229	6,694	2,520,970	122.6	2.0
EXEC2B	24.950	.0340	164	445,815	888,542	463,552	27,555	6,694	1,057,816	69.2	3.3
EXEC3B	2.606	.0129	164	32,955	62,822	24,903	1,744	6,694	220,838	3.7	3.3
EXEC4B	2.173	.0140	176	43,157	86,314	44,146	14,390	6,694	150,894	6.6	3.5
EXEC5B	2.298	.0130	176	43,154	86,308	12,555 Words 28,972 Chars	10,060 Words 1,679 Chars	6,694	177,406	6.2	3.5
EXEC3C	3.082	.0120	164	32,943	62,798	32,315 Words	2,462 Words	6	146,702	4.8	3.3
EXEC4C	2.652	.0170	176	44,630	89,260	41,934 Words	10,668 Words	6,694	169,717	6.3	3.5

DATA FILE: LLFOR . (50 PATTERNS, Fortran TEXT, Low INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	72.861	.0849	461	1,110,088	9,124,925	1,162,417	52,408	36	3,509,634	32289	9.2
EXEC2A	26.623	.0479	525	417,949	3,552,320	448,077	41,085	36	1,004,428	12446	10.5
EXEC3A	2.840	.0150	625	36,139	327,374	42,649	17,444	36	261,972	1184	12.5
EXEC4A	2.246	.0130	1204	53,593	214,372	31,392	9,797	36	160,807	871	24.0
EXEC6A	1.637	.0449	1741	31,114	8,913	25,325	8,622	36	157,689	703	34.8
EXEC1B	64.907	.0629	107	1,110,088	2,375,590	1,162,417	52,408	36	3,561,930	32289	2.1
EXEC2B	24.109	.0310	171	417,949	857,306	448,077	41,085	36	1,068,047	12446	3.4
EXEC3B	3.132	.0179	171	36,139	65,371	42,649	17,444	36	290,801	1184	3.4
EXEC4B	2.144	.0150	216	41,475	82,950	43,057	21,093	36	162,313	1196	4.3
EXEC5B	2.219	.0199	216	41,473	82,946	11,228 Words 26,339 Chars	7,010 Words 8,562 Chars	36	167,235	1043	4.3
EXEC3C	3.242	.0129	771	36,122	65,333	29,285 Words	4,044 Words	36	158,517	813	3.4
EXEC4C	2.466	.0200	216	43,020	86,040	33,384 Words	9,831 Words	36	164,090	927	4.3

DATA FILE: LLENG . (50 PATTERNS, English TEXT, Low INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	51.199	.0859	483	750,786	6,501,863	781,222	30,576	104	2,358,899	7511	9.7
EXEC2A	20.858	.0519	547	314,993	2,999,207	330,414	22,207	104	729,917	3177	10.9
EXEC3A	1.886	.0140	647	24,370	237,127	23,740	6,138	104	163,584	228	12.9
EXEC4A	1.619	.0130	1424	37,323	149,292	22,308	4,944	104	109,607	214	28.5
EXEC6A	1.119	.0610	2037	21,577	6,562	17,375	4,189	104	110,547	167	40.7
EXEC1B	44.105	.0599	112	750,786	1,681,767	781,222	30,576	104	2,389,325	7511	2.2
EXEC2B	17.805	.0349	176	314,993	691,577	330,414	22,207	104	768,712	3177	3.5
EXEC3B	1.992	.0119	176	24,370	46,908	23,740	6,138	104	178,229	228	3.5
EXEC4B	1.585	.0100	220	30,743	61,577	31,007	13,262	104	113,617	298	4.4
EXEC5B	1.635	.0140	220	30,737	61,474	8,296 Words 19,517 Chars	5,814 Words 4,230 Chars	104	122,954	267	4.4
EXEC3C	2.308	.0119	176	24,353	46,869	20,090 Words	2,384 Words	104	107,717	193	3.5
EXEC4C	1.989	.0130	220	32,255	64,510	26,291 Words	6,999 Words	104	120,572	252	4.4

DATA FILE: LLRAN. (50 PATTERNS, Random TEXT, Low INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	45.612	.0769	299	788,140	3,924,940	828,841	40,844	102	2,502,516	8125	6.0
EXEC2A	24.224	.0389	363	445,820	2,633,380	470,759	28,171	102	1,026,688	4615	7.3
EXEC3A	2.363	.0170	463	30,979	239,060	30,134	2,360	102	220,613	295	9.3
EXEC4A	2.172	.0120	740	53,265	213,060	37,503	2,336	102	149,195	367	14.8
EXEC6A	1.607	.0350	1199	28,339	12,577	36,969	2,273	102	172,680	362	24.0
EXEC1B	46.381	.0579	100	788,140	1,576,280	828,841	40,845	102	2,543,212	8125	2.0
EXEC2B	24.994	.0329	164	445,820	888,552	470,759	28,171	102	1,080,058	4615	3.3
EXEC3B	2.595	.0149	164	30,979	58,870	30,134	2,360	102	235,778	295	3.3
EXEC4B	2.485	.0109	192	48,684	97,368	50,329	15,005	102	163,240	493	3.8
EXEC5B	2.681	.0110	192	48,680	97,360	12,555 Words 35,138 Chars	10,039 Words 2,294 Chars	102	200,731	467	3.8
EXEC3C	2.909	.0139	164	30,966	58,844	30,863 Words	2,987 Words	102	142,871	302	3.3
EXEC4C	2.950	.0150	192	51,132	102,264	48,509 Words	10,694 Words	102	182,765	475	3.8

DATA FILE: SHFOR . (10 PATTERNS, Fortran TEXT, High INCIDENCE)

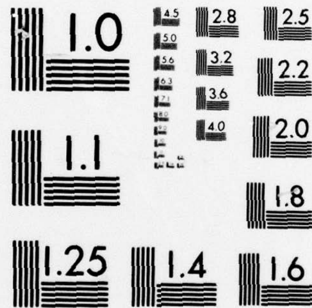
PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	12,534	.0040	54	222,014	976,867	229,657	9,442	1795	713,027	127.0	5.4
EXEC2A	3.019	.0019	118	54,831	270,118	40,273	2,566	1795	141,327	22.0	11.8
EXEC3A	1.224	.0029	138	22,205	114,516	7,647	2,566	1795	86,229	4.3	13.8
EXEC4A	1.276	.0009	200	22,848	119,392	7,647	2,566	1795	83,722	4.3	20.0
EXEC6A	1.405	.0070	284	28,628	6,427	7,198	2,490	1795	123,551	4.0	28.4
EXEC1B	13.046	.0039	20	222,014	444,028	229,657	9,443	1795	720,670	127.0	2.0
EXEC2B	3.137	.0050	84	54,831	94,338	40,273	2,566	1795	155,847	22.0	8.4
EXEC3B	1.411	.0019	84	22,205	29,086	7,647	2,566	1795	96,775	4.3	8.4
EXEC4B	1.052	.0039	84	22,205	44,410	14,520	9,443	1795	79,661	8.0	8.4
EXEC5B	1.174	.0009	84	22,205	22,205	6,828 Words 100 Chars	1,781 Words 66 Chars	1795	55,267	3.8	8.4
EXEC3C	1.606	.0000	84	22,202	29,080	7,637 Words	761 Words	1795	90,286	4.3	8.4
EXEC4C	1.254	.0019	84	22,201	44,402	7,638 Words	2,556 Words	1795	81,157	4.3	8.4

AD-A052 916 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/4
DISCRETE-PATTERN MATCHING ALGORITHMS AND DATA STRUCTURES FOR CY--ETC(U)
MAR 78 J B ISETT

UNCLASSIFIED AFIT/GCS/MA/78M-2 NL

2 OF 2
AD
A052916

END
DATE
FILMED
5-78
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DATA FILE: SHENG . (10 PATTERNS, English TEXT, High INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI- SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	8.2220	.0029	46	150,155	540,556	155,662	6,948	1436	483,483	108.0	4.6
EXEC2A	1.8940	.0039	110	36,840	145,609	27,332	3,334	1436	95,302	19.0	11.0
EXEC3A	.8599	.0000	130	16,508	78,846	4,990	1,323	1436	60,389	3.5	13.0
EXEC4A	.8309	.0020	156	20,319	81,276	5,304	1,323	1436	53,462	3.7	15.6
EXEC6A	.8980	.0070	229	18,324	3,309	4,846	1,323	1436	81,824	3.4	22.9
EXEC1B	8.6360	.0029	20	150,155	300,310	155,662	6,948	1436	488,990	108.0	2.0
EXEC2B	2.0210	.0009	84	36,840	62,279	27,332	3,334	1436	104,423	19.0	8.4
EXEC3B	1.0820	.0009	84	16,508	21,615	4,990	1,323	1436	66,036	3.5	8.4
EXEC4B	.8200	.0010	88	18,163	36,326	8,972	4,937	1436	56,332	6.2	8.8
EXEC5B	.8690	.0019	88	18,163	36,326	3,579 Words 3,388 Chars	2,309 Words 618 Chars	1436	51,980	4.8	8.8
EXEC3C	1.1780	.0040	84	16,504	21,607	5,567 Words	464 Words	1436	62,214	3.9	8.4
EXEC4C	.9850	.0029	88	18,160	36,320	6,967 Words	2,927 Words	1436	67,365	4.8	8.8

DATA FILE: SHRAN. (10 PATTERNS, Random TEXT, High INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	7.858	.0030	34	157,624	378,300	160,887	8,579	5312	503,779	30.20	3.4
EXEC2A	2.599	.0049	98	51,718	155,418	39,219	1,634	5312	125,530	7.40	9.8
EXEC3A	.996	.0019	118	17,294	80,924	3,267	106	5312	62,992	.61	11.8
EXEC4A	.785	.0020	112	19,029	76,116	3,267	106	5312	55,365	.61	11.2
EXEC6A	1.165	.0060	168	22,707	6,945	3,267	106	5312	99,439	.61	16.8
EXEC1B	9.302	.0040	20	157,624	315,248	160,887	8,579	5312	507,042	30.20	2.0
EXEC2B	2.898	.0019	84	51,718	94,619	39,219	4,634	5312	135,738	7.40	8.4
EXEC3B	1.101	.0020	84	17,294	25,771	3,267	106	5312	70,149	.61	8.4
EXEC4B	.838	.0030	84	17,293	34,586	10,208	7,051	5312	64,192	1.90	8.4
EXEC5B	.985	.0029	84	17,293	34,586	6,789 Words 1,613 Chars	5,211 Words 120 Chars	5312	55,442	1.60	8.4
EXEC3C	1.390	.0029	84	17,290	25,763	8,803 Words	330 Words	5312	74,029	1.65	8.4
EXEC4C	1.063	.0009	84	17,290	34,580	8,657 Words	5,496 Words	5312	76,783	1.63	8.4

DATA FILE: SLFOR . (10 PATTERNS, Fortran TEXT, Low INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	14.845	.0049	93	222,018	1,842,747	231,772	9,816	52	717,641	4457	9.3
EXEC2A	2.475	.0019	157	44,116	254,741	31,669	6,075	52	121,367	609	15.7
EXEC3A	1.210	.0020	177	23,773	135,134	9,003	3,749	52	88,897	173	17.7
EXEC4A	1.732	.0020	324	29,600	118,400	7,399	2,888	52	76,211	142	32.4
EXEC6A	1.279	.0139	440	25,742	3,541	7,058	2,786	52	110,907	135	44.0
EXEC1B	13.116	.0029	21	222,018	466,238	231,772	9,815	52	727,386	4457	2.1
EXEC2B	2.454	.0010	85	44,116	70,326	31,669	6,075	52	434,853	609	8.5
EXEC3B	1.485	.0030	85	23,773	29,640	9,003	3,749	52	96,439	173	8.5
EXEC4B	1.045	.0019	94	23,737	47,474	10,894	6,620	52	71,607	209	9.4
EXEC5B	1.112	.0030	94	23,737	47,474	3,707 Words 2,775 Chars	773 Words 1,408 Chars	52	59,546	124	9.4
EXEC3C	1.624	.0029	85	23,767	29,627	6,140 Words	834 Words	52	85,863	118	8.5
EXEC4C	1.278	.0019	94	23,833	47,666	5,902 Words	1,519 Words	52	79,752	113	9.4

DATA FILE: SLENG . (10 PATTERNS, English TEXT, Low INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	10.996	.0070	120	150,159	1,651,762	155,990	5,884	44	483,086	3545	12.0
EXEC2A	2.173	.0029	184	35,352	292,415	26,168	1,806	44	92,385	594	18.4
EXEC3A	.902	.0020	204	15,500	109,455	5,628	1,116	44	61,041	128	20.4
EXEC4A	.819	.0049	460	20,710	82,840	5,695	1,116	44	55,895	129	46.0
EXEC6A	.898	.0140	609	18,890	3,875	5,432	1,073	44	81,596	123	60.9
EXEC1B	8.976	.0029	25	150,159	375,399	155,990	5,884	44	488,914	3545	2.5
EXEC2B	2.056	.0030	89	35,352	68,251	26,168	1,806	44	102,291	594	8.9
EXEC3B	.967	.0009	89	15,500	22,185	5,628	1,116	44	66,284	128	8.9
EXEC4B	.747	.0009	96	16,366	32,732	9,852	5,191	44	52,919	224	9.6
EXEC5B	.819	.0039	96	16,366	32,730	4,046 Words 1,665 Chars	721 Words 321 Chars	44	41,792	127	9.6
EXEC3C	1.132	.0019	89	15,494	22,169	5,144 Words	588 Words	44	61,074	116	8.9
EXEC4C	.947	.0020	96	16,405	32,810	5,932 Words	1,218 Words	44	56,934	135	9.6

DATA FILE: SLRAN . (10 PATTERNS, Random TEXT, Low INCIDENCE)

PROGRAM NAME	SEARCH TIME	TIME TO BUILD DATA STRUCTR	DATA STRUCTR SIZE	NUMBER OF ACCESSES TO DATA STRUCTURE	NUMBER OF WORDS RETURNED	NUMBER OF MATCH CHECKS	NUMBER OF CHECK SUCCESSES	NUMBER PATRNS FOUND	NUMBER OF SEARCH COMPARI-SONS	AVERAGE CHECKS PER PATTERN	AVERAGE WORDS PER PATTERN
EXEC1A	8.826	.0030	51	157,633	646,297	165,973	8,370	16	513,767	10,373	5.1
EXEC2A	2.569	.0020	115	47,673	213,642	40,251	1,854	16	126,981	2,515	11.5
EXEC3A	.974	.0010	135	16,952	92,413	8,049	370	16	79,939	503	13.5
EXEC4A	.974	.0009	184	24,070	96,280	8,308	370	16	68,474	519	18.4
EXEC6A	1.095	.0070	264	22,266	6,504	8,295	370	16	98,844	518	26.4
EXEC1B	9.170	.0049	20	157,633	315,266	165,973	8,370	16	522,105	10,373	2.0
EXEC2B	2.657	.0010	84	47,673	86,100	40,251	1,184	16	141,435	2,515	8.4
EXEC3B	1.140	.0019	84	16,952	24,658	8,049	370	16	86,845	503	8.4
EXEC4B	.935	.0019	86	18,704	37,408	14,821	6,884	16	66,025	926	8.6
EXEC5B	.983	.0000	86	18,704	37,408	6,448 Words 3,137 Chars	1,480 Words 155 Chars	16	53,339	599	8.6
EXEC3C	1.382	.0020	84	16,943	24,640	8,476 Words	781 Words	16	73,504	530	8.4
EXEC4C	1.052	.0009	86	18,724	37,448	9,981 Words	2,003 Words	16	68,699	624	8.6

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Entry
1. REPORT NUMBER	AFIT/GCS/MA/78M-2
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	DISCRETE-PATTERN MATCHING ALGORITHMS AND DATA STRUCTURES FOR CYBER 74
5. TYPE OF REPORT & PERIOD COVERED	MS Thesis
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	John B. Isett Captain USAF
8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Aeronautical Systems Division Wright-Patterson AFB, Ohio 45433
12. REPORT DATE	March, 1978
13. NUMBER OF PAGES	102
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	
15. SECURITY CLASS. (of this report)	Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report)	Approved for public release; distribution unlimited
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	
18. SUPPLEMENTARY NOTES	Approved for public release; IAW AFR 190-17 Jerral F. Guess, Captain, USAF Director of Information
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A description of the discrete-pattern matching process is presented with the key elements described. Six data structure approaches and related search algorithms are presented. Twelve programs were coded to implement five out of the six structures/algorithms using packed and unpacked approaches on a CYBER 74. Runs were made with twelve different data files using Fortran, English, and random text. The effect of the number of patterns in the data structure and the expected incidence in the text were included. The "best" data structures/algorithms were a finite state automata and an alternate/successor linked list.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)