

Final Report

**STANDARDIZED PROTOCOLS BETWEEN
MICROCOMPUTERS AND DATA SERVICE NETWORKS**

By

Dr. Jay H. Schlag

Mr. Henry Owen

Mrs. Katharine L. Schlag

February, 1988

GEORGIA INSTITUTE OF TECHNOLOGY

A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA

SCHOOL OF ELECTRICAL ENGINEERING

ATLANTA, GEORGIA 30332



**STANDARDIZED PROTOCOLS BETWEEN
MICROCOMPUTERS AND DATA SERVICE NETWORKS**

by

**Dr. Jay H. Schlag
Mr. Henry Owen
Mrs. Katharine L. Schlag**

1 February 1988

TABLE OF CONTENTS

SECTION	PAGE
1. INTRODUCTION	1
2. BACKGROUND	2
3. ALGORITHM CONSIDERATIONS	3
Instruction Format Decoding	3
Header Identification	16
Code Segment Identification	16
Data Segment Identification	17
Example of Segment Identification	18
File-Comparison Approaches	22
4. ALGORITHM	24
Overview	24
Host Update Algorithm	26
Update File Format	32
Offset Packet Format	34
New Packet Format	34
Remote Update Algorithm	35
5. SOFTWARE DEVELOPMENT	37

TABLE OF CONTENTS (Continued)

SECTION	PAGE
6. FUTURE DIRECTIONS	49
Header Algorithm	49
Data Segment Algorithm	49
File-compare Algorithm Efficiency	49
Prototype Code Demonstration	50
Analysis of Packet Size Versus Overhead Bytes	50
Examination of Methods for Error Correction and Encryption	51
APPENDIX A	52
APPENDIX B	67

LIST OF FIGURES

FIGURE	PAGE
1. Instruction Format Types	7
2. Algorithm Flow Chart	8
3. 256 Possible TYPE Assignments Based on First Byte	9
4. Header Fields and Their Explanations	10
5. Example of " .map" File	19
6. Example of a Header File	21
7. Programs Used by the Update Algorithms	25
8. Host Update Algorithms	27
9. Update File Format	33
10. Remote Update Algorithms	34
11. CDC Libraries	38
12. Flow Chart of RDBIN.PAS Program	39
13. Flow Chart of LEXEC.PAS Program	40
14. Flow Chart of COMPEX.PAS Program	41
15. Flow Chart of BPACK.PAS Program	42
16. List of Assembly Language Subroutines of COM1 Port Access	43
17. List of Fortran Subroutines of COM1 Port Access	44

Standardized Protocols Between Microcomputers and Data Service Networks

SECTION 1

INTRODUCTION

The rapid growth of computer applications coupled with the development of inexpensive, small computer systems has drastically changed the role of the mainframe computer service network, primarily because computer intelligence has shifted from the main host to the user-based systems. This shift has precipitated a need for changes in many other computer functions such as data base management, applications programs, data security and communication protocols. Control Data Corporation initiated a two-year effort with Georgia Tech to address issues within the single area of computer communication protocols in the computer service network and the impact of modern computer networks on the protocol requirements. Of particular interest are the areas of communication hardware, data transmission structure, error detection, error correction, encryption, data base security, data format exchange, and data base management command exchange. The main thrust of this effort has been to develop of an efficient algorithm to keep the executable files on a remote computer updated by the host system and to investigate the related issues.

This report is the final report on that two-year effort. An annual report was submitted covering the work done in the first year. Included in that report are the results of a literature search with results of recent work in the above referenced areas of interest, both for the assessment of the current status and for future reference; a description of the system elements involved in updating an executable file on the remote from the host system; a discussion of the proposed new update scheme including an analysis of the information fields in an instruction and methods of transferring data; the results of trial data in each of several compiler-level languages processed with the new update scheme and the efficiency that would result from using the new scheme; and a discussion of the problem features of the DOS COM1 communication port. Those results will not be repeated here, but will be summarized where necessary for clarity and continuity. The reader is referred to the annual report for details.

SECTION 2

BACKGROUND

During the 1970 time frame the user in a service computer network was a simple CRT terminal or hard-copy device with no local processing or data storage capability. During the early 1980 period the service networks developed a large variety of computer communication networks to effectively handle larger volumes of more sophisticated data transmissions. These networks effectively removed the communication demands from the central computer but did not remove the host application programs as the center of processing activity. With the development of inexpensive personal and business computers, the typical user will require local application programs and mass storage, with the computer service network supplying access to large data bases for portions of the processing information. A major problem in a system of the 1990's will be to keep the executable files on a remote computer updated by the host system in a timely and cost effective manner. Currently when an application program is changed on the host system, the new executable file is transmitted in toto to the remote user to replace the old executable file. This process is both time consuming and costly. The new update scheme compares at the host system both the old and new executable files. Any differences in the files, including different offsets from change in position, additional data, and information from deleted data, are incorporated into a file, hopefully much smaller than the new executable file, that is transmitted to the remote user. Information in this file is then used to reconstruct the old executable file at the remote into a new executable file that matches the one at the host system. If the changes have been minor, there is the potential to save a great deal of time and money.

To design an algorithm to realize this kind of update scheme, a number of considerations must be taken into account. During the first year of this effort Georgia Tech examined several issues and made some preliminary estimates of the efficiencies that might be realized. The results were promising. More specifically, Georgia Tech examined four types of information fields that comprise machine-level instructions for the

impact that changes in a program would have on them, as well as four ways to transfer data packets to properly account for that impact. During the second year project personnel used new insights gained in the areas of information fields and data transfer to examine the instruction formats, header segments and data segments, and the file comparison algorithms that might be used or modified to detect the differences between the old and new executable files. Results from these examinations will be presented in Section 3 below. Section 4 is devoted to a description of the update algorithm that was designed. Section 5 is a discussion of future directions that are recommended for the refinement and expansion of the update approach developed during this effort.

SECTION 3

ALGORITHM CONSIDERATIONS

To develop an algorithm that will update an executable file using as much of the old file as possible as the building blocks for the new file, Georgia Tech personnel carefully examined a number of issues believed to be critical to the initial development of the update algorithm investigated. The results, presented below, were incorporated into the algorithm outlined in Section 4. The detailed study of some of the remaining issues are outside the scope of this effort, but are identified and discussed briefly in this report as future directions for this program.

The algorithm must identify the instructions that are common between various versions of the program. In identifying these instructions, the algorithm must recognize identical instructions which have different offsets in their data, displacement, or address fields. In order to search an executable file for instructions, the header and data segments must be removed from consideration, since these segments do not contain instructions. The header and data segments are handled separately, with their transmission algorithms being different from the code segment.

INSTRUCTION FORMAT DECODING

The INTEL 8086/8088 instruction set is made up of many different instruction formats. The "iAPX 86/88, 186/188 Users' Manual", 1985, lists the formats for each instruction type in Table 1-22. The instructions are made up of subfields which are summarized in Table 1-21. When an instruction is relocated in a program, subfields within the instructions change to reflect the new instruction location. By examining the subfields in each instruction as shown in Table 1-22, the bytes in an instruction format that may change upon relocation may be identified. The above referenced tables are include as Appendix A of this report. A summary of those fields which may change upon instruction relocation

includes the following bytes in the instructions:

DISP-LO	Lower-order byte of an unsigned displacement
DISP-HI	Higher-order byte of an unsigned displacement
IP-LO	Lower-order byte of a new instruction pointer value
IP-HI	Higher-order byte of a new instruction pointer value
CS-LO	Lower-order byte of a new code segment
CS-HI	Higher-order byte of a new code segment
IP-INC8	8-bit signed increment to instruction pointer
IP-INC-LO	Lower-order byte of signed 16-bit instruction pointer increment
IP-INC-HI	Higher-order byte of signed 16-bit instruction pointer increment
ADDR-LO	Lower-order byte of direct address (offset) of memory operand
ADDR-HI	Higher-order byte of direct address (offset) of memory operand
DATA-LO	Lower-order byte of data
DATA-HI	Higher-order byte of data

For a given instruction, it is necessary to identify the fields that may change upon instruction relocation. By examining Table 1-22, sixteen different cases may be identified. These sixteen different cases are

shown in Figure 1. In this figure, only those bytes which are underlined may change upon relocation

In order to determine the format for a given instruction, several bit fields within a given instruction must be examined. The first field is the most significant eight bits (the first byte) of the instruction. For all instructions which do not contain displacements, the format is now determined. For instructions which may have displacements, the number of displacement bytes must be determined. These are determined from the MOD field bits which are the two most significant bits in the second byte of an instruction. For MOD=11 there are no displacement bytes, for MOD=10 there are two displacement bytes, and for MOD=01 there is one displacement byte. If MOD=00 and also R\M=110 then two displacement bytes follow. The R\M field is located in the three least significant bits in the second byte of an instruction.

There are two special cases in which the OPCODE must be used in conjunction with the register field to determine the instruction format. The register field consists of third, fourth, and fifth bits of the second instruction byte as counted from MSB to LSB. For OPCODE Hex F6 and REG=000, one data byte will be contained at the end of the instruction. For OPCODE Hex F7 and REG=000, two data bytes will be contained at the end of the instruction.

The flow chart for determining the instruction format is shown in Figure 2. The result of this format algorithm is an assigned instruction format type for every possible instruction. It should be noted that the first eight bits of all instructions are not always sufficient to determine the instruction type. Every instruction is assigned an initial instruction type based on the first eight bits and a table look-up scheme; then by examining additional bits in those cases where it is necessary, the correct instruction format assignment is always obtained. The initial instruction format assignment is made from Figure 3. Figure 3 represents all of the 256 possible initial assignments based upon the first eight bits of an instruction. The table is read by referencing the position in the table and equating this position to the corresponding first eight entries of

TYPE	INSTRUCTION FORMAT					
1	BYTE 1	BYTE 2				
2	BYTE 1	BYTE 2	<u>DISP-LO</u>			
3	BYTE 1	BYTE 2	<u>DISP-LO</u>	<u>DISP-HI</u>		
4	BYTE 1	BYTE 2	<u>DATA-LO</u>	<u>DATA-HI</u>		
5	BYTE 1	BYTE 2	<u>DISP-LO</u>	<u>DATA-LO</u>	<u>DATA-HI</u>	
6	BYTE 1	BYTE 2	<u>DISP-LO</u>	<u>DISP-HI</u>	<u>DATA-LO</u>	<u>DATA-HI</u>
7	BYTE 1	BYTE 2	BYTE 3			
8	BYTE 1	BYTE 2	<u>DISP-LO</u>	BYTE 4		
9	BYTE 1	BYTE 2	<u>DISP-LO</u>	<u>DISP-HI</u>	BYTE 5	
10	BYTE 1	<u>DATA-LO</u>	<u>DATA-HI</u>			
11	BYTE 1	BYTE 2				
12	BYTE 1	<u>ADDR-LO</u>	<u>ADDR-HI</u>			
13	BYTE 1					
14	BYTE 1	<u>IP-INC-LO</u>	<u>IP-INC-HI</u>			
15	BYTE 1	<u>IP-LO</u>	<u>IP-HI</u>	<u>CS-LO</u>	<u>CS-HI</u>	
16	BYTE 1	<u>IP-INC8</u>				

Figure 1 Instruction Format Types

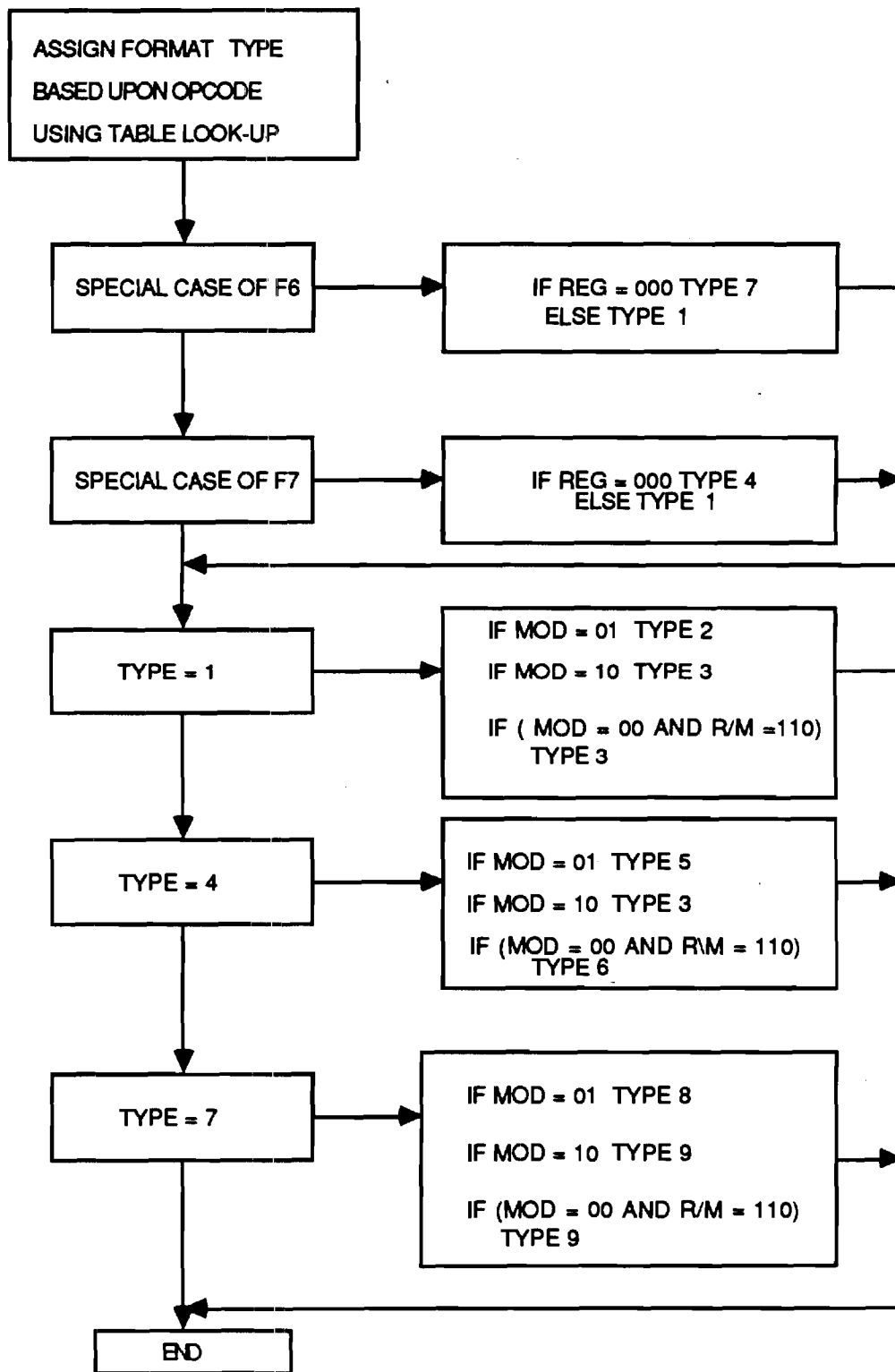


Figure 2 Algorithm Flow Chart

INSTRUCTION FIRST BYTE	INSTR. TYPE
00	1
01	1
02	1
03	1
04	11
05	10
06	13
07	13
08	1
09	1
0A	1
0B	1
0C	11
0D	10
0E	13
0F	13
10	1
11	1
12	1
13	1
14	11
15	10
16	13
17	13
18	1
19	1
1A	1
1B	1
1C	11
1D	10
1E	13
1F	13
20	1
21	1
22	1
23	1
24	11
25	10
26	13
27	13
28	1
29	1
2A	1
2B	1
2C	11
2D	10
2E	13
2F	13

Figure 3. 256 Possible TYPE Assignments Based on First Byte

INSTRUCTION FIRST BYTE	INSTR. TYPE
30	1
31	1
32	1
33	1
34	11
35	10
36	13
37	13
38	1
39	1
3A	1
3B	1
3C	11
3D	10
3E	13
3F	13
40	13
41	13
42	13
43	13
44	13
45	13
46	13
47	13
48	13
49	13
4A	13
4B	13
4C	13
4D	13
4E	13
4F	13
50	13
51	13
52	13
53	13
54	13
55	13
56	13
57	13
58	13
59	13
5A	13
5B	13
5C	13
5D	13
5E	13
5F	13

Figure 3. Continued

INSTRUCTION FIRST BYTE	INSTR. TYPE
60	13
61	13
62	13
63	13
64	13
65	13
66	13
67	13
68	13
69	13
6A	13
6B	13
6C	13
6D	13
6E	13
6F	13
70	16
71	16
72	16
73	16
74	16
75	16
76	16
77	16
78	16
79	16
7A	16
7B	16
7C	16
7D	16
7E	16
7F	16
80	7
81	4
82	7
83	7
84	1
85	1
86	1
87	1
88	1
89	1
8A	1
8B	1
8C	1
8D	1
8E	1
8F	1

Figure 3. Continued

INSTRUCTION FIRST BYTE	INSTR. TYPE
90	13
91	13
92	13
93	13
94	13
95	13
96	13
97	13
98	13
99	13
9A	15
9B	13
9C	13
9D	13
9E	13
9F	13
A0	12
A1	12
A2	12
A3	12
A4	13
A5	13
A6	13
A7	13
A8	11
A9	10
AA	13
AB	13
AC	13
AD	13
AE	13
AF	13
B0	11
B1	11
B2	11
B3	11
B4	11
B5	11
B6	11
B7	11
B8	10
B9	10
BA	10
BB	10
BC	10
BD	10
BE	10
BF	10

Figure 3. Continued

INSTRUCTION FIRST BYTE	INSTR. TYPE
C0	13
C1	13
C2	10
C3	13
C4	1
C5	1
C6	7
C7	4
C8	13
C9	13
CA	10
CB	13
CC	13
CD	11
CE	13
CF	13
D0	1
D1	1
D2	1
D3	1
D4	11
D5	11
D6	13
D7	13
D8	11
D9	1
DA	1
DB	1
DC	1
DD	1
DE	1
DF	11
E0	16
E1	16
E2	16
E3	16
E4	11
E5	11
E6	11
E7	11
E8	14
E9	14
EA	15
EB	11
EC	13
ED	13
EE	13
EF	13

Figure 3. Continued

INSTRUCTION FIRST BYTE	INSTR. TYPE
F0	13
F1	13
F2	13
F3	13
F4	13
F5	13
F6	7
F7	4
F8	13
F9	13
FA	13
FB	13
FC	13
FD	13
FE	1
FF	1

Figure 3. Continued

an instruction. As an example the first eight entries of Figure 3 are repeated below with the corresponding first instruction byte:

First Byte	Type
00	1
01	1
02	1
03	1
04	11
05	10
06	13
07	13

The table entries are determined from Table 1-23 by assuming that the instructions contain no displacements and a type is assigned on that basis. The table type value is then changed later in the algorithm as shown in the flow chart (Figure 2) based upon the MOD bits. The complete table contains an initial type assignment for each of the 256 possible cases for the first byte of an instruction.

The format types contained in the table contain types which appear to be redundant. For example, Type 1 and Type 11 appear the same. The difference is that Types 1,4, and 7 are types that require that the MOD field be examined to determine the number of displacement bytes that are included in the instruction. All other types are not affected by the MOD bits. The algorithm uses Types 1,4, and 7 to signify that the algorithm must examine the MOD bits to make a final determination of the format of the instruction.

Once the instruction format type is known, the number of bytes for that instruction is known, as well as the location of the bytes in the instruction that may change upon instruction relocation. This information is critical in determining which bytes to ignore during a file comparison so as to determine when two segments of code are identical except for instruction relocation offsets.

HEADER IDENTIFICATION

An executable program consists of two parts. The first part is a header record that contains control and relocation information. The second part is the actual load module. The header record contains information about the size of the executable module, where it is to be loaded into memory, and relocation offsets to be inserted into incomplete machine addresses. The header fields and their explanation are shown in Figure 4.

The size of the header may be determined by the following procedure. In the executable file, obtain the hex values in the ninth and tenth bytes of the file. (Count the first hex byte as one, not zero.) The tenth byte is most significant, the ninth byte is least significant. This hex value should be converted to decimal and then multiplied by 16 since the header size is given in 16-byte increments. This is the size of the header in bytes.

CODE SEGMENT IDENTIFICATION

The code segment may be located by using the ".map" output of the linker. The beginning of the code segment in the output from the Microsoft PASCAL compiler is located two bytes after the header; therefore, the first code segment's location, relative to the start of the executable file, is at an offset value equal to the size of the header plus two bytes. The end of the code segment is determined from the ".map" file, which is an output from the linker. The procedure for determining the end of the code segment is as follows. In the ".map" file as output by the linker, find the class of "encode" shown in the "class" column. In the column "start" is shown the end of the code segment. This value is relative to the header size and must be added to the header size which is determined from the header as discussed above. After adding the header size to the code segment value, the end of the code segment's location, relative to the beginning of the executable file, is now known.

DATA SEGMENT IDENTIFICATION

The start of the data segment is determined from the end of the code

Relative Hex Position:	Field:
00	Hex 4D5A. The Linker inserts this code to identify the file as a valid EXE file
02	Reserved
04	Size of the file including the header, in 512-byte increments ("pages")
06	Number of relocation table items following the formatted portion of the header
08	Size of the header in 16-byte increments. The purpose of this field is to help locate the start of the executable module that follows this header
0A	Reserved
0C	High/low loader switch. You decide at the start of LINK whether your program is to load for execution at a low (the usual) or a high memory address. Hex 0000 indicates high and hex FFFF indicates low
0E	Offset location in the executable module of the Stack Segment
10	Address that the Loader is to insert in the SP register when transferring control to the executable module
12	Checksum value - the sum of all the words in the file (ignoring overflows) used as a validation check for lost data
14	The offset that the Loader is to insert in the IP register when transferring control to the executable module
16	The offset location in the executable module of the Code Segment
18	The offset of the first relocation item in this file
1A	Reserved
1B	Relocation table containing a variable number of relocation items, as identified at offset 06

Figure 4 Header Fields and their Explanations

segment. The data segment begins at the end of the code segment and continues to the end of the executable file. This may not always be true, but it is true for all Microsoft Pascal examples run during this effort.

EXAMPLE OF SEGMENT IDENTIFICATION

To illustrate the procedure for determining the locations of the various segments in an executable file, the following example is provided. The required input information is the ".map" file from the linker, and the ".exe" file. An example ".map" file is shown in Figure 5, and an example header portion for the same example is shown in Figure 6.

The procedure to identify the segments is as follows:

Step 1: In the ".exe" file obtain the hex values in the ninth and tenth bytes of the file. In this example the values are 60 and 00, respectively.

Step 2: Convert this value to decimal and multiply the result by 16. In this example, this is 96 times 16, indicating a header size of 1536 bytes.

Step 3: Determine the beginning of the code segment by adding two bytes to the size of the header; therefore, the code segment in this example begins at 1538 bytes into the file.

Step 4: In the ".map" file as output by the linker, find the class of "encode" shown in the "class" column. In the column "start" is shown the hex location of the end of the code segment. In this example, this value is 0436E hex.

Step 5: Add the header size to the value obtained in step 2. The result is the end of the code segment and the beginning of the data segment. In this example, this value is $1536 + 17262$, which yields 18798 bytes.

The summary of the results from this example are:

Header begins at 0 and ends 1536 bytes into the file
Code begins at 1538 and ends 18798 bytes into the file
Data begins at 18799 and ends at the end of the file


```

04FECH 04FECH 00000H PBC          DATA
04FECH 04FECH 00000H P2DE        DATA
04FECH 04FECH 00000H P3DB        DATA
04FECH 04FEFH 00004H P3C          DATA
04FECH 04FECH 00000H P3CE        DATA
04FF0H 04FF0H 0000EH C0DATA      DATA
04FF0H 05180H 00180H CONST      CONST
05180H 05180H 00000H E0DATA      ENDDATA
05180H 05180H 00000H EEND        ENDBSS
05180H 05180H 00000H _BSS        BSS
05190H 0598FH 00900H STACK       STACK

```

```

Origin      Group
0437:0      DGROUP

```

program entry point at 0004:06F0

Figure 5 Example of ".map" File
(continued)

FILE- COMPARISON APPROACHES

In a typical situation only minor changes have been made to the update program. In that case many, perhaps most of the instructions will be the same except for minor differences. The minor differences include different offsets in the data, displacements, or address fields. The approach to compare these files must be able to recognize matching instruction pairs that are either identical or different only in the minor ways mentioned above and explained later and to account for these minor changes in the transmission process so that the new file can be reconstructed from the old with the correct modifications.

To this end a number of existing file-compare algorithms were examined to see if existing software could be used or adapted for this application. Notable among the papers found in the literature were: "A Fast Algorithm for Computing Longest Common Subsequence," by James Hunt of Stanford University and Thomas G. Szymanski, Communications of ACM, Volume 20, No. 5, May 1977; "Algorithms for the Longest Common Subsequence Problem," by Daniel S. Hirschberg of Princeton University, Journal of the Association for Computing Machinery, Volume 24, No. 4, October 1977, pages 664-675; and "File Comparison Algorithms," by Tom Steppe, Dr. Dobb's Journal of Software Tools, September 1987, pages 28-33 and 54-60. Of these three, the last proved to be most useful since it reviewed several types of algorithms including the longest common subsequence type referenced in the first two articles. Copies of the articles are included as Appendix B.

Basically, file-compare algorithms look for line matches, then report lines not included as matches as differences. The differences are usually expressed as insertions, deletions and changes that must be made to make the files match. Algorithms are evaluated to answer the questions: Is it efficient? Is it robust? Can it let differences go undetected? Can it let matches go undetected? Can it detect blocks of text that have been moved? There are several popular algorithms. The "scan until next match" algorithm starts at the tops of both files and matches as many lines as possible. When a difference is detected, the next M lines are scanned until at least N consecutive matches are found. The main advantages of this algorithm is time efficiency and minimal memory requirements; the main problem with this algorithm is that it is not robust over a variety of

situations. A second type of algorithm is called the "longest common subsequence" algorithm. This algorithm finds the longest common, though not necessarily consecutive, sequence of lines in the two files. This type of algorithm often produces the best reports when comparing files that do not involve moved blocks of code, but it can be slow. A third type, called "extended unique matching" is based on the idea that a line that occurs once and only once in each file must be the same line. These pairs of "unique" lines determine the initial set of unmatched lines. Then, in each file, the lines adjacent to each match are examined and, if identical, are added to the set of matched lines, and the process is repeated. Though efficient in time and space, it is prone to detecting false differences. A fourth type of algorithm, developed by Steppe, is called the "recursive longest matching sequence." This method scans both files looking for the longest sequence of consecutive lines. This block then divides the files into top and bottom halves, each of which is then scanned. The process is repeated until no more matches can be found. The space for this algorithm is linear but the time is quadratic. A modification to the algorithm can reduce the time required. All of these algorithms were designed to handle text files, though many of the concepts apply to executable files as well.

An algorithm, based in part on some of these concepts, was conceived to address the special case of the executable file. The basic idea is to do an instruction by instruction comparison on the code sections of the old and new files where a file may have been padded at the end with "never match" code to make them the same size. On the first pass the largest block of consecutive matches will be identified and the size and location noted. The files will then be compared again with one of the files shifted by one line, as if the file were wrapped around to form a circle with the beginning and ending lines of each file touching. This is the reason the files must be made the same length. The process continues until the largest possible block of matches has been found. Note that a match is declared even if the offsets in their data, displacements, or address fields do not match. This block is, in effect, set aside and the whole process is repeated with the remainders of the files, which hopefully are much smaller. This process is continued until it is no longer feasible to search for matches; that is, when the remaining code segments or pieces are of a size that it is more cost effective to transmit the code in toto than to spend the overhead to form packets, etc. to follow the scheme outlined. The algorithm for the update scheme will incorporate the file compare approach just described.

SECTION 4

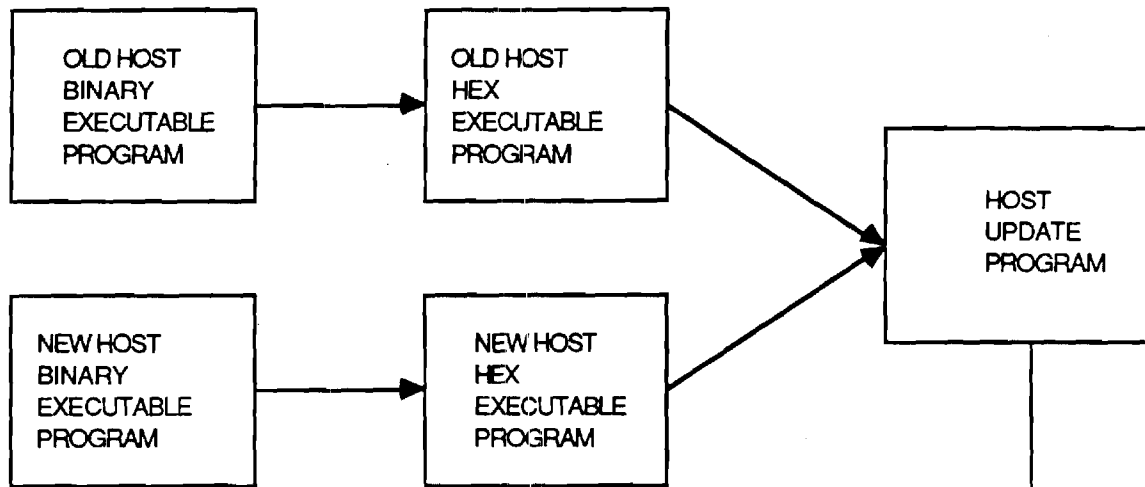
ALGORITHM

Overview

There are two algorithms involved in the transmission of modified code. The host computer algorithm and the remote computer algorithm, run on the host and the remote computers, respectively.

The host computer algorithm requires a copy of the old host binary executable program (the unmodified program) as well as the new host binary executable program (the new version of the program). The host converts both of these binary program versions to hexadecimal representations and then compares the two programs. This comparison identifies the code segments that are identical in both the old and the new programs including those code segments that are different only by offset values in the various instruction subfields. The host algorithm generates a host update file which is decoded by the remote computer algorithm so as to generate the necessary changes to the remote computer's old remote binary executable file. This is accomplished in the remote computer by first converting the old remote binary executable program into an old remote hexadecimal executable program version. The new remote hexadecimal executable program is generated from the old executable program instructions, which are modified as necessary, and the instructions which are transmitted to the remote computer in the host update file. After the new remote hexadecimal executable program is generated, the program is converted into the new remote binary executable program. Figure 7 is an illustration of the files that the algorithm uses and the resulting files that are generated by the algorithm.

Host Computer



Remote Computer

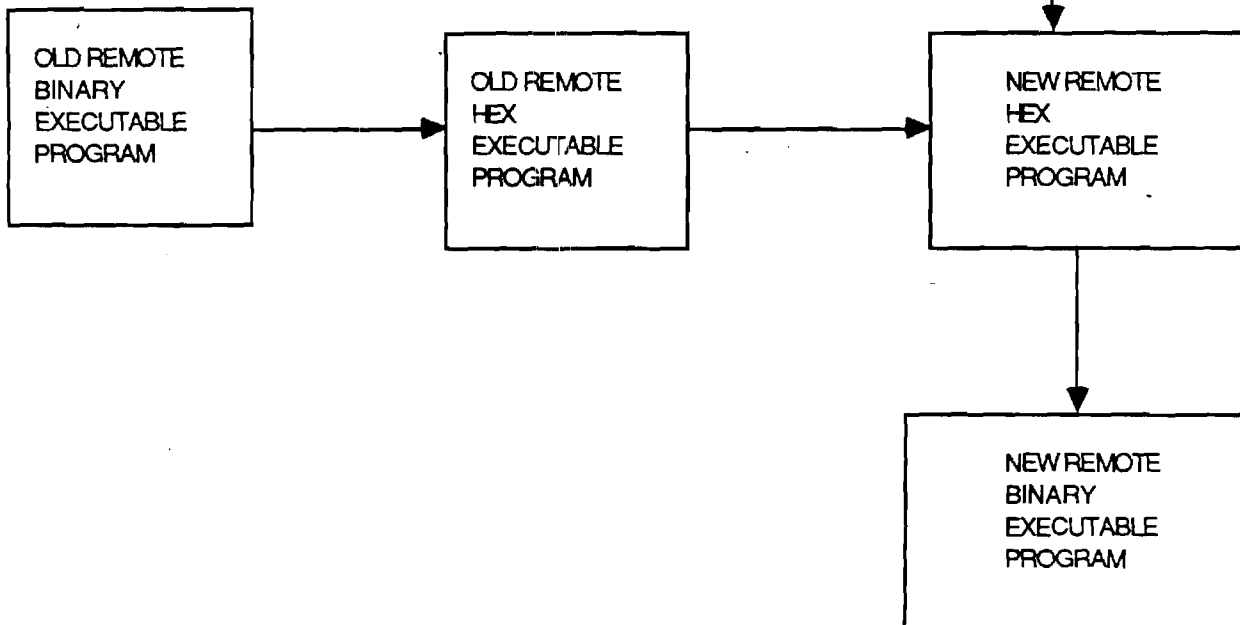


Figure 7 Programs Used by the Update Algorithms

HOST UPDATE ALGORITHM

The host update algorithm is shown in Figure 8. The algorithm begins by converting the binary executable files of both the old and the new programs into hexadecimal representations. The hexadecimal representation of the new program is then used to determine the size of the new header.

The data section of the new executable file as determined from the new header and the ".map" output of the compiler is then identified and is stored in the host update file for transmission as a new packet.

In the next step of the algorithm, the old executable code segment is compared to the new executable code segment with all of the offset fields of each instruction nulled out. This nulling of all instruction offsets allows for the identification of code segments that are identical except for offsets. The offset fields that are nulled out are identified earlier in this report in the section entitled "INSTRUCTION FORMATS".

The compare algorithm orders the matching code segments with the largest matching code segment first. Each succeeding code segment is smaller in size.

Once the matching code segments have been identified, the lines in the new executable code which are not matches with the old code are identified. These code segments are to be transmitted to the remote computer as new packets.

After identifying the instructions in the new program which may be generated from the old program that resides in the remote computer, the process of determining the required offsets that may be added to the old resident code is initiated. During this process the actual transmission packets are formed. Using the matching code segment list, a pointer to the old executable code where the largest matching code segment is located is set to the corresponding value. These pointers will be transmitted to the remote computer to identify where to obtain the old code segments and where to place these offset modified instructions in the new code.

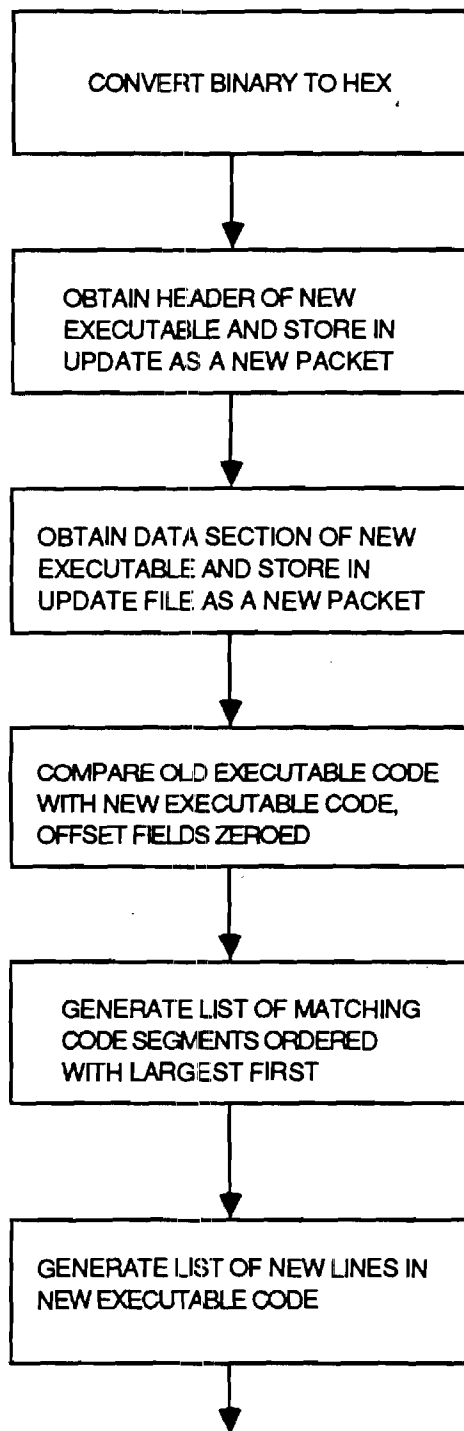


Figure 8 Host Update Algorithm

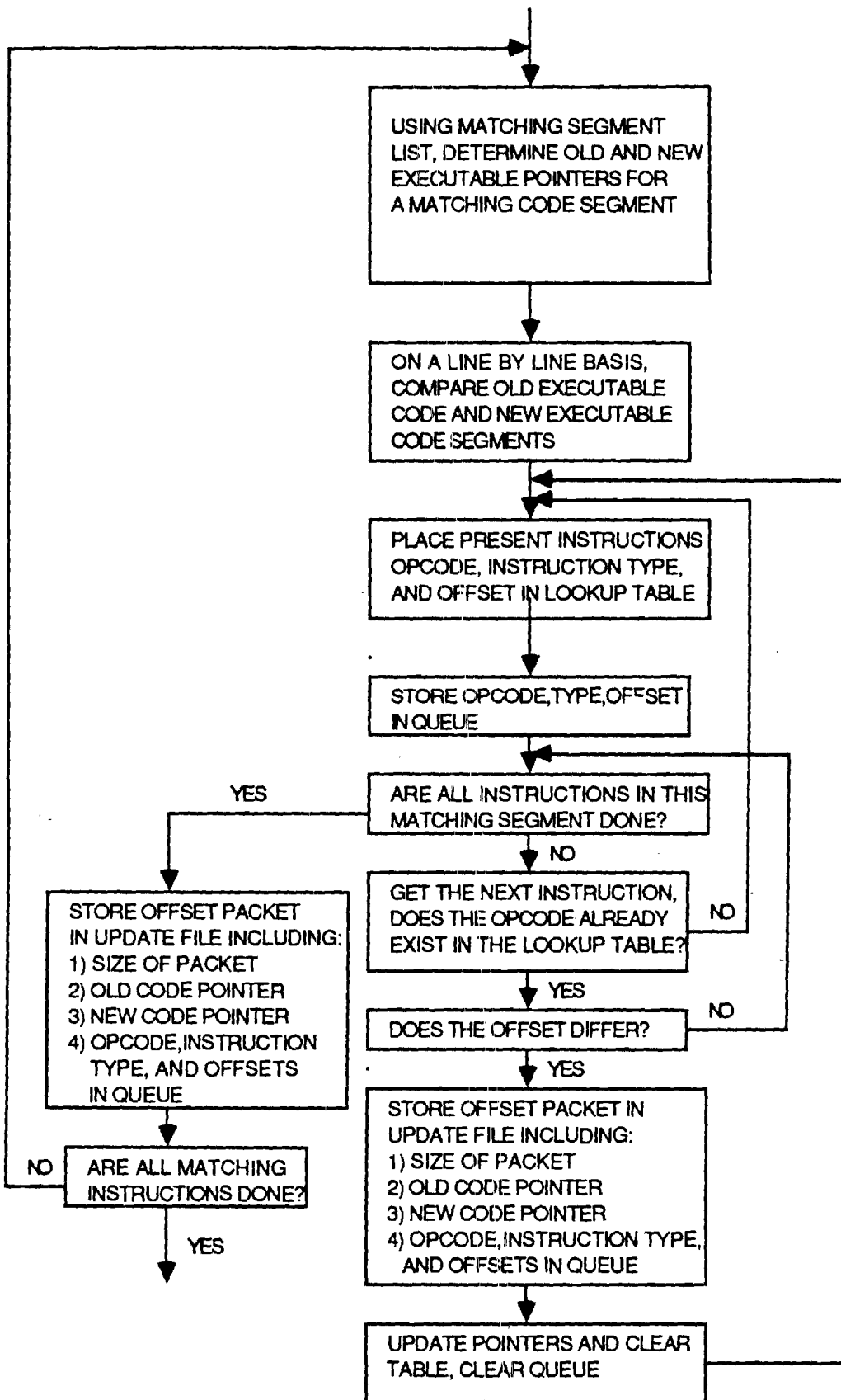


Figure 8 Host Update Algorithm (continued)

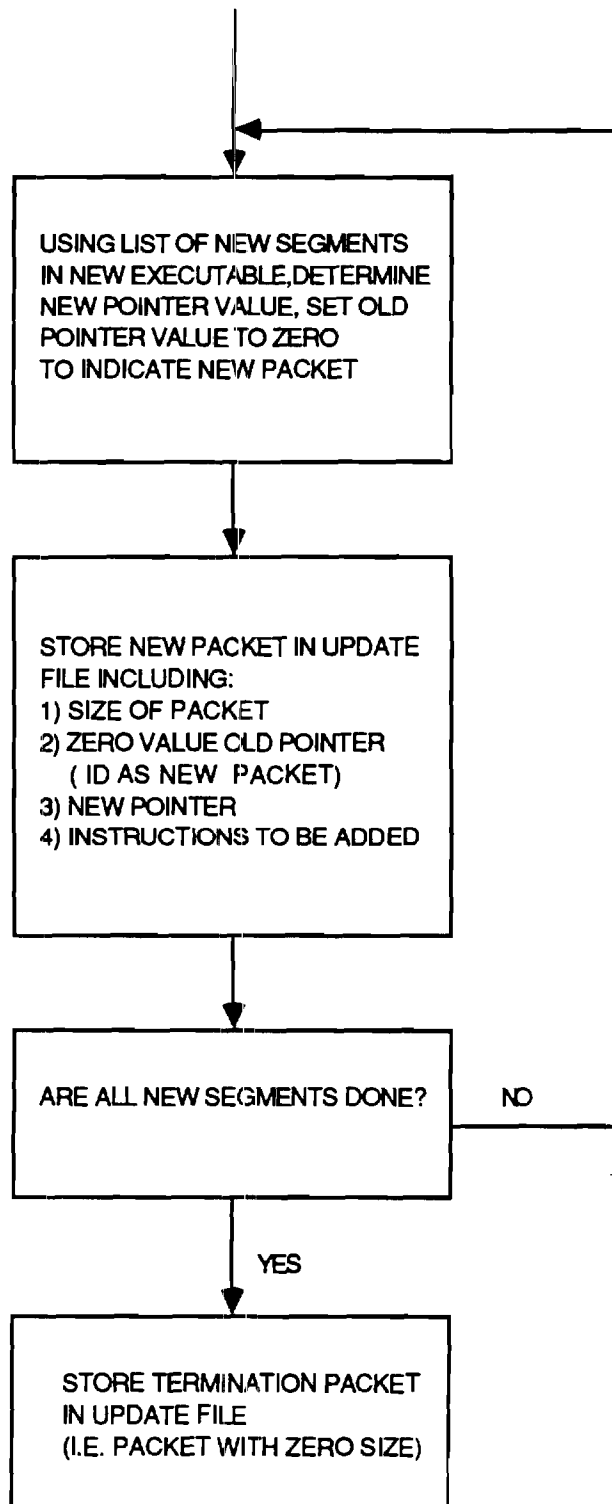


Figure 8 Host Update Algorithm (concluded)

On a line by line basis, the old executable instruction is compared to the new executable instruction. The opcodes are always the same since the compare algorithm used in previous steps has identified the matching segments (with the offsets nulled). The offsets are not nulled out in this portion of the algorithm, so any offset that differs from the instruction's old location and it's new location are identified. Each instruction that contains an offset field may have one, two, three, or up to four bytes of offsets associated with it. The algorithm treats each byte independently and determines the value of an offset byte that is added to the respective instruction's offset byte to obtain the new correct offset byte. The offsets that are required because of code relocation may be either a positive or negative offset. The algorithm always adds the offset byte ignoring any carry out that may be generated. This approach does not implement a signed addition, so that true addition or subtraction is carried out (i.e. not a two's complement type addition); however, a number can always be identified that may be added to obtain the correct final offset value.

Once the offset value for a given instruction is determined, the instruction's opcode, instruction format, and offset values are stored in a dynamic lookup table. The opcode and the instruction type identify how many offset fields are in the instruction and where they are located. For each different opcode and instruction type combination encountered, an entry in the offset table is made. For each instruction which is encountered in a given matching segment, a search in the offset table is made. If the instruction does not already exist in the offset table, it is added, even if all offsets in that instruction are zero. If an encountered instruction does exist in the offset table, the offset of the present version of the instruction must match the previous entry or else the present instruction cannot be included in the present packet. Once all of the instructions in a given matching code segment have been entered into the offset table, or in the event of an instruction that cannot be included in the present packet, the packet itself is formed and entered into the host update file.

A packet which is formed in this part of the algorithm is an offset packet which consists of the following information:

- 1) the size of the packet
- 2) the old code pointer which identifies where the old code is located in the old executable file
- 3) the new code pointer which identifies where the old code will be placed in the new executable file after the instructions are modified as specified in the offset lookup table
- 4) the offset lookup table for this packet of instructions. This included the opcode, the instruction format type, and the offsets for each different instruction contained in the packet.

Once a packet is formed, the offset table is cleared and the process is repeated for the next group of instructions until the groups of matching instructions are depleted.

At this point, the packets of new instructions must be formed. The output of the algorithm includes a list of the new code segments in the new executable file. This list is used to determine which instructions must be transmitted to the remote computer as new code. Using this list of new code segments, a pointer to the location in the new executable file is set, indicating where the new instructions will be placed. A new instruction packet is then formed. This new instruction packet contains the following:

- 1) the size of the packet
- 2) the old program pointer, which is set to a zero value since the new code does not come from any part of the old program and a zero value in the pointer allows the receiver algorithm to identify this packet as a new instruction packet

- 3) the new program pointer which indicates where in the new executable program the new instructions will be placed
- 4) the actual instructions which are to be placed in the new executable program

After all new instruction packets have been formed, the update file has a packet with zero size placed at the end to identify that the end of the file has been reached.

UPDATE FILE FORMAT

The update file (as shown in Figure 9) consists of the following information in this order:

- 1) The new header information which is placed in the file as new instruction packets. This allows the new header to be transmitted as is with no modifications.
- 2) The new data segment information which is placed in the file as new instruction packets. This allows the new data segments to be transmitted as is with no modifications.
- 3) The offset packets which contain the locations of usable old instructions and the new offsets which should be used in the new version
- 4) The new instruction packets which contain those new instructions that are to be added to the new program

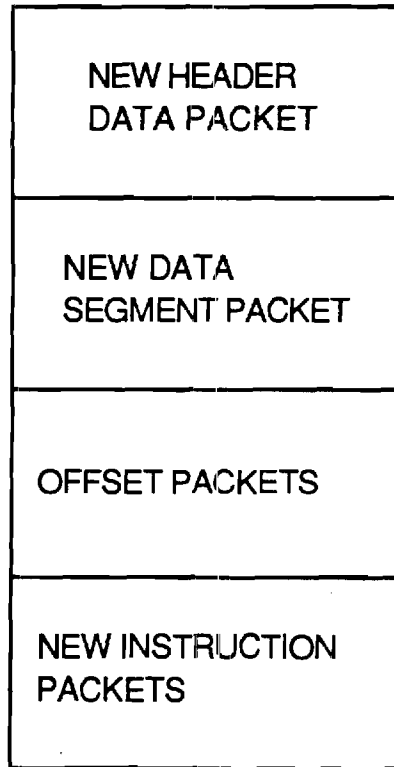


Figure 9 Update File Format

OFFSET PACKET FORMAT

The offset packets contain the size, pointers, and instruction offsets in the following order:

- 1) size of packet (2 bytes)
- 2) old pointer (4 bytes)
- 3) new pointer (4 bytes)
- 4) opcode, instruction type, offsets (variable size)
- 5) opcode, instruction type, offsets (variable size)
- 6) etc., until all different instructions in packet listed. Thus the size of the packet is variable depending upon the number of entries in the offset list

NEW PACKET FORMAT

The new packets contain the size, pointers, and instructions in the following order:

- 1) size of packet (2 bytes)
- 2) old pointer set to zero value (4 bytes)
- 3) new pointer (4 bytes)
- 4) complete instruction (variable size)
- 5) complete instruction (variable size)
- 6) etc., until all new code has been include

REMOTE UPDATE ALGORITHM

The remote computer algorithm accepts as input the update file which has been embedded in it the packets which are used to generate the new code on the remote computer. Thus, the inputs to the remote algorithm are the old remote executable file, and the new update file. The output of the remote algorithm is the new remote executable file. The remote algorithm is shown in Figure 10.

The remote algorithm begins by converting the old remote executable binary file to a hexadecimal representation. Portions of this executable file are used to generate the new executable file.

The update file is read next to identify a packet. The packet size is read to determine if the algorithm is completed. For a nonzero packet size, the old pointer value is read and a pointer is set to this location in the old remote executable file. The new pointer is read next, a pointer to indicate where the code will be placed in the new executable file set. If the old pointer is set to a zero value, the packet is a new instruction packet and all of the new instructions contained in the packet should be written to the new executable file as is.

In the event that the old pointer is equal to a zero value, the packet is an offset packet. The instruction opcodes, instruction type, and offsets are read in and used in a table lookup scheme to modify all instructions that are encountered in the old executable code segment that is being written into the new executable.

After the entire packet is written into the new executable file, the process is repeated until all packets have been transformed into the new code. The process ends when a zero size packet is encountered.

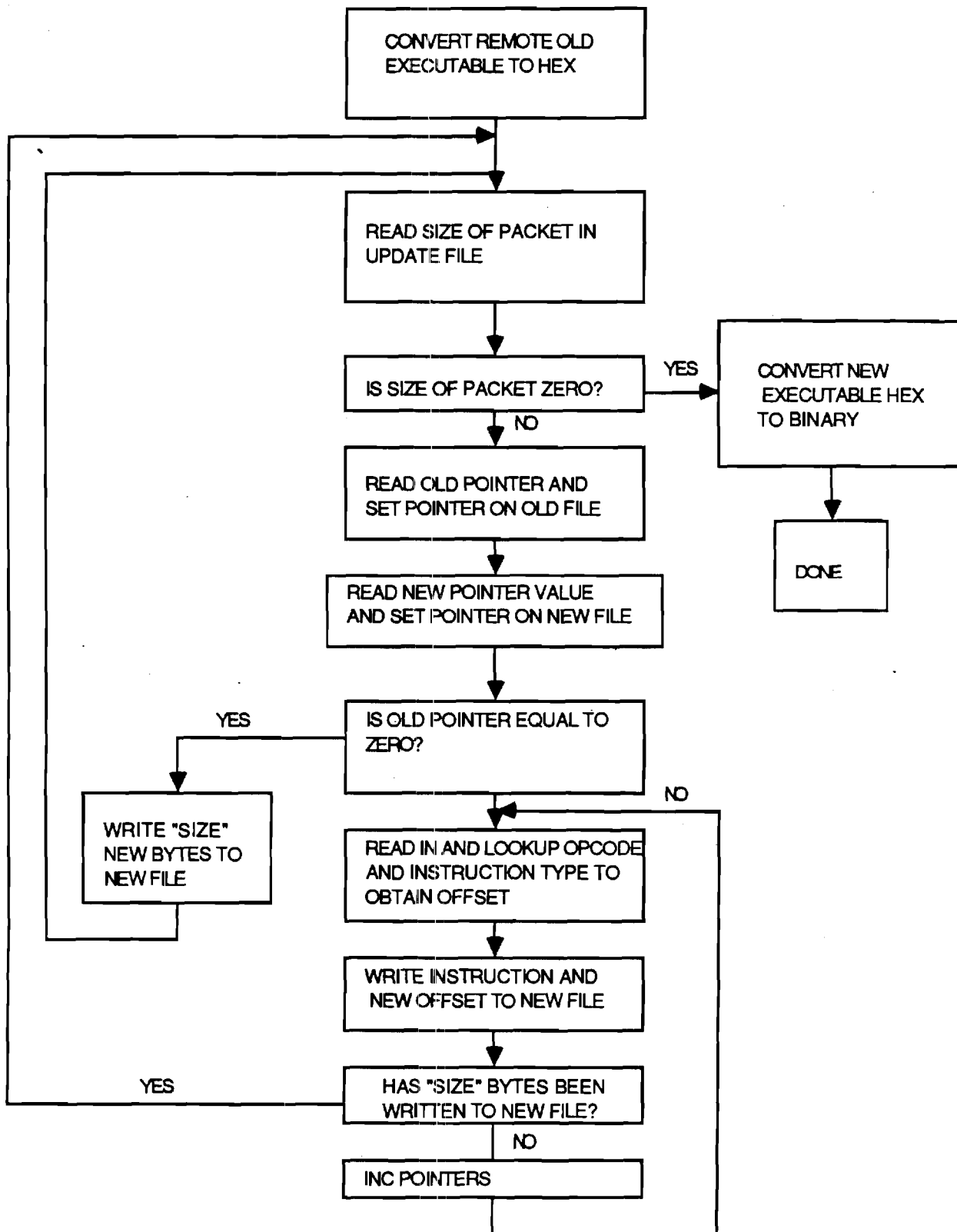


Figure 10 Remote Update Algorithm

SECTION 5

SOFTWARE DEVELOPMENT

An integral part of the design and implementation of an algorithm is software development. Several programs were written both to implement and to test various parts of the algorithm. These programs are listed and described below and are illustrated in corresponding figures, Figures 11-17.

1. CDC Libraries

CONVERT- converts binary integers to ASCII HEX for printing

MODTYPE- implements the algorithm in Figure 2 that modifies the instruction conversion table

BLKDISP- finds the displacement fields in an instruction and makes the fields equal to zero

2. RDBIN.PAS

This program is a byte by byte listing of any file in decimal and hexadecimal form. It is used as a debugging tool to look at headers and data segments.

Inputs:

File to be listed

Outputs:

Decimal listing

Hexadecimal listing

CDC LIBRARY – CDC1.LIB

CONVERT

INTHEX

MODTYPE

BLKDISP

Figure 11 CDC Libraries

READ BINARY – RDBIN.PAS

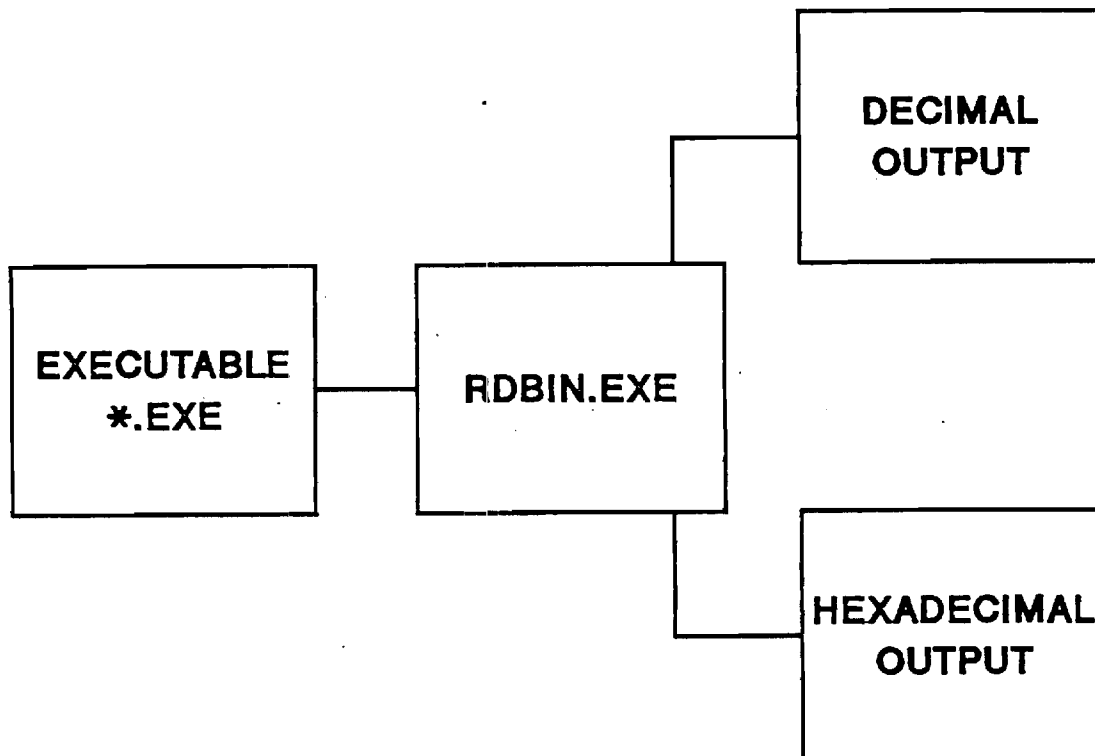


Figure 12 Flow Chart of RDBIN.PAS Program

LIST EXECUTABLE - LEXEC5.EXE

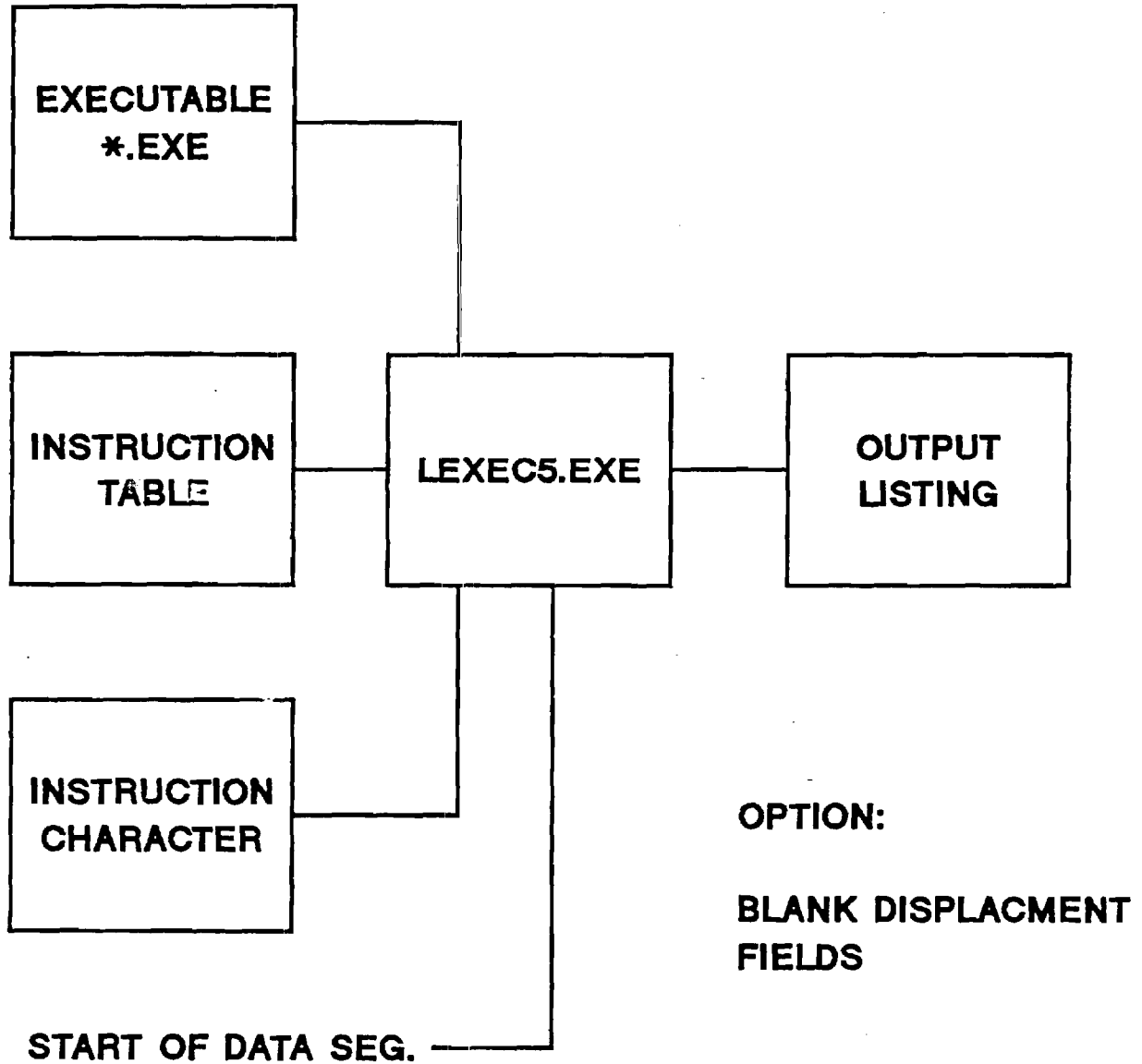


Figure 13 Flow Chart of LEXEC.PAS Program

COMPARE EXECUTABLE - COMPEX.PAS

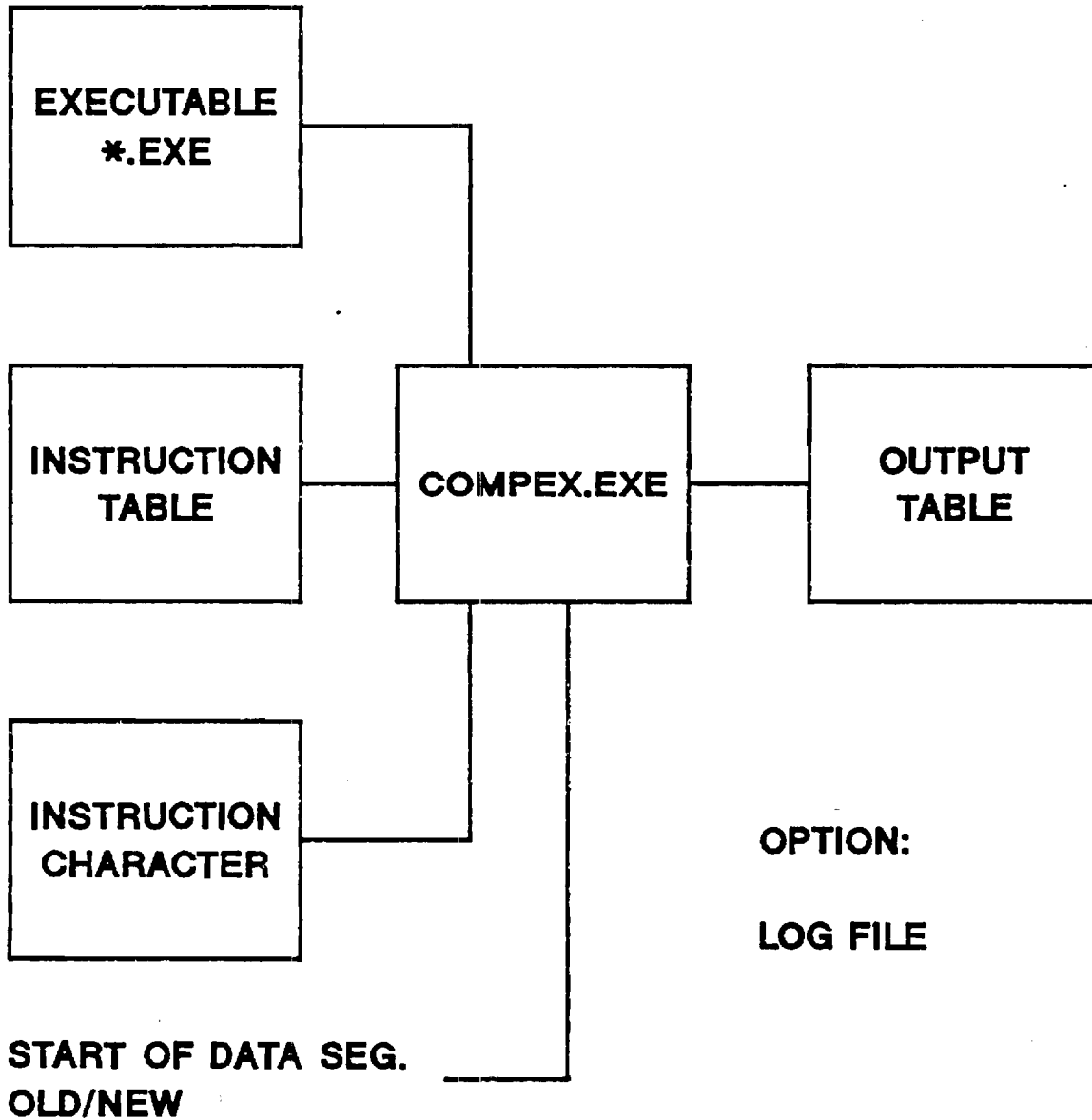


Figure 14 Flow Chart of COMPEX.PAS Program

BUILD PACKETS - BPACK.PAS

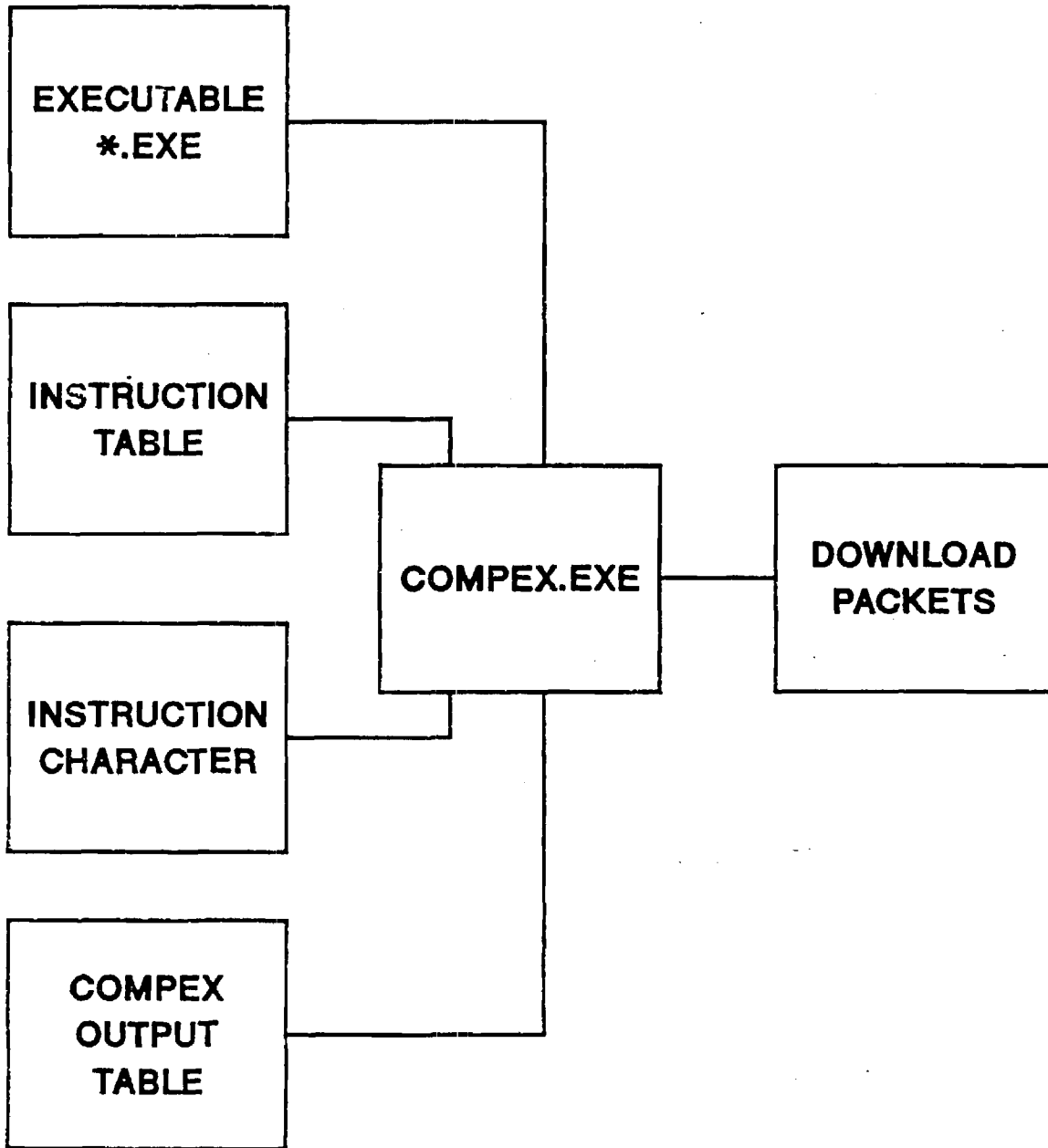


Figure 15 Flow Chart of BPACK.PAS Program

**COM1 PORT ACCESS
ASSEMBLY LANGUAGE SUBROUTINES**

IC1LS - Read Line Status

IC1MS - Read Modem Status

IC1INT - Initialize Configuration

IC1MCO - Set Modem Control Lines

IC1TD - Send One Character

IC1TDW - Send One Character / Wait

IC1RD - Read One Character

IC1RDW - Read One Character / Wait

Figure 16 List of Assembly Language Subroutines
of COM1 Port Access

**COM1 PORT ACCESS
FORTRAN SUBROUTINES**

CIRSTR – Read a String of Characters

CISSTR – Send a String of Characters

Figure 17 List of FORTRAN Subroutines
of COM1 Port Access

3. LEXEC5.PAS

This program lists a file in instruction format and automatically finds the beginning of the code section from data in the header. The header and data segment are listed byte by byte in Hex form. The code section is listed by instruction in Hex form.

Inputs:

- Executable file to be listed
- Instruction Table- Instructions / instruction type
- Instruction Character- Number of bytes and displacement fields of instruction types
- Start of data segment- Determined for Linker Map

Outputs:

- Listing of header, code and data segment

Options:

- Blank displacement fields

4. COMPEX.PAS

This program, the implementation of which is illustrated in Figure 8, compares files for matching blocks. The displacement fields are blanked and the files are compared for largest matching blocks.

Inputs:

- Old executable file
- New executable file
- Instruction Table
- Instruction character
- Start of data segment for old and new program

Outputs:

- Block table of matching blocks

Options:

- Generate a log file of each compare iteration for debugging

5. BPACK.PAS

This program builds packets for the download file. The displacement fields are not blanked. The offsets are determined and the packets formed. Header and Data segments are sent as new data.

Inputs:

- Old executable file
- New executable file
- Instruction table
- Instruction character
- Output from COMPEX program

Outputs:

- Packet ready for download

6. COM1 Port Access Programs

The protocol for the IBM personal computer COM1 communication port has a number of features that make computer-to-computer communications difficult when operating under DOS. These features involve both the hardware and software of the communication port. Below are described the problem features as well as a group of assembly language subroutines that have been written to control the port directly from a higher-level language.

The following hardware and software features create problems in communicating between two computers under DOS:

1. If a character has arrived at the COM1 communication port prior to the operating system accessing the port, the system declares the receiver-full status to be a device error and will cause a run-time error.
2. When the communication port is initialized to receive data, the clear-to-send line is set high, but the data terminal ready line is set low. If the full hardware handshake lines are implemented between COM1 communication ports, the receiving port will inhibit the sending port and no data will be transmitted.

3. The software termination for a DOS input on the COM1 port is a control Z character, but the output protocol is not compatible with the input protocol.

4. The software protocol automatically sends both a carriage return and a line feed at the end of a string output, but the input only requires a carriage return to terminate the string input. This extra line feed character sent on the output usually ends up as a character left in the receiver register at the end of a transmission, which results in a run-time error as described in item 1.

The following assembly language programs were written to allow a user to access the COM1 port directly from a high-level program.

IC1LS - Reads the line status of the COM1 port

IC1MS - Reads the modem status of the COM1 port

IC1INT - Initializes the COM1 port configuration

IC1MCO - Sets the value of the COM1 port modem control lines

IC1TD - Sends one character to the COM1 port

IC1TDW - Sends one character to the COM1 port after the transmitter buffer is empty

IC1RD - Reads one character from the COM1 port with a null indicating no character is available

IC1RDW - Reads one character from the COM1 port after a character becomes available

The following FORTRAN subroutines were written to supplement the

assembly language routines:

C1RSTR - Reads a string of characters into a character array from the COM1 port

C1SSTR - Sends a string of characters from a character array to the COM1 port

SECTION 6

FUTURE DIRECTIONS

HEADER ALGORITHM

The header record contains information about the size of the executable module, where it is to be loaded in memory, and where the address of the stack register and relocation offsets are to be inserted into incomplete machine addresses. The largest amount of information in the header consists of the relocation table containing the relocation items. Each relocation item consists of a two-byte offset value and a two-byte segment value.

At the present the proposed algorithms do not attempt to capitalize on the similarities between an old program header and a new program header. It may be possible to develop an algorithm that can use the old header relocation table to derive the new header relocation table.

DATA SEGMENT ALGORITHM

The proposed procedure for transmitting the changes from a host computer to a remote computer does not attempt to use the old data segments in the generation of the new data segments. At present the new data segment is transmitted in its entirety. The new data segments should be derivable from the old data segments in such a manner as to reduce the amount of information required to be transmitted to generate the new executable file data segments. An additional algorithm can be developed to handle this portion of the files.

FILE-COMPARE ALGORITHM EFFICIENCY

There are several issues that should be addressed that have the potential to improve the file-compare efficiency. The first issue relates to large

amount of memory required with the proposed scheme. For a file of size = N , the present required array size is $6N$. This is hard to accomplish on a PC in PASCAL. Though there is a memory of size 640K bytes, the compiler limits its use to 64K. Perhaps a solution to this problem can be found by using a mainframe, another language, or maybe finding a way around the compiler to get access to the rest of the memory. The second issue relates to the fact that the present scheme requires a full copy of both the old and new files, and hence a large memory. It is true, however, that the operation is done only once for many downloads and that the operation can be done on a mainframe. These advantages may outweigh the disadvantage of a large memory requirement.

Prototype Code Demonstration

A prototype code demonstration should be prepared. There are five steps to the development of a prototype code demonstration, some of which have been executed already. The code listing software, LEXEC5, and the compare algorithm software, COMPEX, are complete. The packet formation software, BPACK, has been written, but has not been debugged. The regeneration software has been flow charted and is illustrated in Figure 10, but has not been coded or debugged. A demonstration of the program has not been debugged. The last three tasks need to be completed before a demonstration can be presented.

Analysis of Packet Size Versus Overhead Bytes

A certain number of overhead bytes are required in the preparation and transmission of each packet. This number varies with the contents of a particular packet. Clearly a packet corresponding to a large block of code warrants the overhead bytes required for its transmission; a packet corresponding to a very small block of code may not. An analysis needs to be done to determine at what point it is more economical to send a block of code in toto as opposed to sending a packet of information describing how to modify the old code to mirror the new code.

Examination of Methods for Error Correction and Encryption

When large amounts of data are transmitted over commercial phone lines, the issues of error detection and correction and data encryption are of primary concern. Methods addressing both of these issues were identified and briefly reviewed early in this effort. Information embedded in overhead bytes is used to detect and correct errors using a variety of pattern recognition techniques such as the higher-order correlation matrix associative memory method of Shiozaki. The various techniques identified in the literature need to be carefully evaluated for their appropriateness to the update scheme developed under this effort.

Through efforts by The National Bureau of Standards and others, data encryption techniques have been greatly improved over the years, especially with the adoption of the IBM-based, DES algorithm under the American National Standards Institute's title, "Data Encryption Algorithm." Work, such as Cipher Block Chaining, which establishes a chained relationship between successive blocks of ciphertext and detects unauthorized modifications, continues to improve encryption techniques. An investigation of these and other pattern recognition techniques pertinent to the data security in the proposed scheme should be investigated.

APPENDIX A

Table 1-22 8085/88 Instruction Encoding

DATA TRANSFER

MOV = Move:

Register/memory to/from register

Immediate to register/memory

Immediate to register

Memory to accumulator

Accumulator to memory

Register/memory to segment register

Segment register to register/memory

7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

1 0 0 0 1 0 d w	mod r1g r/m	(DISP-LO)	(DISP-HI)		
1 1 0 0 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w = 1
1 0 1 1 w reg	data	data if w = 1			
1 0 1 0 0 0 0 w	addr-to	addr-to			
1 0 1 0 0 0 1 w	addr-to	addr-to			
1 0 0 0 1 1 1 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 1 1 0 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		

PUSH = Push:

Register/memory

Register

Segment register

1 1 1 1 1 1 1 1	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)		
0 1 0 1 0 reg					
0 0 0 reg 1 1 0					

POP = Pop:

Register/memory

Register

Segment register

1 0 0 0 1 1 1 1	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)		
0 1 0 1 1 reg					
0 0 0 reg 1 1 1					

Table 1-22 8086/88 Instruction Encoding (continued)

DATA TRANSFER (Cont'd.)

EXCHG = Exchange:

Register / memory with register

7 0 6 4 3 2 1 0 7 0 6 4 3 2 1 0 7 0 6 4 3 2 1 0 7 0 6 4 3 2 1 0 7 0 6 4 3 2 1 0 7 0 6 4 3 2 1 0

1 0 0 0 0 1 1 w	mod	reg	r/m	(DISP-LO)	(DISP-HI)
-----------------	-----	-----	-----	-----------	-----------

Register with accumulator

1 0 0 1 0 reg

IN = Input from:

Fixed port

1 1 1 0 0 1 0 w	DATA-I
-----------------	--------

Variable port

1 1 1 0 1 1 0 w

OUT = Output to:

Fixed port

1 1 1 0 0 1 1 w	DATA-O
-----------------	--------

Variable port

1 1 1 0 1 1 1 w

XLAT = Translate byte to AL

1 1 0 1 0 1 1 1

LEA = Load EA to register

1 0 0 0 1 1 0 1	mod	reg	r/m	(DISP-LO)	(DISP-HI)
-----------------	-----	-----	-----	-----------	-----------

LDS = Load pointer to DS

1 1 0 0 0 1 0 1	mod	reg	r/m	(DISP-LO)	(DISP-HI)
-----------------	-----	-----	-----	-----------	-----------

LES = Load pointer to ES

1 1 0 0 0 1 0 0	mod	reg	r/m	(DISP-LO)	(DISP-HI)
-----------------	-----	-----	-----	-----------	-----------

LAMF = Load AH with flags

1 0 0 1 1 1 1 1

SAMF = Store AH into flags

1 0 0 1 1 1 1 0

PUSHF = Push flags

1 0 0 1 1 1 0 0

POPF = Pop flags

1 0 0 1 1 1 0 1

ARITHMETIC

ADD = Add:

Reg / memory with register to either

0 0 0 0 0 0 d w	mod	reg	r/m	(DISP-LO)	(DISP-HI)
-----------------	-----	-----	-----	-----------	-----------

Immediate to register / memory

1 0 0 0 0 0 s w	mod	0 0 0	r/m	(DISP-LO)	(DISP-HI)	data	data if s = 0
-----------------	-----	-------	-----	-----------	-----------	------	---------------

Immediate to accumulator

0 0 0 0 0 1 0 w	data	data if w = 1
-----------------	------	---------------

ADC = Add with carry:

Reg / memory with register to either

0 0 0 1 0 0 d w	mod	reg	r/m	(DISP-LO)	(DISP-HI)
-----------------	-----	-----	-----	-----------	-----------

Immediate to register / memory

1 0 0 0 0 0 s w	mod	0 1 0	r/m	(DISP-LO)	(DISP-HI)	data	data if s = 0
-----------------	-----	-------	-----	-----------	-----------	------	---------------

Immediate to accumulator

0 0 0 1 0 1 0 w	data	data if w = 1
-----------------	------	---------------

INC = Increment:

Register / memory

1 1 1 1 1 1 1 w	mod	0 0 0	r/m	(DISP-LO)	(DISP-HI)
-----------------	-----	-------	-----	-----------	-----------

Register

0 1 0 0 0 reg

AAA = ASCII adjust for add

0 0 1 1 0 1 1 1

DAA = Decimal adjust for add

0 0 1 0 0 1 1 1

Table 1-22 8086/88 Instruction Encoding (continued)

ARITHMETIC (Cont'd.)

SUB = Subtract:

Reg/memory and register to either
immediate from register/memory
immediate from accumulator

7 0 2 4 3 2 1 0	7 0 5 4 2 2 1 0	7 0 5 4 5 2 1 0	7 0 5 4 3 2 1 0	7 0 5 4 3 3 1 0	7 0 5 4 5 2 1 0
0 0 1 0 1 0 d w	mod r/m r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s = 0
0 0 1 0 1 0 w	data	data if w = 1			

SBB = Subtract with borrow:

Reg/memory and register to either
immediate from register/memory
immediate from accumulator

0 0 0 1 1 0 d w	mod r/m r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s = 0
0 0 0 1 1 0 w	data	data if w = 1			

DEC = Decrement:

Register/memory
Register
NEG Change sign

1 1 1 1 1 1 1 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)		
0 1 0 0 1	reg				
1 1 1 0 1 1 w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		

CMPS = Compare:

Register/memory and register
immediate with register/memory
immediate with accumulator
AAS ASCII adjust for subtract
DAS Decimal adjust for subtract

0 0 1 1 1 0 d w	mod r/m r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s = 0
0 0 1 1 1 0 w	data				
0 0 1 1 1 1 1					
0 0 1 0 1 1 1					

IMUL = Multiply (unsigned):

IMUL = Integer multiply (signed):

AAM = ASCII adjust for multiply:

DIV = Divide (unsigned):

IDIV = Integer divide (signed):

AAD = ASCII adjust for divide:

CBB = Convert byte to word:

CWD = Convert word to double word:

1 1 1 1 0 1 1 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)		
1 1 1 1 0 1 1 w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
1 1 1 1 0 1 1 w	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)		
1 1 1 1 0 1 1 w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
1 0 0 1 1 0 0 0					
1 0 0 1 1 0 0 1					

LOGIC

NOT = Invert:

SHL/SAL = Shift logical/arithmetic left:

SHR = Shift logical right:

SAR = Shift arithmetic right:

ROL = Rotate left:

1 1 1 1 0 1 1 w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 0 v w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 0 v w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 0 v w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 0 v w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)		

Table 1-22 8086/88 Instruction Encoding (continued)

LOGIC (Cont'd.)

- ROR** Rotate right
- RCL** Rotate through carry flag left
- RCR** Rotate through carry flag right

7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
1 1 0 1 0 0 v w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 0 v w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 0 v w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		

AND = And:

- Reg/memory and register to either
- immediate to register/memory
- immediate to accumulator

0 0 1 0 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
0 0 1 0 0 1 0 w		data	data if w=1		

TEST = And function to flags no result:

- Register/memory and register
- immediate data and register/memory
- immediate data and accumulator

0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 1 1 1 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
1 0 1 0 1 0 0 w		data			

OR = Or:

- Reg/memory and register to either
- immediate to register/memory
- immediate to accumulator

0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
0 0 0 1 1 0 w		data	data if w=1		

XOR = Exclusive or:

- Reg/memory and register to either
- immediate to register/memory
- immediate to accumulator

0 0 1 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
0 0 1 1 0 1 0 w		data	(DISP-LO)	(DISP-HI)	data
0 0 1 1 0 1 0 w		data	data if w=1		

STRING MANIPULATION

- REP** = Repeat
- MOVS** = Move byte/word
- CMPS** = Compare byte/word
- SCAS** = Scan byte/word
- LDS** = Load byte/word to AL/AX
- STD** = Store byte/word from AL/AX

1 1 1 1 0 0 1 z
1 0 1 0 0 1 0 w
1 0 1 0 0 1 1 w
1 0 1 0 1 1 1 w
1 0 1 0 1 1 0 w
1 0 1 0 1 0 1 w

Table 1-22 8086/88 Instruction Encoding (continued)

CONTROL TRANSFER

CALL = Call:

	7 6 6 4 3 2 1 0	7 6 6 4 3 2 1 0	7 6 6 4 3 2 1 0	7 6 6 4 3 2 1 0	7 6 6 4 3 2 1 0	7 6 6 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	IP-INC-LO	IP-INC-HI			
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
Direct intersegment	1 0 0 1 1 0 1 0	IP-lo	IP-hi			
		CS-lo	CS-hi			
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		

JMP = Unconditional Jump:

Direct within segment	1 1 1 0 1 0 0 1	IP-INC-LO	IP-INC-HI		
Direct within segment-short	1 1 1 0 1 0 1 1	IP-INC3			
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)	
Direct intersegment	1 1 1 0 1 0 1 0	IP-lo	IP-hi		
		CS-lo	CS-hi		
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)	

RET = Return from CALL:

Within segment	1 1 0 0 0 0 1 1		
Within segment adding immediate to SP	1 1 0 0 0 0 1 0	data-lo	data-hi
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-lo	data-hi
JE/JZ = Jump on equal / zero	0 1 1 1 0 1 0 0	IP-INC3	
JL/JNGE = Jump on less / not greater or equal	0 1 1 1 1 1 0 0	IP-INC3	
JLE/JNG = Jump on less or equal / not greater	0 1 1 1 1 1 1 0	IP-INC3	
JB/JNAE = Jump on below / not above or equal	0 1 1 1 0 0 1 0	IP-INC3	
JBE/JNA = Jump on below or equal / not above	0 1 1 1 0 1 1 0	IP-INC3	
JP/JPE = Jump on parity / parity even	0 1 1 1 1 0 1 0	IP-INC3	
JO = Jump on overflow	0 1 1 1 0 0 0 0	IP-INC3	
JS = Jump on sign	0 1 1 1 1 0 0 0	IP-INC3	
JNE/JNZ = Jump on not equal / not zero	0 1 1 1 0 1 0 1	IP-INC3	
JNL/JGE = Jump on not less / greater or equal	0 1 1 1 1 1 0 1	IP-INC3	
JNLE/JG = Jump on not less or equal / greater	0 1 1 1 1 1 1 1	IP-INC3	
JNB/JAE = Jump on not below / above or equal	0 1 1 1 0 0 1 1	IP-INC3	
JNBE/JA = Jump on not below or equal / above	0 1 1 1 0 1 1 1	IP-INC3	
JNP/JPO = Jump on not par / par odd	0 1 1 1 1 0 1 1	IP-INC3	
JNO = Jump on not overflow	0 1 1 1 0 0 0 1	IP-INC3	

Table 1-22 8086/88 Instruction Encoding (continued)

CONTROL TRANSFER (Cont'd.)

RET = Return from CALL:

7 6 6 4 3 2 1 0 7 6 6 4 3 2 1 0 7 6 6 4 3 2 1 0 7 6 6 4 3 2 1 0 7 6 6 4 3 2 1 0 7 6 6 4 3 2 1 0

JNS = Jump on not sign

0 1 1 1 0 0 1	IP-INCS
---------------	---------

LOOP = Loop CX times

1 1 1 0 0 0 1 0	IP-INCS
-----------------	---------

LOOPZ/LOOPE = Loop while zero/equal

1 1 1 0 0 0 0 1	IP-INCS
-----------------	---------

LOOPNZ/LOOPNE = Loop while not zero/equal

1 1 1 0 0 0 0 0	IP-INCS
-----------------	---------

JCXZ = Jump on CX zero

1 1 1 0 0 0 1 1	IP-INCS
-----------------	---------

INT = Interrupt:

Type specified

1 1 0 0 1 1 0 1	DATA-8
-----------------	--------

Type 3

1 1 0 0 1 1 0 0	
-----------------	--

INTO = Interrupt on overflow

1 1 0 0 1 1 1 0	
-----------------	--

IRET = Interrupt return

1 1 0 0 1 1 1 1	
-----------------	--

PROCESSOR CONTROL

CLC = Clear carry

1 1 1 1 1 0 0 0	
-----------------	--

CMC = Complement carry

1 1 1 1 0 1 0 1	
-----------------	--

STC = Set carry

1 1 1 1 1 0 0 1	
-----------------	--

CLD = Clear direction

1 1 1 1 1 1 0 0	
-----------------	--

STD = Set direction

1 1 1 1 1 1 0 1	
-----------------	--

CLI = Clear interrupt

1 1 1 1 1 0 1 0	
-----------------	--

STI = Set interrupt

1 1 1 1 1 0 1 1	
-----------------	--

HLT = Halt

1 1 1 1 0 1 0 0	
-----------------	--

WAIT = Wait

1 0 0 1 1 0 1 1	
-----------------	--

ESC = Escape (to external device)

1 1 0 1 1 x x x	mod y y y / r	(DIBP-LO)	(DIBP-HI)
-----------------	---------------	-----------	-----------

LOCK = Bus lock prefix

1 1 1 1 0 0 0 0	
-----------------	--

REP/REPE/REPZ = Override prefix

0 0 1 rrr 1 1 0	
-----------------	--

Table 1-23 Machine Instruction Decoding Guide

1ST BYTE		2ND BYTE	BYTES 3, 4, 5, 6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
00	0000 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG8/MEM8,REG8
01	0000 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG16/MEM16,REG16
02	0000 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG8,REG8/MEM8
03	0000 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG16,REG16/MEM16
04	0000 0100	DATA-8		ADD	AL,IMMED8
05	0000 0101	DATA-LO	DATA-HI	ADD	AX,IMMED16
06	0000 0110			PUSH	ES
07	0000 0111			POP	ES

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
08	0000 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG8/MEM8,REG8
09	0000 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16/MEM16,REG16
0A	0000 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG8,REG8/MEM8
0B	0000 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16,REG16/MEM16
0C	0000 1100	DATA-8		OR	AL,IMMED8
0D	0000 1101	DATA-LO	DATA-HI	OR	AX,IMMED16
0E	0000 1110			PUSH	CS
0F	0000 1111			(not used)	
10	0001 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8/MEM8,REG8
11	0001 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16/MEM16,REG16
12	0001 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8,REG8/MEM8
13	0001 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16,REG16/MEM16
14	0001 0100	DATA-8		ADC	AL,IMMED8
15	0001 0101	DATA-LO	DATA-HI	ADC	AX,IMMED16
16	0001 0110			PUSH	SS
17	0001 0111			POP	SS
18	0001 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8/MEM8,REG8
19	0001 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16/MEM16,REG16
1A	0001 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8,REG8/MEM8
1B	0001 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16,REG16/MEM16
1C	0001 1100	DATA-8		SBB	AL,IMMED8
1D	0001 1101	DATA-LO	DATA-HI	SBB	AX,IMMED16
1E	0001 1110			PUSH	DS
1F	0001 1111			POP	DS
20	0010 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8/MEM8,REG8
21	0010 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16/MEM16,REG16
22	0010 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8,REG8/MEM8
23	0010 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16,REG16/MEM16
24	0010 0100	DATA-8		AND	AL,IMMED8
25	0010 0101	DATA-LO	DATA-HI	AND	AX,IMMED16
26	0010 0110			ES:	(segment override prefix)
27	0010 0111			DAA	
28	0010 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8/MEM8,REG8
29	0010 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG16/MEM16,REG16
2A	0010 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8,REG8/MEM8
2B	0010 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG16,REG16/MEM16
2C	0010 1100	DATA-8		SUB	AL,IMMED8
2D	0010 1101	DATA-LO	DATA-HI	SUB	AX,IMMED16
2E	0010 1110			CS:	(segment override prefix)
2F	0010 1111			DAS	
30	0011 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8/MEM8,REG8
31	0011 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16/MEM16,REG16
32	0011 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8,REG8/MEM8
33	0011 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16,REG16/MEM16
34	0011 0100	DATA-8		XOR	AL,IMMED8
35	0011 0101	DATA-LO	DATA-HI	XOR	AX,IMMED16
36	0011 0110			SS:	(segment override prefix)

Table 1-23 Machine Instruction Decoding Guide (continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
37	0011 0110			AAA
38	0011 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP REG8/MEM8,REG8
39	0011 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP REG16/MEM16,REG16
3A	0011 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP REG8,REG8/MEM8
3B	0011 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP REG16,REG16/MEM16
3C	0011 1100	DATA-8		CMP AL,IMMED8
3D	0011 1101	DATA-LO	DATA-HI	CMP AX,IMMED16
3E	0011 1110			DS: (segment override prefix)
3F	0011 1111			AAS
40	0100 0000			INC AX
41	0100 0001			INC CX
42	0100 0010			INC DX
43	0100 0011			INC BX
44	0100 0100			INC SP
45	0100 0101			INC BP
46	0100 0110			INC SI
47	0100 0111			INC DI
48	0100 1000			DEC AX
49	0100 1001			DEC CX
4A	0100 1010			DEC DX
4B	0100 1011			DEC BX
4C	0100 1100			DEC SP
4D	0100 1101			DEC BP
4E	0100 1110			DEC SI
4F	0100 1111			DEC DI
50	0101 0000			PUSH AX
51	0101 0001			PUSH CX
52	0101 0010			PUSH DX
53	0101 0011			PUSH BX
54	0101 0100			PUSH SP
55	0101 0101			PUSH BP
56	0101 0110			PUSH SI
57	0101 0111			PUSH DI
58	0101 1000			POP AX
59	0101 1001			POP CX
5A	0101 1010			POP DX
5B	0101 1011			POP BX
5C	0101 1100			POP SP
5D	0101 1101			POP BP
5E	0101 1110			POP SI
5F	0101 1111			POP DI
60	0110 0000			(not used)
61	0110 0001			(not used)
62	0110 0010			(not used)
63	0110 0011			(not used)
64	0110 0100			(not used)
65	0110 0101			(not used)
66	0110 0110			(not used)
67	0110 0111			(not used)

Table 1-23 Machine Instruction Decoding Guide (continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
68	0110 1000			(not used)
69	0110 1001			(not used)
6A	0110 1010			(not used)
6B	0110 1011			(not used)
6C	0110 1100			(not used)
6D	0110 1101			(not used)
6E	0110 1110			(not used)
6F	0110 1111			(not used)
70	0111 0000	IP-INC8		JO SHORT-LABEL
71	0111 0001	IP-INC8		JNO SHORT-LABEL
72	0111 0010	IP-INC8		JB/JNAE/ SHORT-LABEL
				JC
73	0111 0011	IP-INC8		JNB/JAE/ SHORT-LABEL
				JNC
74	0111 0100	IP-INC8		JE/JZ SHORT-LABEL
75	0111 0101	IP-INC8		JNE/JNZ SHORT-LABEL
76	0111 0110	IP-INC8		JBE/JNA SHORT-LABEL
77	0111 0111	IP-INC8		JNBE/JA SHORT-LABEL
78	0111 1000	IP-INC8		JS SHORT-LABEL
79	0111 1001	IP-INC8		JNS SHORT-LABEL
7A	0111 1010	IP-INC8		JP/JPE SHORT-LABEL
7B	0111 1011	IP-INC8		JNP/JPO SHORT-LABEL
7C	0111 1100	IP-INC8		JL/JNGE SHORT-LABEL
7D	0111 1101	IP-INC8		JNL/JGE SHORT-LABEL
7E	0111 1110	IP-INC8		JLE/JNG SHORT-LABEL
7F	0111 1111	IP-INC8		JNLE/JG SHORT-LABEL
80	1000 0000	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	ADD REG8/MEM8,IMMED8
80	1000 0000	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-8	OR REG8/MEM8,IMMED8
80	1000 0000	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC REG8/MEM8,IMMED8
80	1000 0000	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB REG8/MEM8,IMMED8
80	1000 0000	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-8	AND REG8/MEM8,IMMED8
80	1000 0000	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB REG8/MEM8,IMMED8
80	1000 0000	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-8	XOR REG8/MEM8,IMMED8
80	1000 0000	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP REG8/MEM8,IMMED8
81	1000 0001	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADD REG16/MEM16,IMMED16
81	1000 0001	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	OR REG16/MEM16,IMMED16
81	1000 0001	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADC REG16/MEM16,IMMED16
81	1000 0001	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SBB REG16/MEM16,IMMED16

Table 1-23 Machine Instruction Decoding Guide (continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-36 INSTRUCTION FORMAT	
HEX	BINARY				
81	1000 0001	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	AND	REG16/MEM16,IMMED16
81	1000 0001	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SUB	REG16/MEM16,IMMED16
81	1000 0001	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	XOR	REG16/MEM16,IMMED16
81	1000 0001	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	CMP	REG16/MEM16,IMMED16
82	1000 0010	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	ADD	REG8/MEM8,IMMED8
82	1000 0010	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-8	(not used)	
82	1000 0010	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC	REG8/MEM8,IMMED8
82	1000 0010	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB	REG8/MEM8,IMMED8
82	1000 0010	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-8	(not used)	
82	1000 0010	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB	REG8/MEM8,IMMED8
82	1000 0010	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-8	(not used)	
82	1000 0010	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP	REG8/MEM8,IMMED8
83	1000 0011	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-SX	ADD	REG16/MEM16,IMMED8
83	1000 0011	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-SX	(not used)	
83	1000 0011	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-SX	ADC	REG16/MEM16,IMMED8
83	1000 0011	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-SX	SBB	REG16/MEM16,IMMED8
83	1000 0011	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-SX	(not used)	
83	1000 0011	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-SX	SUB	REG16/MEM16,IMMED8
83	1000 0011	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-SX	(not used)	
83	1000 0011	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-SX	CMP	REG16/MEM16,IMMED8
84	1000 0100	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG8/MEM8,REG8
85	1000 0101	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG16/MEM16,REG16
86	1000 0110	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG8,REG8/MEM8
87	1000 0111	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG16,REG16/MEM16
88	1000 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8/MEM8,REG8
89	1000 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16/MEM16/REG16
8A	1000 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8,REG8/MEM8
8B	1000 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16,REG16/MEM16
8C	1000 1100	MOD 0SR R/M	(DISP-LO),(DISP-HI)	MOV	REG16/MEM16,SEGREG
8C	1000 1100	MOD 1- R/M	(DISP-LO),(DISP-HI)	(not used)	
8D	1000 1101	MOD REG R/M	(DISP-LO),(DISP-HI)	LEA	REG16,MEM16
8E	1000 1110	MOD 0SR R/M	(DISP-LO),(DISP-HI)	MOV	SEGREG,REG16/MEM16
8E	1000 1110	MOD 1- R/M	(DISP-LO),(DISP-HI)	(not used)	
8F	1000 1111	MOD 000 R/M	(DISP-LO),(DISP-HI)	POP	REG16/MEM16
8F	1000 1111	MOD 001 R/M	(DISP-LO),(DISP-HI)	(not used)	
8F	1000 1111	MOD 010 R/M	(DISP-LO),(DISP-HI)	(not used)	

8086/8088 CPU

Table 1-23 Machine Instruction Decoding Guide (continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
8F	1000 1111	MOD 011 R/M		(not used)
8F	1000 1111	MOD 100 R/M		(not used)
8F	1000 1111	MOD 101 R/M		(not used)
8F	1000 1111	MOD 110 R/M		(not used)
8F	1000 1111	MOD 111 R/M		(not used)
90	1001 0000			NOP (exchange AX,AX)
91	1001 0001			XCHG AX,CX
92	1001 0010			XCHG AX,DX
93	1001 0011			XCHG AX,BX
94	1001 0100			XCHG AX,SP
95	1001 0101			XCHG AX,BP
96	1001 0110			XCHG AX,SI
97	1001 0111			XCHG AX,DI
98	1001 1000			CBW
99	1001 1001			CWD
9A	1001 1010	DISP-LO	DISP-HI,SEG-LO, SEG-HI	CALL FAR_PROC
9B	1001 1011			WAIT
9C	1001 1100			PUSHF
9D	1001 1101			POPF
9E	1001 1110			SAHF
9F	1001 1111			LAHF
A0	1010 0000	ADDR-LO	ADDR-HI	MOV AL,MEM8
A1	1010 0001	ADDR-LO	ADDR-HI	MOV AX,MEM16
A2	1010 0010	ADDR-LO	ADDR-HI	MOV MEM8,AL
A3	1010 0011	ADDR-LO	ADDR-HI	MOV MEM16,AL
A4	1010 0100			MOVS DEST-STR8, SRC-STR8
A5	1010 0101			MOVS DEST-STR16, SRC-STR16
A6	1010 0110			CMPS DEST-STR8, SRC-STR8
A7	1010 0111			CMPS DEST-STR16, SRC-STR16
A8	1010 1000	DATA-8		TEST AL,IMMED8
A9	1010 1001	DATA-LO	DATA-HI	TEST AX,IMMED16
AA	1010 1010			STOS DEST-STR8
AB	1010 1011			STOS DEST-STR16
AC	1010 1100			LDS SRC-STR8
AD	1010 1101			LDS SRC-STR16
AE	1010 1110			SCAS DEST-STR8
AF	1010 1111			SCAS DEST-STR16
B0	1011 0000	DATA-8		MOV AL,IMMED8
B1	1011 0001	DATA-8		MOV CL,IMMED8
B2	1011 0010	DATA-8		MOV DL,IMMED8
B3	1011 0011	DATA-8		MOV BL,IMMED8
B4	1011 0100	DATA-8		MOV AH,IMMED8
B5	1011 0101	DATA-8		MOV CH,IMMED8
B6	1011 0110	DATA-8		MOV DH,IMMED8
B7	1011 0111	DATA-8		MOV BH,IMMED8
B8	1011 1000	DATA-LO	DATA-HI	MOV AX,IMMED16
B9	1011 1001	DATA-LO	DATA-HI	MOV CX,IMMED16
BA	1011 1010	DATA-LO	DATA-HI	MOV DX,IMMED16
BB	1011 1011	DATA-LO	DATA-HI	MOV BX,IMMED16

8086/8088 CPU

Table 1-23 Machine Instruction Decoding Guide (continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
BC	1011 1100	DATA-LO	DATA-HI	MOV SP,IMMED16
BD	1011 1101	DATA-LO	DATA-HI	MOV BP,IMMED16
BE	1011 1110	DATA-LO	DATA-HI	MOV SI,IMMED16
BF	1011 1111	DATA-LO	DATA-HI	MOV DI,IMMED16
C0	1100 0000			(not used)
C1	1100 0001			(not used)
C2	1100 0010	DATA-LO	DATA-HI	RET IMMED16 (intra-seg)
C3	1100 0011			RET (intra-segment)
C4	1100 0100	MOD REG R/M	(DISP-LO),(DISP-HI)	LES REG16, MEM16
C5	1100 0101	MOD REG R/M	(DISP-LO),(DISP-HI)	LDS REG16, MEM16
C6	1100 0110	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	MOV MEM8, IMMED8
C6	1100 0110	MOD 001 R/M		(not used)
C6	1100 0110	MOD 010 R/M		(not used)
C6	1100 0110	MOD 011 R/M		(not used)
C6	1100 0110	MOD 100 R/M		(not used)
C6	1100 0110	MOD 101 R/M		(not used)
C6	1100 0110	MOD 110 R/M		(not used)
C6	1100 0110	MOD 111 R/M		(not used)
C7	1100 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO, DATA-HI	MOV MEM16, IMMED16
C7	1100 0111	MOD 001 R/M		(not used)
C7	1100 0111	MOD 010 R/M		(not used)
C7	1100 0111	MOD 011 R/M		(not used)
C7	1100 0111	MOD 100 R/M		(not used)
C7	1100 0111	MOD 101 R/M		(not used)
C7	1100 0111	MOD 110 R/M		(not used)
C7	1100 0111	MOD 111 R/M		(not used)
C8	1100 1000			(not used)
C9	1100 1001			(not used)
CA	1100 1010	DATA-LO	DATA-HI	RET IMMED16 (inter-segment)
CB	1100 1011			RET (inter-segment)
CC	1100 1100			INT 3
CD	1100 1101	DATA-8		INT IMMED8
CE	1100 1110			INTO
CF	1100 1111			IRET
D0	1101 0000	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL REG8/MEM8,1
D0	1101 0000	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR REG8/MEM8,1
D0	1101 0000	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL REG8/MEM8,1
D0	1101 0000	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR REG8/MEM8,1
D0	1101 0000	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL REG8/MEM8,1
D0	1101 0000	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR REG8/MEM8,1
D0	1101 0000	MOD 110 R/M		(not used)
D0	1101 0000	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR REG8/MEM8,1
D1	1101 0001	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL REG16/MEM16,1
D1	1101 0001	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR REG16/MEM16,1
D1	1101 0001	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL REG16/MEM16,1
D1	1101 0001	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR REG16/MEM16,1
D1	1101 0001	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL REG16/MEM16,1

8086/8088 CPU

Table 1-23 Machine Instruction Decoding Guide (continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
D1	1101 0001	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR REG16/MEM16,1 (not used)
D1	1101 0001	MOD 110 R/M	(DISP-LO),(DISP-HI)	SAR REG16/MEM16,1
D1	1101 0001	MOD 111 R/M	(DISP-LO),(DISP-HI)	ROL REG8/MEM8,CL
D2	1101 0010	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROR REG8/MEM8,CL
D2	1101 0010	MOD 001 R/M	(DISP-LO),(DISP-HI)	RCL REG8/MEM8,CL
D2	1101 0010	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL REG8/MEM8,CL
D2	1101 0010	MOD 011 R/M	(DISP-LO),(DISP-HI)	ROR REG8/MEM8,CL
D2	1101 0010	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL REG8/MEM8,CL
D2	1101 0010	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR REG8/MEM8,CL
D2	1101 0010	MOD 110 R/M	(DISP-LO),(DISP-HI)	(not used)
D2	1101 0010	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR REG8/MEM8,CL
D3	1101 0011	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL REG16/MEM16,CL
D3	1101 0011	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR REG16/MEM16,CL
D3	1101 0011	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL REG16/MEM16,CL
D3	1101 0011	MOD 011 R/M	(DISP-LO),(DISP-HI)	ROR REG16/MEM16,CL
D3	1101 0011	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL REG16/MEM16,CL
D3	1101 0011	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR REG16/MEM16,CL
D3	1101 0011	MOD 110 R/M	(DISP-LO),(DISP-HI)	(not used)
D3	1101 0011	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR REG16/MEM16,CL
D4	1101 0100	00001010		AAM
D5	1101 0101	00001010		AAD
D6	1101 0110			(not used)
D7	1101 0111			XLAT SOURCE-TABLE
D8	1101 1000	MOD 000 R/M	(DISP-LO),(DISP-HI)	ESC OPCODE.SOURCE
		1XXX MOD YYY R/M		
DF	1101 1111	MOD 111 R/M		
E0	1110 0000	IP-INC-8		LOOPNE/ SHORT-LABEL LOOPNZ
E1	1110 0001	IP-INC-8		LOOPE/ SHORT-LABEL LOOPZ
E2	1110 0010	IP-INC-8		LOOP SHORT-LABEL
E3	1110 0011	IP-INC-8		JCXZ SHORT-LABEL
E4	1110 0100	DATA-8		IN AL,IMMED8
E5	1110 0101	DATA-8		IN AX,IMMED8
E6	1110 0110	DATA-8		OUT AL,IMMED8
E7	1110 0111	DATA-8		OUT AX,IMMED8
E8	1110 1000	IP-INC-LO	IP-INC-HI	CALL NEAR-PROC
E9	1110 1001	IP-INC-LO	IP-INC-HI	JMP NEAR-LABEL
EA	1110 1010	IP-LO	IP-HI,CS-LO,CS-HI	JMP FAR-LABEL
EB	1110 1011	IP-INC8		JMP SHORT-LABEL
EC	1110 1100			IN AL,DX
ED	1110 1101			IN AX,DX
EE	1110 1110			OUT AL,DX
EF	1110 1111			OUT AX,DX
F0	1111 0000			LOCK (prefix)
F1	1111 0001			(not used)
F2	1111 0010			REPNE/REPZ
F3	1111 0011			REP/REPE/REPZ
F4	1111 0100			HLT
F5	1111 0101			CMC

Table 1-23 Machine Instruction Decoding Guide (continued)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
F6	1111 0110	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	TEST	REG8/MEM8,IMMED8
F6	1111 0110	MOD 001 R/M		(not used)	
F6	1111 0110	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT	REG8/MEM8
F6	1111 0110	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG	REG8/MEM8
F6	1111 0110	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL	REG8/MEM8
F6	1111 0110	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL	REG8/MEM8
F6	1111 0110	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV	REG8/MEM8
F6	1111 0110	MOD 111 R/M	(DISP-LO),(DISP-HI)	IDIV	REG8/MEM8
F7	1111 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	TEST	REG16/MEM16,IMMED16
F7	1111 0111	MOD 001 R/M		(not used)	
F7	1111 0111	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT	REG16/MEM16
F7	1111 0111	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG	REG16/MEM16
F7	1111 0111	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL	REG16/MEM16
F7	1111 0111	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL	REG16/MEM16
F7	1111 0111	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV	REG16/MEM16
F7	1111 0111	MOD 111 R/M	(DISP-LO),(DISP-HI)	IDIV	REG16/MEM16
F8	1111 1000			CLC	
F9	1111 1001			STC	
FA	1111 1010			CLI	
FB	1111 1011			STI	
FC	1111 1100			CLD	
FD	1111 1101			STD	
FE	1111 1110	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC	REG8/MEM8
FE	1111 1110	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC	REG8/MEM8
FE	1111 1110	MOD 010 R/M		(not used)	
FE	1111 1110	MOD 011 R/M		(not used)	
FE	1111 1110	MOD 100 R/M		(not used)	
FE	1111 1110	MOD 101 R/M		(not used)	
FE	1111 1110	MOD 110 R/M		(not used)	
FE	1111 1110	MOD 111 R/M		(not used)	
FF	1111 1111	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC	MEM16
FF	1111 1111	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC	MEM16
FF	1111 1111	MOD 010 R/M	(DISP-LO),(DISP-HI)	CALL	REG16/MEM16 (intra)
FF	1111 1111	MOD 011 R/M	(DISP-LO),(DISP-HI)	CALL	MEM16 (intersegment)
FF	1111 1111	MOD 100 R/M	(DISP-LO),(DISP-HI)	JMP	REG16/MEM16 (intra)
FF	1111 1111	MOD 101 R/M	(DISP-LO),(DISP-HI)	JMP	MEM16 (intersegment)
FF	1111 1111	MOD 110 R/M	(DISP-LO),(DISP-HI)	PUSH	MEM16
FF	1111 1111	MOD 111 R/M		(not used)	

Table 1-21 Key to Machine Instruction Encoding and Decoding

IDENTIFIER	EXPLANATION
MOD	Mode field; described in this chapter.
REG	Register field; described in this chapter.
R/M	Register/Memory field; described in this chapter.
SR	Segment register code: 00=ES, 01=CS, 10=SS, 11=DS.
W, S, D, V, Z	Single-bit instruction fields; described in this chapter.
DATA-8	8-bit immediate constant.
DATA-SX	8-bit immediate value that is automatically sign-extended to 16-bits before use.
DATA-LO	Low-order byte of 16-bit immediate constant.
DATA-HI	High-order byte of 16-bit immediate constant.
(DISP-LO)	Low-order byte of optional 8- or 16-bit unsigned displacement; <u>MOD</u> indicates if present.
(DISP-HI)	High-order byte of optional 16-bit unsigned displacement; <u>MOD</u> indicates if present.
IP-LO	Low-order byte of new IP value.
IP-HI	High-order byte of new IP value.
CS-LO	Low-order byte of new CS value.
CS-HI	High-order byte of new CS value.
IP-INC8	8-bit signed increment to instruction pointer.
IP-INC-LO	Low-order byte of signed 16-bit instruction pointer increment.
IP-INC-HI	High-order byte of signed 16-bit instruction pointer increment.
ADDR-LO	Low-order byte of direct address (offset) of memory operand; EA not calculated.
ADDR-HI	High-order byte of direct address (offset) of memory operand; EA not calculated.
--	Bits may contain any value.
XXX	First 3 bits of ESC opcode.
YYY	Second 3 bits of ESC opcode.
REG8	8-bit general register operand.
REG16	16-bit general register operand.
MEM8	8-bit memory operand (any addressing mode).
MEM16	16-bit memory operand (any addressing mode).
IMMED8	8-bit immediate operand.
IMMED16	16-bit immediate operand.
SEGREG	Segment register operand.
DEST-STR8	Byte string addressed by DI.

Table 1-21 Key to Machine Instruction Encoding and Decoding (continued)

IDENTIFIER	EXPLANATION
SRC-STR8	Byte string addressed by SI.
DEST-STR16	Word string addressed by DI.
SRC-STR16	Word string addressed by SI.
SHORT-LABEL	Label within ± 127 bytes of instruction.
NEAR-PROC	Procedure in current code segment.
FAR-PROC	Procedure in another code segment.
NEAR-LABEL	Label in current code segment but farther than -128 to $+127$ bytes from instruction.
FAR-LABEL	Label in another code segment.
SOURCE-TABLE	XLAT translation table addressed by BX.
OPCODE	ESC opcode operand.
SOURCE	ESC register or memory operand.

APPENDIX B

Algorithms for the Longest Common Subsequence Problem

DANIEL S. HIRSCHBERG

Princeton University, Princeton, New Jersey

ABSTRACT. Two algorithms are presented that solve the longest common subsequence problem. The first algorithm is applicable in the general case and requires $O(pn + n \log n)$ time where p is the length of the longest common subsequence. The second algorithm requires time bounded by $O(p(m + 1 - p) \log n)$. In the common special case where p is close to m , this algorithm takes much less time than n^2 .

KEY WORDS AND PHRASES: subsequence, common subsequence, algorithm

CR CATEGORIES: 3.73, 3.79, 5.25, 5.39

Introduction

We start by defining conventions and terminology that will be used throughout this paper.

String $C = c_1 c_2 \dots c_p$ is a *subsequence* of string $A = a_1 a_2 \dots a_m$ if there is a mapping $F: \{1, 2, \dots, p\} \rightarrow \{1, 2, \dots, m\}$ such that $F(i) = k$ only if $c_i = a_k$ and F is a monotone strictly increasing function (i.e. $F(i) = u$, $F(j) = v$, and $i < j$ imply that $u < v$). C can be formed by deleting $m - p$ (not necessarily adjacent) symbols from A . For example, "course" is a subsequence of "computer science."

String C is a *common subsequence* of strings A and B if C is a subsequence of A and also a subsequence of B .

String C is a *longest common subsequence* (abbreviated LCS) of string A and B if C is a common subsequence of A and B of maximal length, i.e. there is no common subsequence of A and B that has greater length.

Throughout this paper, we assume that A and B are strings of lengths m and n , $m \leq n$, that have an LCS C of (unknown) length p .

We assume that the symbols that may appear in these strings come from some alphabet of size t . A symbol can be stored in memory by using $\log t$ bits, which we assume will fit in one word of memory. Symbols can be compared ($a \leq b?$) in one time unit.

The number of different symbols that actually appear in string B is defined to be s (which must be less than n and t).

The longest common subsequence problem has been solved by using a recursion relationship on the length of the solution [7, 12, 16, 21]. These are generally applicable algorithms that take $O(mn)$ time for any input strings of lengths m and n even though the lower bound on time of $O(mn)$ need not apply to all inputs [2]. We present algorithms that, depending on the nature of the input, may not require quadratic time to recover an LCS. The first algorithm is applicable in the general case and requires $O(pn + n \log n)$ time. The second algorithm requires time bounded by $O((m + 1 - p)p \log n)$. In the common special case where p is close to m , this algorithm takes time

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This research was supported by a National Science Foundation graduate fellowship and by the National Science Foundation under Grant GJ-35570.

Author's present address: Department of Electrical Engineering, Rice University, Houston, TX 77001.

much less than n^2 . We conclude with references to other algorithms for the LCS problem that may be of interest.

pn Algorithm

We present in this section algorithm *ALGD*, which will find an LCS in time $O(pn + n \log n)$ where p is the length of the LCS. Thus this algorithm may be preferred for applications where the expected length of an LCS is small relative to the lengths of the input strings.

Some preliminary definitions are as follows:

We represent the concatenation of strings X and Y by $X\|Y$.

A_i represents the string $a_1 a_2 \dots a_i$ (elements 1 through i of string A). Similarly, the prefix of length j of string B is represented by B_j .

We define $L(i, j)$ to be the length of the LCS of prefixes of lengths i and j of strings A and B , i.e. the length of the LCS of A_i and B_j .

$\langle i, j \rangle$ represents the positions of a_i and b_j , the i th element of string A and the j th element of string B . We refer to i (j) as the i -value (j -value) of $\langle i, j \rangle$.

We define $\langle 0, 0 \rangle$ to be the set of 0-candidates, and we define $\langle i, j \rangle$ to be a k -candidate (for $k \geq 1$) if $a_i = b_j$ and there exist i' and j' such that $i' < i, j' < j$, and $\langle i', j' \rangle$ is a $(k - 1)$ -candidate. We say that $\langle i', j' \rangle$ generates $\langle i, j \rangle$.

Define $a_0 = b_0 = \$$ where $\$$ is some symbol that does not appear in strings A or B .

LEMMA 1. For $k \geq 1$, $\langle i, j \rangle$ is a k -candidate iff $L(i, j) \geq k$ and $a_i = b_j$. Thus there is a common subsequence of length k of A_i and B_j .

PROOF. By induction on k . $\langle i, j \rangle$ is a 1-candidate iff $a_i = b_j$ (by definition), in which case $L(i, j)$ necessarily is at least 1. Thus the lemma is true for $k = 1$. Assume it is true for $k - 1$. Consider k . If $\langle i, j \rangle$ is a k -candidate then there exist $i' < i$ and $j' < j$ such that $\langle i', j' \rangle$ is a $(k - 1)$ -candidate. By assumption, there is a common subsequence $D' = d_1 d_2 \dots d_{k-1}$ of $A_{i'}$ and $B_{j'}$. Since $a_i = b_j$ ($\langle i, j \rangle$ is a k -candidate), $D = D' \| a_i$ is a common subsequence of length k of A_i and B_j . Thus $L(i, j) \geq k$.

Conversely, if $L(i, j) \geq k$ and $a_i = b_j$, then there exist $i' < i$ and $j' < j$ such that $a_{i'} = b_{j'}$ and $L(i', j') = L(i, j) - 1 \geq k - 1$. $\langle i', j' \rangle$ is a $(k - 1)$ -candidate (by inductive hypothesis) and thus $\langle i, j \rangle$ is a k -candidate. \square

The length of an LCS is p , the maximum value of k such that there exists a k -candidate. As we shall see, to recover an LCS, it suffices to maintain the sequence of a 0-candidate, 1-candidate, ..., $(p - 1)$ -candidate, and a p -candidate such that in this sequence each i -candidate can generate the $(i + 1)$ -candidate for $0 \leq i < p$.

Rule. Let $x = \langle x_1, x_2 \rangle$ and $y = \langle y_1, y_2 \rangle$ be two k -candidates. If $x_1 \geq y_1$ and $x_2 \geq y_2$, then we say that y rules out x (x is a superfluous k -candidate) since any $(k + 1)$ -candidate that could be generated by x can also be generated by y . Thus, from the set of k -candidates, we need consider only those that are minimal under the usual vector ordering. Note that if x and y are minimal elements then $x_1 < y_1$ iff $x_2 > y_2$.

LEMMA 2. Let the set of k -candidates be $\{\langle i_r, j_r \rangle\}$ ($r = 1, 2, \dots$). We can rule out candidates so that (after renumbering) $i_1 < i_2 < \dots$ and $j_1 > j_2 > \dots$.

PROOF. Any two k -candidates $\langle i, j \rangle$ and $\langle i', j' \rangle$ satisfy one of the following (without loss of generality, $i \leq i'$):

- (1) $i < i', j \leq j'$.
- (2) $i < i', j > j'$.
- (3) $i = i', j \leq j'$.
- (4) $i = i', j > j'$.

In cases (1) and (3) $\langle i', j' \rangle$ can be ruled out; in case (4) $\langle i, j \rangle$ can be ruled out; and case (2) satisfies the statement of the lemma. Thus any set of k -candidates which cannot be reduced by further application of the rule will satisfy the condition stated in the lemma. \square

The set of k -candidates, reduced by application of the rule so as to satisfy the statement of Lemma 2, are the minimal elements of the set of k -candidates (since no

element can rule out a minimal element) and will be called the set of *minimal k-candidates*. By Lemma 2, there is at most one minimal *k*-candidate for each *i*-value.

We note that if (i, j) is a minimal *k*-candidate then $L(i, j) = k$ and (i, j) is the *k*-candidate with *i*-value *i* having smallest *j*-value *j* such that $L(i, j) = k$.

LEMMA 3. For $k \geq 1$, (i, j) is a minimal *k*-candidate iff *j* is the minimum value such that $b_j = a_i$ and $low < j < high$, where *high* is the minimum *j*-value of all *k*-candidates whose *i*-value is less than *i* (no upper limit if there are no such *k*-candidates) and *low* is the minimum *j*-value of all $(k - 1)$ -candidates whose *i*-value is less than *i*.

PROOF. Assume that (i, j) is a minimal *k*-candidate. If $j \geq high$ then there is a *k*-candidate (i', j') such that $i' < i$ and $j' = high \leq j$. (i, j) would be ruled out by (i', j') and thus would not be minimal.

If $j \leq low$, then there is no $(k - 1)$ -candidate that can generate (i, j) . (i, j) would not be a *k*-candidate.

$b_j = a_i$ is required by the definition of *k*-candidate and $low < j < high$ has just been shown. If *j* and *j'* both satisfy these constraints, $j < j'$, then (i, j') is ruled out by (i, j) . Thus, for a particular *i*, *j* must be the minimum *j*-value of all *k*-candidates satisfying these constraints.

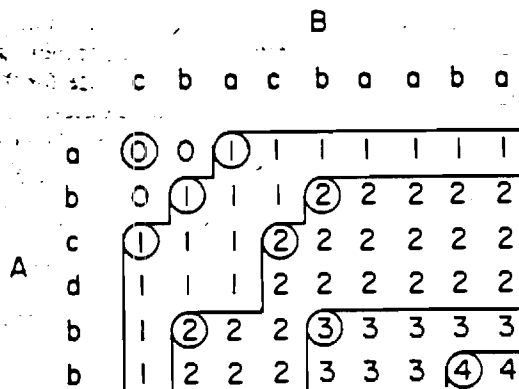
The *if* of the lemma has thus been shown.

The converse is easily shown: If (i, j) is not a *k*-candidate, then either $a_i \neq b_j$ or there is no $(k - 1)$ -candidate that can generate (i, j) . That is, the *j*-value of all $(k - 1)$ -candidates with *i*-value less than *i* is greater than or equal to *j*. This is equivalent to $j \leq low$.

If (i, j) is a *k*-candidate but is not minimal, say (i', j') rules out (i, j) , then $i' \leq i$ and $j' \leq j$. If $i' < i$, then clearly $j < high$ is violated. Otherwise, $i' = i$. In this case $j' > low$ since (i', j') must be generated from a $(k - 1)$ -candidate and $b_{j'} = a_i$ since (i', j) is a *k*-candidate. Also $j' < j < high$. Thus j' satisfies all the constraints and *j* is not the minimum value that does so, a contradiction. \square

We present algorithm *ALGD*, which, using the results of Lemma 3, obtains an LCS *C* of length *p* of input strings *A* and *B* in time $O(pn + n \log n)$.

The algorithm is based on an efficient representation of the *L* matrix. Since *L* is nondecreasing in both arguments, we may draw contours in its matrix as shown in the following example:



The entire matrix is specified by its contours. The contours are described by sets of minimal *k*-candidates. The contour between *L*-values of $k - 1$ and *k* is defined by the set of minimal *k*-candidates whose elements are positioned at the convex corners of the contour.

To keep track of the minimal *k*-candidates, we use the matrix *D*. $D[k, i]$ is the *j*-value of the unique minimal *k*-candidate having *i*-value of *i* or 0 if there is no such minimal *k*-candidate. Thus *D* describes the contours by giving the number of the first column of row *i* that is in region *k* (if that number is different from $D[k, i - 1]$).

$lowcheck$ is the smallest i -value of a $(k - 1)$ -candidate. $FLAG$ has value 1 iff there are any k -candidates.

$NB[\theta]$ is the number of times symbol θ occurs in string B . $PB[\theta, 1], \dots, PB[\theta, NB[\theta]]$ is the ordered list, smallest first, of positions in B in which symbol θ occurs.

If t , the size of the symbol alphabet, is not large compared to n , then we may index an array by the bit representation of a symbol. Otherwise, if $t \gg n$, then we construct a balanced binary search tree which provides a mapping from symbols that appear in string B to the integers 1 through s (there are s different symbols that appear in B). Whenever string element a_i appears as an array subscript (as in $N[a_i]$), it should be understood that we are indexing N by the integer s_i which has been obtained (during initialization for $ALGD$) from traversing the search tree just described. If a_i does not appear in B , then the integer s_i is zero. An equivalent assumption is followed for subscript b_j in step 1.

$ALGD(m, n, A, B, C, p)$

```

1.  $NB[\theta] \leftarrow 0$  for  $\theta = 1, \dots, s$ 
    $PB[\theta, 0] \leftarrow 0$  for  $\theta = 1, \dots, s$ 
    $PB[0, 0] \leftarrow 0$ ;  $PB[0, 1] \leftarrow 0$ 
   for  $j \leftarrow 1$  step 1 until  $n$  do
   begin
      $NB[b_j] \leftarrow NB[b_j] + 1$ 
      $PB[b_j, NB[b_j]] \leftarrow j$ 
   end
2.  $D[0, i] \leftarrow 0$  for  $i = 0, \dots, m$ 
    $lowcheck \leftarrow 0$ 
3. for  $k \leftarrow 1$  step 1 do
   begin
4.  $N[\theta] \leftarrow NB[\theta]$  for  $\theta = 1, \dots, s$ 
      $N[0] \leftarrow 1$ 
      $FLAG \leftarrow 0$ 
      $low \leftarrow D[k - 1, lowcheck]$ 
      $high \leftarrow n + 1$ 
5. for  $i \leftarrow lowcheck + 1$  step 1 until  $m$  do
   begin
6.   while  $PB[a_i, N[a_i] - 1] > low$  do  $N[a_i] \leftarrow N[a_i] - 1$ 
7.   if  $high > PB[a_i, N[a_i]] > low$ 
     then begin
        $high \leftarrow PB[a_i, N[a_i]]$ 
        $D[k, i] \leftarrow high$ 
       if  $FLAG = 0$  then ( $lowcheck \leftarrow i$ ;  $FLAG \leftarrow 1$ )
     end
     else  $D[k, i] \leftarrow 0$ 
8.   if  $D[k - 1, i] > 0$  then  $low \leftarrow D[k - 1, i]$ 
   end loop of step 5
9. if  $FLAG = 0$  then go to step 10
   end loop of step 3
10.  $p \leftarrow k - 1$ 
      $k \leftarrow p$ 
     for  $i \leftarrow m + 1$  step -1 until 0 do
       if  $D[k, i] > 0$  then
         begin
            $c_k \leftarrow a_i$ 
            $k \leftarrow k - 1$ 
         end
   end

```

The loop of step 3 evaluates the set of minimal k -candidates for $k = 1, 2, \dots$. The loop of step 5 evaluates the set of minimal k -candidates, smallest i -value first, and fills in the D array accordingly (in the example given previously this is left-to-right) while scanning the chains of occurrences of a given character in B with largest j -value first (right-to-left). For each i , i can be the i -value of a minimal k -candidate if there is a j satisfying the constraints of Lemma 3. This is tested by determining the minimum j -value of symbol a_i that is greater than low . If that value is less than $high$, then (i, j) is a minimal k -candidate.

Steps 7 and 8 are done in constant time. Total time is $O(pm)$. Step 9 is done in constant time. Total time is $O(p)$. Step 10 is done in time $O(m)$. Total execution time is thus as stated above. \square

Note that for $p \geq O(\log s)$, *ALGD* requires time $O(pn)$.

pe log n Algorithm

We now consider a special case that often occurs in applications such as determining the discrepancies between two files, one of which was obtained by making minor alterations to the other (and we wish to recover those alterations). We assume that there is an LCS of length at least $m - \epsilon$ (for some given ϵ).

If C is an LCS of A and B , there will be at most ϵ elements of A that do not appear in C . The position of each such element will be called a *skipped position*. Thus there are at most ϵ skipped positions. We define e to be $\epsilon + 1$.

If (i, j) is a minimal k -candidate that can be an element in an LCS (that is, $a_i = b_j$ is the k th element of an LCS), then $k \leq i \leq k + \epsilon$ (otherwise more than ϵ positions in A would be skipped). We shall call such candidates *feasible k -candidates*. Let $h = i - k$. Then $0 \leq h \leq \epsilon$ and h is the number of positions in A that have been skipped thus far (through a_{k+h}). By Lemma 2, there is at most one feasible k -candidate with i -value of i .

Let the feasible k -candidate pairs (i -value and j -value) be held in arrays F and G , e.g. $(h + k, j)$ would be described by $F[h] = h + k$, $G[h] = j$. If there is no feasible k -candidate with i -value $h + k$, let $F[h] = F[h - 1]$, $G[h] = G[h - 1]$, and define $F[-1] = 0$, $G[-1] = n + 1$. By this construction and by Lemma 2, F is a nondecreasing sequence and G is a nonincreasing sequence.

Define $NEXTB(\theta, j)$ to be the minimum $r > j$ such that $b_r = \theta$. If there is no such r , then $NEXTB(\theta, j)$ is defined to be $n + 1$.

LEMMA 6. If (i, j) is a feasible k -candidate, then $j = NEXTB(a_i, G[h])$, where $h = i - k$ and where $G[h]$ is the value associated with the set of feasible $(k - 1)$ -candidates.

PROOF. Let (i, j) be a feasible k -candidate. By definition of k -candidate, there must exist $i' < i$ and $j' < j$ such that (i', j') is a feasible $(k - 1)$ -candidate. By Lemma 3, j is the minimum (over possible j') of $NEXTB(a_i, j')$. But $i' < i$ implies that $NEXTB(\theta, j') \leq NEXTB(\theta, j)$. Therefore $j = NEXTB(a_i, \text{min possible } j')$. Since j -values of minimal k -candidates decrease as their i -values increase, the minimum possible j' is the j -value of the feasible $(k - 1)$ -candidate whose i -value is as large as possible but less than $i = h + k$, i.e. not more than $h + (k - 1)$. $G[h]$ is precisely that j -value. So we conclude that $j = NEXTB(a_i, G[h])$. \square

In order to be able to recover an LCS, we shall keep track (for each feasible k -candidate) of which h positions in A have been skipped. A straightforward method, keeping values of $F[h]$ for all h and k , requires space of $O(pk)$. We shall use a data structure that requires only $O(e^2 + n)$ space without changing the order of magnitude of time requirements.

Let there be an array *KEEP* whose elements are triples such that

$$KEEP[x] = \langle aa[x], nskip[x], pt[x] \rangle.$$

P is an array of size e such that, after the set of feasible k -candidates has been determined, $x = F[h]$ will be the index of the element of *KEEP* that has information enabling recovery of a common subsequence that has $a_{F[h]} = b_{G[h]}$ as its k th element. $F[h] = h + k$, and thus precisely h of the elements $a_1, \dots, a_{F[h]}$ will not appear in the common subsequence. To recover the common subsequence, it is sufficient to recover these h skipped positions. If $x = 0$, then no positions were skipped, and if $x < 0$, then there is no common subsequence to be recovered.

The method of recovery is as follows:

If x is zero, there are no more skipped positions to be recovered.

Otherwise, $aa[x]$ is the largest index of a skipped position in string A . $nskip[x]$ is the number of consecutive positions ending in $aa[x]$, all of which are skipped positions.

If all of the skipped A -positions have been recovered, then $pr[x]$ is zero.

Otherwise, $pr[x]$ is the index of $KEEP$ that has information enabling recovery of the skipped A -positions having indices smaller than $aa[x] - nskip[x] + 1$.

Example. If positions 2, 5, 6, 7, 9, 10 in string A correspond to a common subsequence of length 6 (of $A_{1..10}$), then $h = 4$ and $KEEP[P[4]]$ will enable recovery of positions 1, 3, 4, 8: $aa[P[4]] = 8$, $nskip[P[4]] = 1$, $pr[P[4]] = y$ (another index of $KEEP$). $aa[y] = 4$, $nskip[y] = 2$ (positions 3 and 4 have been skipped), $pr[y] = z$. $aa[z] = 1$, $nskip[z] = 1$, $pr[z] = 0$ (all skipped positions have been recovered).

Reference counts are kept for each element of $KEEP$. Spaces in the $KEEP$ array are maintained by garbage collection functions $GETSPACE$ which provides an available space and $PUTSPACE$ which places a newly available space (i.e. one whose reference count drops to zero) on the garbage linked list. See [10] for implementation techniques.

We now present $ALGE$, which uses Lemma 6 in order to solve the LCS problem in time $O(pe \log n)$:

$ALGE(m, n, A, B, C, p, \epsilon)$

1. $F[h], G[h] \leftarrow 0$ for $h = 0, \dots, \epsilon$
 $P[0] \leftarrow 0; P[h] \leftarrow -1$ for $h = 1, \dots, \epsilon$
2. for $k \leftarrow 1$ step 1 while there were candidates found in the last pass do
begin
3. $imax \leftarrow 0$
 $jmin \leftarrow n + 1$
4. for $h \leftarrow 0$ step 1 until ϵ do
begin
5. $i \leftarrow h + k$
 $j \leftarrow NEXTB(a_i, G[h])$
if $j \geq jmin$
6. then begin
 $F[h] \leftarrow imax$
 $G[h] \leftarrow jmin$
 $NEWP[h] \leftarrow -1$
end
7. else begin
 $nskip \leftarrow (i - 1) - F[h]$
if $nskip = 0$
then $NEWP[h] \leftarrow P[h]$
else begin
 $NEWP[h] \leftarrow GETSPACE$
 $KEEP[NEWP[h]] \leftarrow (i - 1, nskip, P[h - nskip])$
end
8. $imax \leftarrow i$
 $jmin \leftarrow j$
 $F[h] \leftarrow i$
 $G[h] \leftarrow j$
end
9. end loop of step 4
10. if no k -candidates were found then goto step 13
for $i \leftarrow 0$ step 1 until ϵ do
begin
11. $REMOVE(P[i])$
 $P[i] \leftarrow NEWP[i]$
end loop of step 10
12. end loop of step 2
13. $x \leftarrow \min h$ such that $P[h] \geq 0, -1$ if none such
 $p \leftarrow k - 1$
if $x < 0$ OR $p < m - \epsilon$ then (print "NO"; goto step 15)
14. RECOVER
15. END of $ALGE$

SUBROUTINE RECOVER

1. $SKIP[x + 1] \leftarrow 0$
 $lastmatch \leftarrow F[x]$
 $y \leftarrow P[x]$

```

2. while  $y \neq \epsilon$  do
  begin
     $count \leftarrow nskip[y]$ 
     $position \leftarrow aa[y]$ 
3. while  $count > 0$  do
  begin
     $SKIP[x] \leftarrow position$ 
     $x \leftarrow x - 1$ 
     $position \leftarrow position - 1$ 
     $count \leftarrow count - 1$ 
  end loop of step 3
   $y \leftarrow pt[y]$ 
end loop of step 2
4.  $x \leftarrow 1$ 
    $k \leftarrow 1$ 
   for  $i \leftarrow 1$  step 1 until  $lastmatch$  do
     if  $i = SKIP[x]$  then  $x \leftarrow x + 1$ 
     else begin
        $c_k \leftarrow a_i$ 
        $k \leftarrow k + 1$ 
     end
   end
5. END OF RECOVER

```

The loop of step 2 evaluates sets of feasible k -candidates for $k = 1, 2, \dots$. The loop of step 4 evaluates whether there is a feasible k -candidate having precisely h skipped positions, for $h = 0, 1, \dots, \epsilon$, by using Lemma 6 to determine the j -value for a particular i -value and then checking, by using Lemma 2, whether (i, j) is minimal. $imax$ is the maximum i -value of feasible k -candidates generated thus far (i.e. with i -values less than the current value of i); $jmin$ is the corresponding j -value (which is the minimum j -value of feasible k -candidates generated thus far). If (i, j) is a feasible k -candidate, then it is stored in the F and G arrays and information will be stored in $P[h]$ as well as the skipped positions occurring before $F[h]$ ($\langle F[h], G[h] \rangle$ is a $(k-1)$ -candidate that can generate (i, j)). The h skipped positions corresponding to $\langle F[h], G[h] \rangle$ are recoverable by accessing $KEEP[P[h]]$. In general there may be more than one feasible k -candidate that will be generated by $\langle F[h], G[h] \rangle$. Thus we must not destroy $P[h]$ until all required references to $KEEP[P[h]]$ are made. For this reason, new values for the P array are stored in the $NEWP$ array. When we no longer need the old values of P (after the inner loop of steps 4-9), we can then replace them with the new values, being careful to decrement reference counts of $KEEP$ elements that were pointed to by the old P array.

Function $REMOVE(x)$ decrements the reference count of $KEEP[x]$ (unless $x \leq 0$, in which case nothing is done), and, if $KEEP[x]$ now has reference count zero, then a call will be made to $REMOVE(pt[x])$ after $KEEP[x]$ has been put on the garbage linked list by using $PUTSPACE$.

Implementation of NEXTB

The following should be done before using $ALGE$:

- Sort the symbols in A and then construct a balanced binary search tree of symbols that appear in string A .
Let there be sr such symbols ($sr \leq m$).
- for $k \leftarrow 1$ step 1 until sr do $LAST[k] \leftarrow 0$
- for $i \leftarrow 1$ step 1 until n do


```

begin
  find out that  $b_i = \theta_k$ 
   $j \leftarrow LAST[k]$ 
   $LAST[k] \leftarrow i$ 
  if  $j \neq 0$  then  $NEXT[j] \leftarrow i$ 
  else  $FIRST[k] \leftarrow i$ 
end loop of step 3

```

```

4. start ← 1
   for k ← 1 step 1 until s do
     begin
       Place the positions  $j$  of  $B$  such that  $b_j = \theta_k$  into  $N[start]$  through  $N[start + nn - 1]$  where  $\theta_k$  occurs  $nn$ 
       times in string  $B$ . The first position in  $B$  at which  $\theta_k$  occurs is at  $FIRST[k]$ . If  $\theta_k$  occurs at position  $j$ , then
       the next occurrence of  $\theta_k$  in  $B$  will be at position  $NEXT[j]$  unless  $LAST[k] = j$ , in which case there are no
       more occurrences of  $\theta_k$  in  $B$ .
        $S[k] \leftarrow start$ 
        $start \leftarrow start + nn$ 
     end

```

We can find out that $a_i = \theta_k$ in time $O(\log s)$. $N[S[1]:S[k+1]-1]$ holds the block of positions j with $b_j = \theta_k$. This block of cells can be searched by using binary search of a linearly ordered array [11, Sec. 6.2.1]. $NEXT(a_i, j)$ can thus be executed in time $O(\log n)$.

If s is very small, then the following alternate way of computing $NEXTB(\theta, j)$ may be preferred: Instead of constructing a compressed array in step 4, construct a $NEXTB$ matrix while in step 3. For each i , set $NEXTB[k, i] = i$ for $j \leq i < i$. This will result in time and space complexity (of the setup) of $O(sn)$. The function $NEXTB(\theta, j)$ can be evaluated by determining that $\theta = \theta_k$ in time $O(\log S)$ and by doing a simple table look-up.

ALGE retains k -candidates, as did *ALGD*, except for those candidates that cannot lead to a sufficiently long common subsequence because too many A -positions have already been skipped. The $(k+1)$ -candidates that can be generated by the dropped k -candidates also skip too many A -positions.

LEMMA 7. *ALGE retains all feasible k -candidates.*

PROOF. By induction on k . It is trivially true for $k = 0$ (the F and G arrays are initialized to zero in step 1). Assume that the set of feasible $(k-1)$ -candidates has been evaluated and stored in arrays F and G . *ALGE* generates the set of feasible k -candidates in order of increasing i -value. $F[h]$ is to hold $i = h + k$ if i is an i -value of a feasible k -candidate; otherwise $F[h]$ is to hold the maximum $i' < i$ such that i' is a feasible k -candidate. $G[h]$ is to hold the corresponding j -value. $imax$ and $jmin$ hold the last-generated feasible k -candidate, which, by Lemma 2, has the maximum i -value and minimum j -value generated thus far. Step 3 initializes them to correctly indicate that no k -candidates have yet been generated. Step 5 evaluates the j -value for a given potential k -candidate by using Lemma 6. If $j \geq jmin$ then, even though the necessary condition for feasibility has been met, (i, j) is not minimal since it would be ruled out by $(imax, jmin)$. In this case step 6 sets $F[h]$ and $G[h]$ to $imax$ and $jmin$. If $j < jmin$, then (i, j) is minimal since it cannot be ruled out by any previously generated k -candidate ($j < jmin$) and it cannot be ruled out by any future generated k -candidate (all future $i' > i$). In this case step 8 sets $F[h]$ and $G[h]$ and also updates $imax$ and $jmin$. \square

THEOREM 3. *ALGE correctly computes the LCS of strings A and B if the LCS is of length at least $m - \epsilon$.*

PROOF. By Lemma 7, *ALGE* correctly keeps minimal k -candidates. Thus, if there is a common subsequence of length $p \geq m - \epsilon$, then there is a minimal p -candidate which will be feasible. The data structure of *ALGE* keeps track, for each feasible k -candidate (i, j) , of the $h = i - k$ positions in string A that have been skipped in the common subsequence of length k of $A_{1..h}$ and $B_{1..j}$. $P[h]$ points to the element of *KEEP* that contains the necessary information. P is updated in step 7 when a feasible k -candidate is generated. If any additional positions are skipped (between the k -candidate (i, j) and the $(k-1)$ -candidate (i', j') that generated (i, j)), then that information is recorded in an element of *KEEP* as well as a pointer, enabling recovery of the $h - nskip$ previously skipped A -positions (of (i', j')). Subroutine *RECOVER* recovers the skipped positions of a feasible p -candidate by reversing the process in which they were stored and then computes the LCS by deleting the skipped positions from string A . \square

THEOREM 4. *For $\epsilon \leq O(n^{1/2})$, *ALGE* requires space linear in n .*

PROOF. The *KEEP* array requires $O(\epsilon^2)$ space: The common subsequence implied by k -candidate $(h+k, j)$ has h skipped A -positions, $h \leq \epsilon$, and thus can use at most h spaces in the *KEEP* array. The total number of spaces referred to by all feasible k -candidates is thus at most $\epsilon(\epsilon+1)/2$. Adding to that the (exactly) ϵ references to get the set of feasible $(k+1)$ -candidates gives a total of no more than $(\epsilon^2 + \epsilon)/2$. Each element of array *KEEP* requires four words (*aa*, *nskip*, *pt*, and a reference counter).

The arrays and space that they use are as follows: $F[\epsilon]$, $G[\epsilon]$, $C[p]$, $P[\epsilon]$, $NEWP[\epsilon]$, $KEEP[2\epsilon^2 + 2\epsilon]$, $FIRST[ss]$, $NEXT[n]$, $LAST[ss]$, $SKIP[\epsilon]$, $S[ss]$, $N[n]$.

The *NEXTB* function requires at most $2n$ locations to store the various balanced binary search trees.

Thus a total of at most $2\epsilon^2 + 7\epsilon + 4n + p + 3ss$ locations is used. For $\epsilon \leq O(n^{1/2})$, space requirements are linear in n . \square

THEOREM 5. *ALGE* requires time $O(pe \log n)$.

PROOF. Preprocessing for the *NEXTB* function requires time $O(n \log m)$. Step 1 takes time $O(\epsilon)$. Step 2 executes steps 3–12 p times. Step 3 takes constant time for a total time of $O(p)$. Step 4 executes steps 5–9 at most ϵ times. Step 5 takes time $O(\log n)$ for a total time of $O(pe \log n)$. Steps 6–9 take constant time for a total time of $O(pe)$. Steps 10–12, excluding time spent in function *REMOVE*, take time $O(\epsilon)$ for a total time of $O(pe)$.

Subroutine *RECOVER* recovers at most ϵ skipped positions (taking time $O(\epsilon)$) and then deletes them from string A (taking time $O(m)$) for a total time of $O(m)$.

The number of references (to array *KEEP*) removed is at most the number of references inserted. There are at most pe references inserted (one per execution of step 7), and the amount of time (per reference removal) spent in function *REMOVE* is constant. Therefore the total time spent in function *REMOVE* is $O(pe)$.

Therefore the total time of execution of *ALGE* is $O(pe \log n)$. \square

It is noted that step 5, requiring $O(\log n)$ time, is the bottleneck, causing total time requirements of $O(pe \log n)$. P. van Emde Boas's recent algorithm for priority queues [19] appears capable of solving the position-finding problem in time $O(\log \log n)$. If so, this would reduce the time bound of this problem to $O(pe \log \log n)$.

ALGE assumes that ϵ is known. If ϵ is not known, then set $\epsilon \leftarrow 2$ and proceed through the algorithm. If that value of ϵ is insufficient (i.e. there is no common subsequence of length $m - \epsilon$), then double the guess for ϵ and continue iteratively until a common subsequence is found.

Total time spent will be (letting k be the multiplicative coefficient of the time requirement)

$$2pk \log n + 4pk \log n + \dots + epk \log n,$$

which is less than $2pek \log n$. Since $\epsilon < 2(m+1-p)$, we can recover an LCS in time $O(p(m+1-p) \log n)$.

Other Algorithms

The only known algorithm for the LCS problem with worst-case behavior less than quadratic is due to Paterson [14]. The algorithm has complexity $O(n^2 \log \log n / \log n)$. It uses a "Four Russians" approach (see [3] or [1, pp. 244–247]). Essentially, instead of matrix L (where $L[i, j]$ is the length of an LCS of A_{1i} and B_{1j}) being calculated one element at a time (see [7]), the matrix is broken up into boxes of some appropriate size k . The *high* sides of a box (the $2k-1$ elements of L on the edges of the box with largest indices) are computed from L -values known for boxes adjacent to it on the *low* side and from the relevant symbols of A and B by using a look-up table which was precomputed.

The algorithm assumes a fixed alphabet size although modifications to the algorithm may be able to get around that condition.

There are $2k + 1$ elements of L adjacent to a box on the low side. Two adjacent L -elements can differ by either zero or one. There are thus 2^{2k} possibilities in this respect. The A - and B -values range over an alphabet of size s for each of $2k$ elements, yielding a multiplicative factor of s^{2k} , and the total number of boxes to be precomputed is therefore $2^{2k(1+\log s)}$. Each such box can be precomputed in time $O(k^2)$ for a total precomputing time of $O(k^2 2^{2k(1+\log s)})$.

There are $(n/k)^2$ boxes to be looked up, each of which will require $O(k \log k)$ time to be read, for a total time of $O(n^2 \log k/k)$.

The total execution time will therefore be $O(k^2 2^{2k(1+\log s)} + n^2 \log k/k)$. If we let $k = \log n/2(1 + \log s)$, we see that the total execution time will be $O(n^2 \log \log n/\log n)$.

Restrictions on the ECS Problem

Szymanski [17] shows that if we consider the LCS problem with the restriction that no symbol appears more than once within either input string, then this problem can be solved in time $O(n \log n)$.

In addition if one of the input strings is the string of integers $1 - n$, this problem is equivalent to finding the longest ascending subsequence in a string of distinct integers. If we assume that a comparison between integers can be done in unit time, this problem can be solved in time $O(n \log \log n)$ by using the techniques of van Emde Boas [18].

ACKNOWLEDGMENT. I would like to thank the (anonymous) referees for their many helpful suggestions which have led to a material improvement in the readability of this paper.

REFERENCES

(Note. References [4-6, 8, 9, 13, 15, 20, 22, 23] are not cited in the text.)

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. AHO, A.V., HIRSCHBERG, D.S., AND ULLMAN, J.D. Bounds on the complexity of the longest common subsequence problem. *J. ACM* 23, 1 (Jan. 1976), 1-12.
3. ABLAZAROV, V.L., BINIC, E.A., KRONROD, M.A., AND FARADZEV, I.A. On economic construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR* 194 (1970), 487-488 (in Russian). English transl. in *Soviet Math. Dokl. J.*, 5 (1970), 1209-1210.
4. CHVATAL, V., KLASBER, D.A., AND KNUTH, D.E. Selected combinatorial research problems. STAN-CS-72-292, Stanford U., Stanford, Calif., 1972, p. 26.
5. CHVATAL, V., AND SANKOFF, D. Longest common subsequences of two random sequences. STAN-CS-75-477, Stanford U., Stanford, Calif., Jan. 1975.
6. HIRSCHBERG, D.S. On finding maximal common subsequences. TR-156, Comptr. Sci. Lab., Princeton U., Princeton, N.J., Aug. 1974.
7. HIRSCHBERG, D.S. A linear space algorithm for computing maximal common subsequences. *Comm. ACM* 18, 6 (June 1975), 341-343.
8. HIRSCHBERG, D.S. The longest common subsequence problem. Ph.D. Th., Princeton U., Princeton, N.J., Aug. 1975.
9. HUNT, J.W., AND SZYMANSKI, T.G. A fast algorithm for computing longest common subsequences. *Comm. ACM* 20, 5 (May 1977), 350-353.
10. KNUTH, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., sec. ed., 1973.
11. KNUTH, D. E. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
12. LOWRANCE, R., AND WAGNER, R.A. An extension of the string-to-string correction problem. *J. ACM* 22, 2 (April 1975), 177-183.
13. NEEDLEMAN, S.B., AND WUNSCH, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biology* 48 (1970), 443-453.
14. PATERSON, M.S. Unpublished manuscript. U. of Warwick, Coventry, England, 1974.
15. SANKOFF, D. Matching sequences under deletion/insertion constraints. *Proc. Nat. Acad. Sci. USA* 69, 1 (Jan. 1974), 4-6.
16. SELLERS, P.H. An algorithm for the distance between two finite sequences. *J. Combinatorial Theory, Ser. A*, 16 (1974), 253-258.

A Fast Algorithm for Computing Longest Common Subsequences

James W. Hunt
Stanford University
Thomas G. Szymanski
Princeton University

Previously published algorithms for finding the longest common subsequence of two sequences of length n have had a best-case running time of $O(n^2)$. An algorithm for this problem is presented which has a running time of $O((r + n) \log n)$, where r is the total number of ordered pairs of positions at which the two sequences match. Thus in the worst case the algorithm has a running time of $O(n^2 \log n)$. However, for those applications where most positions of one sequence match relatively few positions in the other sequence, a running time of $O(n \log n)$ can be expected.

Key Words and Phrases: Longest common subsequence, efficient algorithms

CR Categories: 3.73, 3.63, 5.25

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

The work of the first author was partially supported by Bell Laboratories' Cooperative Research Fellowship Program. The work of the second author was partially supported by NSF Grants GJ-35570 and DCR74-21939.

Author's addresses: J.W. Hunt, Department of Electrical Engineering, Stanford University, Stanford CA 94305; T.G. Szymanski, Dept. of Electrical Engineering and Computer Science, Princeton University, Brackett Hall, Engineering Quadrangle, Princeton, NJ 98540.

Many algorithms [1, 4, 6] for finding the longest common subsequence of two sequences of length n have appeared in the literature. These algorithms all have a worst-case (as well as a best-case) running time of $O(n^2)$.¹

A more relevant parameter for this problem is r , the total number of matching pairs of positions within the sequences in question. We shall present an $O((r+n) \log n)$ algorithm for the longest common subsequence problem. In the worst case this is of course $O(n^2 \log n)$. However, for a large number of applications, we can expect r to be close to n . In these situations our algorithm will exhibit an $O(n \log n)$ behavior. Typical of such applications are the following:

- 1) Finding the longest ascending subsequence of a permutation of the integers from 1 to n [3].
- 2) Finding a maximum cardinality linearly ordered subset of some finite collection of vectors in 2-space [7].
- 3) Finding the edit distance between two files in which the individual lines of the files are considered to be atomic. The longest common subsequence of these files, considered as sequences, represents that common "core" which does not have to be changed if we desire to edit one file into the other.

Thus in the general case our algorithm will not take much longer than the algorithms of [1, 4, 6], whereas in many common applications, our algorithm will perform substantially better.

Let A be a finite sequence of elements chosen from some alphabet. We denote the length of A by $|A|$. $A[i]$ is the i th element of A and $A[i:j]$ denotes the subsequence $A[i], A[i+1], \dots, A[j]$.

If U and V are finite sequences, then U is said to be a subsequence of V if there exists a monotonically increasing sequence of integers $r_1, r_2, \dots, r_{|U|}$ such that $U[i] = V[r_i]$ for $1 \leq i \leq |U|$. U is a common subsequence of A and B if U is a subsequence of both A and B . A longest common subsequence is a common subsequence of greatest possible length.

Throughout this paper A and B will be used to denote the sequences in question. For ease of presentation, we shall assume both sequences have the same length which will be denoted by n . The number of elements in the set $\{(i, j) \text{ such that } A[i] = B[j]\}$ will be denoted by r .

Summary Results

The key data structure needed by our algorithm is an array of "threshold values" $T_{i,k}$ defined by $T_{i,k} = \text{smallest } j \text{ such that } A[1:i] \text{ and } B[1:j] \text{ contain a common subsequence of length } k$. For example, given sequences $A = abcbbda$, $B = badbabd$ we have $T_{5,1} =$

$1, T_{5,2} = 3, T_{5,3} = 6, T_{5,4} = 7, T_{5,5} = \text{undefined}$.

Each $T_{i,k}$ may thus be considered as a pointer which tells us how much of the B sequence is needed to produce a common subsequence of length k with the first i elements of A .

Note that each row of the T array is strictly increasing; that is,

LEMMA 1. If $T_{i,1}, T_{i,2}, \dots, T_{i,p}$ are defined, then $T_{i,1} < T_{i,2} < \dots < T_{i,p}$.

PROOF. Consider the common subsequence of length k contained in $A[1:i]$ and $B[1:T_{i,k}]$. Clearly $B[T_{i,k}]$ is the last member of this common subsequence or else $T_{i,k}$ would not be minimal. Therefore $A[1:i]$ and $B[1:T_{i,k}-1]$ contain a common subsequence of length $k-1$, that is, $T_{i,k-1} \leq T_{i,k}-1$. \square

This linear ordering is of paramount importance in the efficient implementation of our algorithm.

Suppose that we have computed $T_{i,k}$ for all values of k and wish to compute $T_{i+1,k}$ for all values of k . We first show $T_{i+1,k}$ must lie in a specific range of values.

LEMMA 2. $T_{i,k-1} < T_{i+1,k} \leq T_{i,k}$.

PROOF. If $A[1:i]$ and $B[1:T_{i,k}]$ have a common subsequence of length k , then certainly $A[1:i+1]$ and $B[1:T_{i,k}]$ do also. Thus $T_{i+1,k} \leq T_{i,k}$.

By definition, $A[1:i+1]$ and $B[1:T_{i+1,k}]$ have a common subsequence of length k . Deleting the last element from each of these sequences can remove at most one element from this common subsequence. Thus $A[1:i]$ and $B[1:T_{i+1,k}-1]$ have a common subsequence of length $k-1$. Accordingly $T_{i,k-1} \leq T_{i+1,k}-1$ and $T_{i,k-1} < T_{i+1,k}$. \square

The following rule suffices to compute $T_{i+1,k}$ from $T_{i,k-1}$ and $T_{i,k}$.

LEMMA 3.

$$T_{i+1,k} = \begin{cases} \text{smallest } j \text{ such that } A[i+1] = B[j] \\ \text{and } T_{i,k-1} < j \leq T_{i,k} \\ T_{i,k} \text{ if no such } j \text{ exists.} \end{cases}$$

PROOF.

Case 1. No such j exists. By the minimality of $T_{i+1,k}$, any common subsequence of the sequences $A[1:i+1]$ and $B[1:T_{i+1,k}]$ must have $B[T_{i+1,k}]$ as its last element. Moreover, by Lemma 2 and the premise of this case, $B[T_{i+1,k}]$ does not match $A[i+1]$. Therefore the same common subsequence of length k is also contained in $A[1:i]$ and $B[1:T_{i+1,k}]$. Thus $T_{i,k} \leq T_{i+1,k}$ and by Lemma 2, $T_{i,k}$ must equal $T_{i+1,k}$.

Case 2. There exists a minimal j for which $A[i+1] = B[j]$ and $T_{i,k-1} < j \leq T_{i,k}$. Certainly $A[1:i+1]$ and $B[1:j]$ contain a common subsequence of length k , namely the length $k-1$ common subsequence of

¹ An unpublished result of Michael Paterson shows how to construct an $O(n^2/\log n)$ algorithm for the longest common subsequence problem for sequences over a finite alphabet, and an $O((n^2 \log \log n)/\log n)$ algorithm for sequences over an infinite ordered alphabet. All results of this paper apply to the case of the infinite ordered alphabet.

$A[1:i]$ and $B[T_{i,k-1}]$ with the pair $A[i+1], B[j]$ "tacked" onto the end. Thus $T_{i+1,k} \leq j$.

Assume temporarily that $T_{i+1,k} < j$. Since Lemma 2 guarantees that $T_{i,k-1} < T_{i+1,k}$ we can conclude that the last element of the length k common subsequence of $A[1:i+1]$ and $B[1:T_{i+1,k}]$ does not match $A[i+1]$. Thus $A[1:i]$ and $B[1:T_{i+1,k}]$ also contain a common subsequence of length k which implies that $T_{i,k} \leq T_{i+1,k}$. By Lemma 2 then, $T_{i,k} = T_{i+1,k}$. However, by the above assumption and the premise of this case, $T_{i+1,k} < j \leq T_{i,k}$, implying that $T_{i,k} \neq T_{i+1,k}$. This contradiction leads us to conclude that the original assumption of $T_{i+1,k} < j$ is incorrect and hence we must have $T_{i+1,k} = j$. \square

We can now present an $O(n^2 \log n)$ algorithm for determining the length of the longest common subsequence. Subsequent refinements will enable us to not only improve the running time to $O((r+n) \log n)$ but also recover the actual longest common subsequence.

Algorithm 1

```

element array A[1:n], B[1:n];
integer array THRESH[0:n];
integer i, j, k;
THRESH[0] := 0;
for i := 1 step 1 until n do
  THRESH[i] := n + 1;
for i := 1 step 1 until n do
  for j := n step -1 until 1 do
    if A[i] = B[j] then
      begin
        find k such that THRESH[k-1] < j ≤ THRESH[k];
        THRESH[k] := j;
      end;
  print largest k such that THRESH[k] ≠ n + 1;

```

The correctness of the algorithm follows from consideration of the invariant relation " $THRESH[k] = T_{i-1,k}$ for all k " which holds at the start of each iteration on i , and the invariant relation " $THRESH[k] = T_{i,k}$ for all k " which holds at the end of each iteration on i .

Since the $THRESH$ array is monotonically increasing (Lemma 1) we can utilize a binary search to implement the "find" operation in time $O(\log n)$. Thus Algorithm 1 may be implemented to run in $O(n^2 \log n)$ time.

Finally, notice that the direction of the loop on j is crucial. Suppose that for some value of i , $A[i]$ matches several different B elements in a given "threshold" interval, say $B[j_1], \dots, B[j_m]$ with $THRESH[k-1] = T_{i-1,k-1} < j_1 < \dots < j_m \leq T_{i-1,k} = THRESH[k]$. From Lemma 3, we see that $T_{i,k} = j_1$ and that $THRESH[k]$ should be updated to this value. Since the inner loop of Algorithm 1 considers values of j in decreasing order, each of the values j_m, j_{m-1}, \dots, j_1 will cause $THRESH[k]$ to take on successively smaller values until it is set equal to the desired value of j_1 . If instead the loop on j ran upwards from 1 to n , then not only would $THRESH[k]$ be set to j_m , but $THRESH[k+1]$ would be set to j_2 , $THRESH[k+2]$

would be set to j_3 and so forth. Since these latter assignments are unwarranted, we see that the loop on j must run downwards.

The Algorithm

A small amount of preprocessing will vastly improve the performance of Algorithm 1. The main source of inefficiency in this algorithm is the inner loop on j in which we repeatedly search for elements of the B sequence which match $A[i]$. Linked list techniques obviate the need for this search.

For each position i we need a list of corresponding j positions such that $A[i] = B[j]$. These lists must be kept in decreasing order in j . All positions of the A sequence which contain the same element may be set up to use the same physical list of matching j 's; for the sequences $A = abcbdda$, $B = badbabd$ the desired lists are

```

MATCHLIST[1] = (5, 2)
MATCHLIST[2] = (6, 4, 1)
MATCHLIST[3] = ( )
MATCHLIST[4] = MATCHLIST[2]
MATCHLIST[5] = (7, 3)
MATCHLIST[6] = MATCHLIST[5]
MATCHLIST[7] = MATCHLIST[1].

```

We can now display our final algorithm.

Algorithm 2

```

element array A[1:n], B[1:n];
integer array THRESH[0:n];
list array MATCHLIST[1:n];
pointer array LINK[1:n];
pointer PTR;
comment Step 1: build linked lists;
for i := 1 step 1 until n do
  set MATCHLIST[i] := (j1, j2, ..., jp) such that
    j1 > j2 > ... > jp and A[i] = B[jq] for 1 ≤ q ≤ p;
comment Step 2: initialize the THRESH array;
THRESH[0] := 0;
for i := 1 step 1 until n do
  THRESH[i] := n + 1;
LINK[0] := null;
comment Step 3: compute successive THRESH values;
for i := 1 step 1 until n do
  for j on MATCHLIST[i] do
    begin
      find k such that THRESH[k-1] < j ≤ THRESH[k];
      if j < THRESH[k] then
        begin
          THRESH[k] := j;
          LINK[k] := newnode (i, j, LINK[k-1]);
        end;
    end;
comment Step 4: recover longest common subsequence in reverse order;
k := largest k such that THRESH[k] ≠ n + 1;
PTR := LINK[k];
while PTR ≠ null do
  begin
    print (i, j) pair pointed to by PTR;
    advance PTR;
  end;

```

The subroutine *newnode* invoked in step 3 is a subroutine which creates a list node whose fields contain the values of the arguments to *newnode*. These arguments are, respectively, an index of a position in the *A* sequence, an index of a position in the *B* sequence, and a pointer to some other list node. The value returned by *newnode* is a pointer to the list node just created.

THEOREM 1. *Algorithm 2 finds and prints a longest common subsequence of the sequences A and B in time $O((r + n) \log n)$ and space $O(r + n)$.*

PROOF. Step 1 can be implemented by sorting each sequence while keeping track of each element's original position. We may then merge the sorted sequences creating the *MATCHLISTS* as we go. This step takes a total of $O(n \log n)$ time and $O(n)$ space.

Step 2 clearly takes $O(n)$ time.

The two outer loops of step 3 should be considered as a single loop over all pairs (i, j) such that $A[i] = B[j]$ taken in order of decreasing j within increasing i . In other words, the outer loops of step 3 induce exactly r executions of the innermost statements of step 3. Since these innermost statements involve one binary search plus a few operations which require constant time, we conclude that the time requirement for step 3 is $O(n + r \log n)$.

In this step we also implement a simple backtracking device that will allow us to recover the longest common subsequence. We record each (i, j) pair which causes an element of the *THRESH* array to change value. Thus whenever *THRESH*[k] is defined, *LINK*[k] points to the head of a list of (i, j) pairs describing a common subsequence of length k . Since at most one list node is created per search, Step 3 will require the allocation of at most $O(r)$ list nodes.

In step 4 we recover the actual longest common subsequence. Clearly this takes at most $O(n)$ time. \square

We note that certain input sequences such as $A = "aabaabaab \dots"$ and $B = "ababab \dots"$ cause Algorithm 2 to use $O(r)$ space even if list nodes are reclaimed whenever they become inaccessible. See [4] for an algorithm which never uses more than $O(n)$ space nor less than $O(n^2)$ time.

A Final Note

The key operations in the implementation of our algorithm are the operations of inserting, deleting, and testing membership of elements in a set where all elements are restricted to the first n integers. Peter van Emde Boas has shown that each such operation can be performed in $O(\log \log n)$ time [2]. His data structure requires $O(n \log \log n)$ time for initialization. Although the necessary algorithms are quite complex, we can use them to present the following theoretical result.

THEOREM 2. (a) *Algorithm 2 can be implemented to have a running time of $O(r \log \log n + n \log n)$ over an infinite alphabet.* (b) *Algorithm 2 can be implemented to have a running time of $O((n + r) \log \log n)$ over a fixed finite alphabet.* (c) *The longest ascending subsequence of a permutation of the first n integers may be found in $O(n \log \log n)$ time.*

PROOF. The problem of part (c) is, of course, equivalent to finding the longest common subsequence of the given permutation and the sequence $1, 2, \dots, n$. All three parts of the theorem use basically the same algorithm although the implementation of some of the steps varies slightly. We shall present a common analysis.

In all three cases we require $O(n \log \log n)$ time to initialize van Emde Boas's data structures. Step 1 entails a sorting procedure to set up the *MATCHLISTS*. For the infinite alphabet case, this sort can be done in $O(n \log n)$ time. In the other two cases, we can use a distribution sort to create the *MATCHLISTS* in $O(n)$ time. Step 2 takes $O(n)$ time, step 3 takes $O(n + r \log \log n)$ time and step 4 takes $O(n)$ time. Finally, for the permutation case note that each integer appears exactly once in each sequence and thus we have $r = n$. \square

Acknowledgments. The authors are indebted to M. Douglas McIlroy who first suggested this problem to us. Harold Stone suggested a variant of the problem (described and solved in [5]) which led to the development of the present algorithm. Alfred V. Aho and Jeffrey D. Ullman provided us with several enlightening conversations including the particular example given following Theorem 1 which shows that our algorithm can require as much as $O(r)$ space. Peter van Emde Boas made several helpful comments on an early draft of this paper.

Received May 1975; revised January 1976

References

1. Chvatal, V., Klarner, D.A., and Knuth, D.E. Selected combinatorial research problems. STAN-CS-72-292, Dep. Comptr. Sci., Stanford U., Stanford, Calif., June 1972.
2. van Emde Boas, P. Preserving order in a forest in less than logarithmic time. 16th Annual Symp. on Foundations Comptr. Sci., Oct. 1975, pp. 75-84.
3. Fredman, M.L. On computing the length of longest increasing subsequences. *Discrete Mathematics* 11, 1 (Jan. 1975), 29-35.
4. Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences. *Comm. ACM* 18, 6 (June 1975), 341-343.
5. Szymanski, T.G. A special case of the maximal common subsequence problem. TR-170, Dep. Electrical Eng., Princeton U., Princeton, N.J., Jan. 1975.
6. Wagner, R.A. and Fischer, M.J. The string-to-string correction problem. *J. ACM* 21, 1 (Jan. 1975), 168-173.
7. Yao, A.C. and Yao, F.F. On computing the rank function for a set of vectors. UIUCDCS-R-75-699, Dep. Comptr. Sci., U. of Illinois at Urbana-Champaign, Urbana, Ill., Feb. 1975.

File Comparison Algorithms

by Tom Steppe

Several popular algorithms exist for comparing two files. All of these actually look first for matches rather than differences. After the matching process has been completed, the remainders of the files that are not included in the matches are then reported as differences. (See Figure 1, page 29.)

The algorithms differ greatly in their conceptualization of the problem, however. In this article, I examine several algorithms for comparing text files—specifically, source code files—using a line as the basic unit of comparison. The ideas and algorithms I present here, however, can be extended to other types of files and other units of comparison as well. I also present a new algorithm with some interesting properties.

Evaluating The Algorithms

Any file comparison algorithm should be evaluated according to several criteria:

- Is it efficient? Time efficiency (speed) and space efficiency (memory usage) are both practical considerations. Usually they are related to the lengths of the files being compared.
- Is it robust? No algorithm is flawless. For any given file comparison algorithm, it is always possible to concoct devious situations in which its performance appears less than perfect. The algorithm should, however, be able to produce reasonable difference reports for a variety of test cases.
- Can it let differences go undetected? No algorithm should allow a file difference to go undetected.

Tom Steppe, P.O. Box 2887, Ann Arbor, MI 48106. Tom designs and develops software written exclusively in C.

Determining which files are more equal than others

• Can it let matches go undetected? If an algorithm can overlook matching lines, it will report these lines as differences when they are not. If the file comparison is being performed to produce a delta file, this usually is not a major problem, even though each undetected match does increase the size of the delta file unnecessarily. If the differences are to be inspected visually, however, a report of false differences can be a serious drawback.

Say, for example, that you do not have a file comparison utility and so you have to compare two files by eye. This process is certainly tedious and prone to error, especially if some of the differences are subtle. If you now use a file comparison utility that is known to report false differences, you have to inspect the output by eye and decide which reported differences are true differences. The utility has not really done the job for you, it has only made your "by eye" inspection a smaller job that is still prone to error.

• Can it detect blocks of text that have been moved? Typically, if a block of text has been moved, it simply shows up in the report of differences as a large deletion of text at one location and a large insertion of text at another. Unfortunately, no differences within the moved block are highlighted.

When a file comparison is used to create a delta file, the ability to detect

moved blocks of text is probably desirable because it can lead to smaller delta files. But, when a file comparison is performed so that the differences can be inspected visually, the ability to detect moved blocks is not always as handy as it might seem to be. Trying to report the moved blocks is often difficult and can lead to complicated reports of the differences, especially when a large block of text is moved, a piece of that block is moved to another location, a piece of that piece is moved to still another location, and so on. Also, the difference report can sometimes be overburdened by uninteresting reports of small blocks (one-line and two-line blocks of text) being moved all over the place.

Only one algorithm discussed here can inherently detect moved blocks of text. The other algorithms, however, can be extended to do so, as follows. After applying the algorithm, replace each matching line in each file with a line that is guaranteed never to match. This leaves only the differences, which could contain moved blocks of text. Next, reapply the algorithm to the transformed files. Any match that is found in this pass will represent a moved block of text (see Figure 2, page 29). Continue this process iteratively until no new matches can be found. Of course, the cost of this iterative behavior is longer execution time.

These criteria help to provide a useful basis for surveying popular file comparison algorithms.

Popular Algorithms for Finding Matches

Scan Until Next Match

The "scan until next matching se-

quence" algorithm is probably the oldest method of file comparison. This algorithm starts at the tops of both files and matches as many lines as possible. When a difference is detected, the next M lines are scanned until at least N consecutive matching lines are found. If a sequence of N or more consecutive matching lines is found, the process begins again after the matching sequence. If such a sequence is not found, the process begins again M lines further down in the files. This process is repeated until the ends of the files are reached.

The values of M and N can be adjusted to affect the algorithm's performance. The value of M is used to control efficiency by restricting the number of lines that will be examined while searching for a sequence of matching lines. When an improper sequence of matching lines is discovered, the algorithm can be reapplied using a new value for N that is larger than the length of the improper sequence. In this way, the algorithm will overlook the undesirable sequence because it contains fewer than N matching lines, but as is always the case, the algorithm will also overlook any legitimate matching sequences that contain fewer than N lines (see Figure 3, page 30). Unfortunately, these matching lines are then reported as differences. All too often, this algorithm produces bad reports in common situations.

Although this algorithm is often highly time efficient, requires minimal memory, and frequently produces good difference reports, it does not take long to become frustrated with its shortcomings and inherent problems and begin looking for a better solution.

Longest Common Subsequence

Think of a file as representing a sequence of lines. A subsequence of those lines is defined simply as any sequence of lines that results from removing zero or more lines from the original sequence—for example, the longest subsequence of any sequence of lines is the sequence itself, with zero lines removed. Also, a sequence of zero lines would be a subsequence of any sequence because it could be created by removing all the lines from any sequence.

The "longest common subse-

quence" approach to file comparison takes the two files to be compared and finds the longest sequence of lines that is a subsequence of each of the files' lines—the longest common subsequence (see Figure 4, page 30). The details of the algorithm are not discussed here, but sources of such discussions are included in the bibliography. The Unix diff command is based on this algorithm.

This algorithm provides a simple, compact formalization of the file comparison problem and produces reasonable difference reports in a variety of test cases. The reports are quite acceptable whether the comparison is being used for visual inspection of the differences or for creating a delta file. In fact, among all the algorithms discussed here, it is probably safe to say that this one con-

```

file1      file2
A          A
B          BB
C          C
D          E
E          F

Difference report:
#####00001#####file1
#      2  B
# ----- changed to
#      2  BB
#####00001#####file2

#####00001#####file1
#      4  D
# -----
# deleted
#####00001#####file2

#####00001#####file1
# inserted
# -----
#      5  F
#####00001#####file2

```

Figure 1: File comparison algorithms actually look first for line matches and then report lines that are not included in the matches as differences. The differences are usually expressed as the changes, insertions, and deletions that can be applied to one file to make it identical to the other.

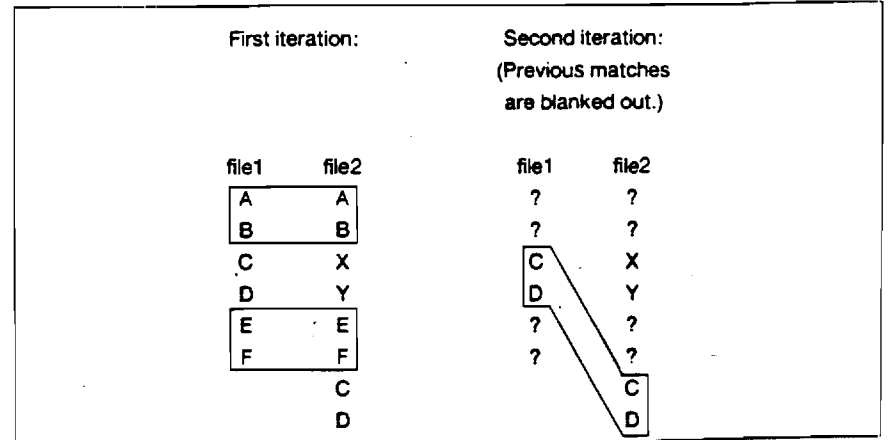


Figure 2: Moved blocks of text can be found by applying a standard line-matching algorithm to the files and then reapplying the algorithm iteratively to the remainders of both files.

FILE COMPARISONS
(continued from page 29)

sistently produces the best reports when comparing files that do not involve blocks of text that have been moved.

Sometimes the quality of the reports can be overshadowed by issues of time and space efficiency. This is not always true, but situations that include a poor combination of large files and limited computer resources can lead to less than desirable performance by this algorithm. A basic implementation of the algorithm requires linear space and quadratic time. In some cases, the quadratic time can prove to be unacceptable. In summary, the "longest common subsequence" algorithm produces excellent reports, but it can be slow.

Extended Unique Line Matching
The "extended unique line matching" algorithm is based on the idea that a line that occurs once and only once in each file must be the same line. These pairs of "unique" lines determine the initial set of matched

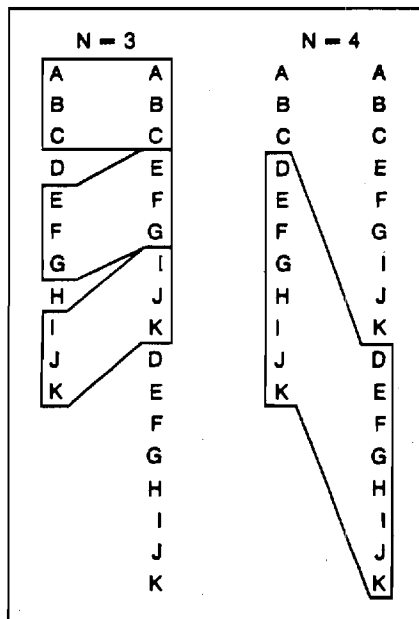


Figure 3: The "scan until next matching sequence" algorithm often produces bad reports in common situations. When $N=3$, the algorithm settles for matches of three lines, never realizing that a match of eight lines is possible. When $N=4$, it discovers the match of eight lines but does not detect the remaining match of three lines (A, B, C).

lines. (Imaginary lines at the tops and the bottoms of the files are also added to the set of matched lines.) Then, in each file, the lines adjacent to each match are examined and, if identical, are added to the set of matched lines. This process is repeated until no new matches can be found.

This algorithm has strong intuitive appeal. It is efficient, being linear in both time and space. Also, it is the only popular algorithm that inherently detects blocks of text that have been moved (even if some differences exist within the blocks). Moved blocks can be detected because the search for pairs of unique lines is in no way sequential and, therefore, can result in matches that indicate that a block of text has been moved. Note that the algorithm can find a moved block of text only if it contains a unique line match within it.

A significant problem with this algorithm is that it is prone to allowing some matches to go undetected. This occurs when matching lines are not neatly flanked by either unique line matches or the adjacent matches that have grown outward from unique line matches (see Figure 5, below).

This algorithm is fast and can frequently detect moved blocks of text, but a sacrifice is often made in the quality of the difference report. Probably its best application is in the generation of delta files when speed is the primary concern.

A New Algorithm

The "recursive longest matching sequence" algorithm uses a simple yet effective approach to the problem.

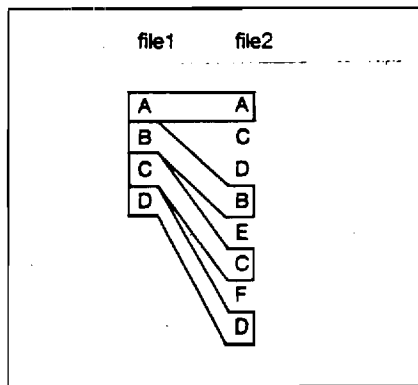


Figure 4: The "longest common sequence" algorithm finds the longest (not necessarily consecutive) sequence of lines that is contained in both files.

This method first scans both files from beginning to end, looking for the longest sequence of consecutive matching lines. That sequence is then thought of as dividing each of the two files into an upper section and a lower section. Then, the algorithm proceeds by scanning both upper sections looking for the longest sequence of consecutive matching lines and, similarly, both lower sections for the same. These matching sequences then divide their respective sections, and the process continues recursively until no more matches can be found.

This method of file comparison is easy to understand and produces acceptable difference reports across a spectrum of test cases. It uses linear space but quadratic time. Because time efficiency can be a problem in some situations, a simple modification of the algorithm is needed. An explanation of the modification requires an understanding of the method used to locate the longest sequence of matching lines between sections of two files.

First of all, once the longest sequence is known, it can be identified by a pair of starting lines—one line from each file that specifies where the sequence begins in that file. So, when searching for the longest sequence, candidate pairs of starting lines are examined successively (in some intelligent order that starts at the beginnings of both file sections), and information is continually maintained about the length and location of the longest sequence of matching lines that has been discovered so far.

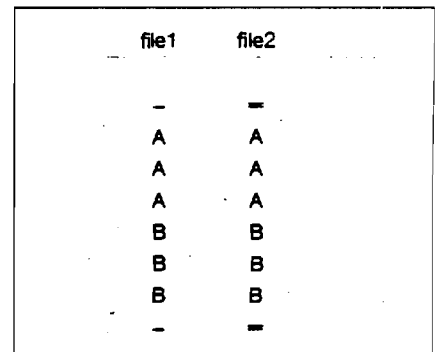


Figure 5: The "extended unique line matching" algorithm is prone to detecting false differences. In this case, no matches are found (because there are no unique line matches) and all lines are reported as differences.

FILE COMPARISONS
(continued from page 30)

When the ends of the file sections are reached, the longest sequence is known and information about the sequence is reported.

The modification to this algorithm allows the searching to stop if a sequence of N matching lines is found, realizing that it might not be the longest sequence that would be discovered if the searching were allowed to continue to the ends of the sections.

This allows the searching to end prematurely (before the longest sequence has been assured) and can save considerable time. N is called the "long-enough" value. The effects of the long-enough value can be examined by choosing some test pairs of files and comparing the behavior of the algorithm when a long-enough value is used and when one is not used. Quite often, the use of a reasonable long-enough value will find exactly the same sequences of matching lines (although the discoveries may occur in a different order), thus producing an identical report of the differences but with a significant improvement in speed (see Figure 6, page 32). In fact, the use of a reasonable long-enough value allows this algorithm to perform in essentially linear time for typical cases, overcoming the previous worry of time efficiency.

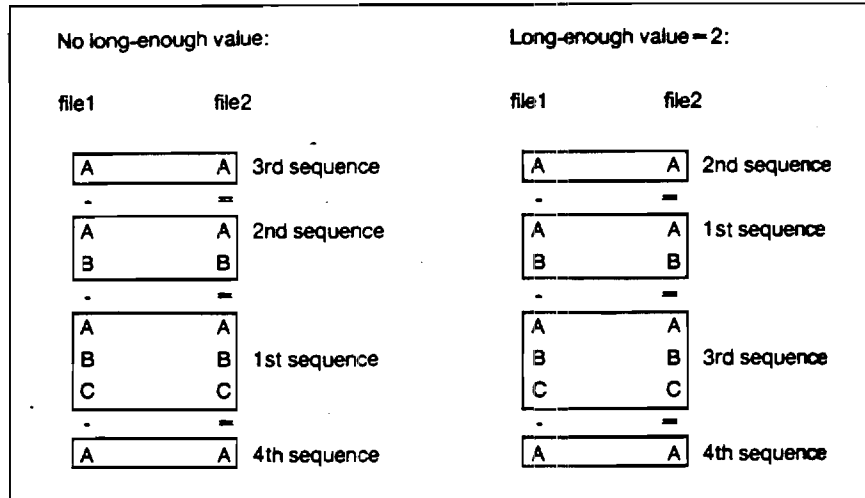


Figure 6: With the "recursive longest matching sequence" algorithm, the use of a long-enough value often finds exactly the same sequences of matching lines although the discoveries may occur in a different order.

The long-enough value is a parameter that you can specify. To determine a good value for your purposes, first guess at the length of the longest

Delta Files and User Reports

A file comparison utility is a versatile tool for a range of situations. It is useful to partition these situations into two distinct cases.

In the most common case, a file comparison is performed so that the differences between two versions of a text file can be inspected visually. The differences are usually expressed as the changes, insertions, and deletions that can be applied to one file to make it identical to the other file. In this case, the primary job of the comparison is to produce a concise and readable report of the differences.

In the course of editing, a file comparison can be used in this way to highlight the differences between a previous version of a file and the current version. Valid modifications can be verified, and spurious edits can be detected. As another example, if a new version of a program is produced, a partial test of its integrity could include a file comparison of its output with the output from a previous version of the program that is known to be correct. If the two out-

puts compare favorably, the new program passes this integrity test. If they do not compare favorably, another file comparison can be used in the debugging process to highlight the changes between a version of a source code file that is known to work and the version that does not work.

In the second case, a file comparison is performed to generate a delta file—a file that contains a report of the differences between the two files. If the file comparison is thought of as comparing an old file with a new file, a backward delta file is designed so that it contains all the information necessary to recreate the old file, given the new file. A forward delta file is designed to be able to recreate the new file, given the old file. In either case, one of the original files can be eliminated without loss of information. If the delta file is smaller than the file it allows to be eliminated, this will result in a savings of disk space. The primary job of a file comparison in this case is to produce a compact delta file.

This use of a file comparison utility is particularly common in version control systems that maintain multiple historical versions of source code files. Only the current version of a source code file is saved, whereas a backward delta file is saved for each historical version. Any historical version can be recreated by applying the appropriate delta files to the current version of the file. The savings in disk space can be tremendous. (Alternatively, some version control systems save the first version of the file and the subsequent forward delta files.)

This usage is also common in telecommunications applications where a file at one or more remote sites has to be updated from a host. A forward delta file is created on the host by comparing the new file with the old file (a copy of the file that exists at the remote site). If the delta file is small, it is often more efficient to transmit the forward delta file and apply it to the old file than it is to transmit the new file in its entirety.

FILE COMPARISONS (continued from page 32)

sequence of lines you can imagine appearing more than once in a typical file. The long-enough value should be at least one larger than your guess. This will help the algorithm to avoid matching the wrong instance when a sequence of lines appears multiple times in a file. If a particular choice of long-enough value produces unsatisfactory difference reports, the algorithm can always be applied again with a larger value. When comparing C source code, I typically choose a generous value of 25, and I rarely have to re-run the comparison.

The "recursive longest matching sequence" algorithm is particularly well suited to take advantage of some common hash code technology as a means of improving time performance even more. In applications that involve repetitive string comparisons, it is often useful to calculate hash codes initially for all the strings. Then, the hash codes are compared instead of the strings themselves. The comparison of two hash code values is much quicker than is the comparison of two strings. If the hash codes are not equal, the strings cannot possibly be the same and need not be compared. If the hash codes are equal, only then must the strings be compared to prove or disprove their equality.

The performance benefits are even more dramatic when hash codes are used with the "recursive longest matching sequence" algorithm. When searching for the longest sequence of matching lines, strings do not have to be compared every time a pair of matching hash codes is found. Instead, strings only have to be compared once a sequence of matching hash codes is found that is longer than the longest sequence yet found.

The time efficiency can be improved even further if a hash code table is maintained for each file. The table should consist of an array that contains as many elements as there are possible hash code values. Each element of the array should consist of a linked list of line numbers for lines whose hash code values are equal to the array index. This table

can easily be created by processing each line in the file, calculating its hash code value, and adding its line number to the proper linked list. Now, while searching for the longest sequence of matching lines by examining pairs of starting line numbers, the number of candidate pairs can be greatly reduced. For any given line in one file, only those lines in the other file that have the same hash code value (as can be easily determined from the file's hash code table) need to be considered.

A basic C implementation of the "recursive longest matching sequence" algorithm is shown in Listing One, page 54. Its simplicity, combined with a long-enough value modification and some clever use of hash codes, makes it a viable solution to the file comparison problem. It is suitable for both delta creation and visual inspection purposes.

Availability

All the source code for articles in this issue is available on a single disk. To order, send \$14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext.

216. Please specify issue number and format (MS-DOS, Macintosh, Kaypro).

You can also purchase a full-featured executable version of this algorithm from Stepping Stone Software, P.O. Box 2887, Ann Arbor, MI 48106 for \$30. The available format is MS-DOS 5¼-inch DSDD.

Bibliography

Heckel, Paul. "A Technique for Isolating Differences Between Files." *Communications of the ACM*, vol. 21, no. 4 (April 1978): 264-268.

Hirschberg, D. S. "A Linear Space Algorithm for Computing Maximal Common Subsequences." *Communications of the ACM*, vol. 18, no. 6 (June 1975): 341-343.

Wagner, Robert A.; and Fischer, Michael J. "The String-to-String Correction Problem." *Journal of the Association for Computing Machinery*, vol. 21, no. 1 (January 1974): 168-173.

DDJ

(Listing begins on page 54.)

Vote for your favorite feature/article.
Circle Reader Service No. 2.

Listing One (Text begins on page 28.)

```

/*
** Copyright (c) 1987, Tom Steppe. All rights reserved.
**
** This module compares two arrays of lines (representing
** files) and reports the sequences of consecutive matching
** lines between them using the "recursive longest matching
** sequence" algorithm. This is useful for implementing a
** file comparison utility.
**
** Compiler: Microsoft (R) C Compiler Version 4.00
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <malloc.h>

/* Boolean type and values. */
typedef int    BOOLEAN;
#define TRUE   1
#define FALSE  0

/* Minimum macro. */
#define min(x, y)  (((x) <= (y)) ? (x) : (y))

/* Value to indicate identical strings with strcmp. */
#define ALIKE    0

/* Result of hashing function for a line of text. */
typedef unsigned int    HASH;

/* Mask for number of bits in hash code. (12 bits). */
#define MASK    (unsigned int) 0x0FFF

/* Number of possible hash codes. */
#define HASHSIZ    (MASK + 1)

/* Information about an entry in a hash table. */
typedef struct tblentry
{
    int    frst;    /* First line # with this hash code. */
    int    last;    /* Last line # with this hash code. */
} TBLENTRY;

/* Information about a line of text. */
typedef struct lineinf
{
    HASH    hash;    /* Hash code value. */
    int    nxtln;    /* Next line with same hash (or 0). */
} LINEINF;

/* Information about a file. */
typedef struct fileinf
{
    char    **txt;    /* Array of lines of text. */
    LINEINF *line;    /* Array of line info structs. */
    TBLENTRY *hashtbl; /* Hash table. */
} FILEINF;

/* Function declarations. */
BOOLEAN filcmp    (char **, int, char **, int, int);
BOOLEAN get_inf   (char **, int, FILEINF *);
HASH    calc_hash (char *);
void    find_seq  (FILEINF *, int, int,
                  FILEINF *, int, int, int);
BOOLEAN chk_hashes (LINEINF *, LINEINF *, int);
int    cnt_matches (char **, char **, int);
void    rpt_seq    (int, int, int);

/*****
** compare compares two arrays of lines and reports the
** sequences of consecutive matching lines. The zeroth

```

```

** element of each array is unused so that the index into
** the array is identical to the associated line number.
**
** RETURNS: TRUE if comparison succeeded.
**          FALSE if not enough memory.
**
**
*****/

BOOLEAN compare (a1, n1, a2, n2, lngval)

char  **a1; /* (I) Array of lines of text in #1. */
int   n1;  /* (I) Number of lines in a1.
           (Does not count 0th element.) */
char  **a2; /* (I) Array of lines of text in #2. */
int   n2;  /* (I) Number of lines in a2.
           (Does not count 0th element.) */
int   lngval; /* (I) "Long enough" value. */
{
    FILEINF  f1; /* File information for #1. */
    FILEINF  f2; /* File information for #2. */
    BOOLEAN  rtn; /* Return value. */

    /* Gather information for each file, then compare. */
    if (rtn =
        (get_inf (a1, n1, &f1) && get_inf (a2, n2, &f2)))
    {
        fnd_seq (&f1, 1, n1, &f2, 1, n2, lngval);
    }

    return (rtn);
}

/*****
** get_inf calculates hash codes and builds a hash table.
**
** RETURNS: TRUE if get_inf succeeded.
**          FALSE if not enough memory.
**
**
*****/

static BOOLEAN get_inf (a, n, f)

char  **a; /* (I) Array of lines of text. */
int   n;  /* (I) Number of lines in a. */
FILEINF *f; /* (O) File information. */
{
    unsigned int  size; /* Size of hash table. */
    register int  i; /* Counter. */
    TBENTRY *entry; /* Entry in hash table. */

    /* Assign the array of text. */
    f->txt = a;

    /* Allocate and initialize a hash table. */
    size = HASHsiz * sizeof (TBENTRY);
    if (f->hashtbl = (TBENTRY *) malloc (size))
    {
        memset ((char *) f->hashtbl, '\0', size);
    }
    else
    {
        return (FALSE);
    }

    /* If there are any lines: */
    if (n > 0)
    {
        /* Allocate an array of line structures. */
        if (f->line = (LINEINF *)
            malloc ((n + 1) * sizeof (LINEINF *)))
        {
            /* Loop through the lines. */
            for (i = 1; i <= n; i++)
            {

```

(continued on next page)

FILE COMPARISONS

Listing One *(Listing continued, text begins on page 28.)*

```
/* Calculate the hash code value. */
f->line[i].hash = calc_hash (f->txt[i]);

/* Locate the entry in the hash table. */
entry = f->hashtbl + f->line[i].hash;

/* Update the linked list of lines with */
/* the same hash code. */
f->line[entry->last].nxtln = i;
f->line[i].nxtln = 0;

/* Update the first and last line */
/* information in the hash table. */
if (entry->first == 0)
{
    entry->first = i;
}
entry->last = i;
}
}
else
{
    return (FALSE);
}
}
else
{
    f->line = NULL;
}

return (TRUE);
}

/*****
** calc_hash calculates a hash code for a line of text.
**
** RETURNS: a hash code value.
*****/
static HASH calc_hash (buf)

char *buf; /* (I) Line of text. */
{
    register unsigned int  chksum; /* Checksum. */
    char                  *s;      /* Pointer. */
    HASH                   hash;   /* Hash code value. */

    /* Build up a checksum of the characters in the text. */
    for (chksum = 0, s = buf; *s; chksum ^= *s++)
    {
        ;
    }

    /* Combine the 7-bit checksum and as much of the */
    /* length as is possible. */
    hash = ((chksum & 0x7F) | ((s - buf) << 7)) & MASK;

    return (hash);
}

/*****
** Given starting and ending line numbers, fnd_seq finds a
** "good sequence" of lines within those ranges. fnd_seq
** then recursively finds "good sequences" in the sections
** of lines above the "good sequence" and below it.
*****/
static void fnd_seq (f1, beg1, end1, f2, beg2, end2, lngval)

FILEINF *f1; /* (I) File information for #1. */
int beg1; /* (I) First line # to compare in #1. */
int end1; /* (I) Last line # to compare in #1. */
```

```

FILEINF *f2; /* (I) File information for #2. */
int beg2; /* (I) First line # to compare in #2. */
int end2; /* (I) Last line # to compare in #2. */
int lngval; /* (I) "Long enough" value. */
(
LINEINF *line1; /* Line information ptr in #1. */
LINEINF *line2; /* Line information ptr in #2. */
register int limit; /* Looping limit. */
int ln1; /* Line number in #1. */
int ln2; /* Line number in #2. */
register int ln; /* Working line number. */
BOOLEAN go; /* Continue to loop? */
int most; /* Longest possible seq. */
int most1; /* Longest possible due to #1. */
int most2; /* Longest possible due to #2. */
int cnt; /* Length of longest seq. */
int oldcnt; /* Length of prev longest seq. */
int n; /* Length of cur longest seq. */
int m1; /* Line of longest seq. in #1. */
int m2; /* Line of longest seq. in #2. */

/* Initialize. */
go = TRUE;
line1 = f1->line;
line2 = f2->line;

/* Initialize longest sequence information. */
cnt = 0; /* Length of longest seq. */
m1 = beg1 - 1; /* Line # of longest seq. in #1. */
m2 = beg2 - 1; /* Line # of longest seq. in #2. */
oldcnt = 0; /* Length of prev longest seq. */

/* Calculate maximum possible number of consecutive */
/* lines that can match (based on line # ranges). */
most1 = end1 - beg1 + 1;
most2 = end2 - beg2 + 1;

/* Scan lines looking for a "good sequence".
** Compare lines in the following order of line numbers:
**
** (1, 1)
** (1, 2), (2, 1), (2, 2)
** (1, 3), (2, 3), (3, 1), (3, 2), (3, 3)
** etc.
*/
for (ln1 = beg1, ln2 = beg2; TRUE; ln1++, ln2++)
(
if (ln2 <= end2 - cnt)
/* There are enough lines left in #2 such that it */
/* is possible to find a longer sequence. */
(
/* Determine the limit in #1 that both */
/* enforces the order scheme and still makes */
/* it possible to find a longer sequence. */
limit = min (ln1 - 1, end1 - cnt);

/* Calculate first potential match in #1. */
for (ln = f1->hashtbl[line2[ln2].hash].frst;
ln && ln < beg1; ln = line1[ln].nxtln)
{
;
}

/* Loop through the lines in #1. */
for (; ln && ln <= limit; ln = line1[ln].nxtln)
{
if (line1[ln].hash == line2[ln2].hash &&
line1[ln + cnt].hash ==
line2[ln2 + cnt].hash &&
!(ln - m1 == ln2 - m2 &&
ln < m1 + cnt && m1 != beg1 - 1))
/* A candidate for a longer sequence has */

```

(continued on next page)

FILE COMPARISONS

Listing One (Listing continued, text begins on page 28.)

```
/* been located. The current lines */
/* match, the current lines + cnt match, */
/* and this sequence is not a subset of */
/* the longest sequence so far. */
(
    /* Calculate most possible matches. */
    most = min (endl - ln + 1, most2);

    /* First compare hash codes. If the */
    /* number of matches exceeds the */
    /* longest sequence so far, then */
    /* compare the actual text. */
    if (chk_hashes (line1 + ln,
                    line2 + ln2, cnt) &&
        (n = cnt_matches (f1->txt + ln,
                          f2->txt + ln2, most)) > cnt)
    /* This is the longest seq. so far. */
    (
        /* Update longest sequence info. */
        oldcnt = cnt;
        cnt = n;
        m1 = ln;
        m2 = ln2;

        /* If it's long enough, end the */
        /* search. */
        if (cnt >= lngval)
        {
            break;
        }

        /* Update limit, using new count. */
        limit = min (ln1 - 1, endl - cnt);
    )
)

/* If it's long enough, end the search. */
if (cnt >= lngval)
{
    break;
}
most2--;
)
else
{
    go = FALSE; /* This file is exhausted. */
}

/* Repeat the process for the other file. */
if (ln1 <= endl - cnt)
{
    limit = min (ln2, end2 - cnt);

    for (ln = f2->hashtbl[line1[ln1].hash].frst;
         ln && ln < beg2; ln = line2[ln].nxtln)
    {
        ;
    }

    for (; ln && ln <= limit; ln = line2[ln].nxtln)
    {
        if (line1[ln1].hash == line2[ln].hash &&
            line1[ln1 + cnt].hash ==
            line2[ln + cnt].hash &&
            !(ln1 - m1 == ln - m2 &&
              ln1 < m1 + cnt && m2 != beg2 - 1))
        {
            most = min (end2 - ln + 1, most1);

            if (chk_hashes (line1 + ln1,
                            line2 + ln, cnt) &&
```

```

        (n = cnt_matches (f1->txt + ln1,
                        f2->txt + ln, most)) > cnt)
    {
        oldcnt = cnt;
        cnt     = n;
        m1     = ln1;
        m2     = ln;

        if (cnt >= lngval)
        {
            break;
        }

        limit = min (ln2, end2 - cnt);
    }
}

if (cnt >= lngval)
{
    break;
}
most1--;
}
else if (!go)
{
    break; /* This file is exhausted, also. */
}
}

/* If the longest sequence is shorter than the "long */
/* enough" value, the "long enough" value can be */
/* adjusted for the rest of the comparison process. */
if (cnt < lngval)
{
    lngval = cnt;
}

if (cnt >= 1)
/* Longest sequence exceeds minimum necessary size. */
{
    if (m1 != beg1 && m2 != beg2 && oldcnt > 0)
    /* There is still something worth comparing */
    /* previous to the sequence. */
    {
        /* Use knowledge of the previous longest seq. */
        fnd_seq (f1, beg1, m1 - 1,
                f2, beg2, m2 - 1, oldcnt);
    }

    /* Report the sequence. */
    rpt_seq (m1, m2, cnt);

    if (m1 + cnt - 1 != end1 && m2 + cnt - 1 != end2)
    /* There is still something worth comparing */
    /* subsequent to the sequence. */
    {
        fnd_seq (f1, m1 + cnt, end1,
                f2, m2 + cnt, end2, lngval);
    }
}
}

/*****
** chk_hashes determines whether this sequence of matching
** hash codes is longer than cnt. It knows that the first
** pair of hash codes is guaranteed to match.
**
** RETURNS: TRUE if this sequence is longer than cnt.
**          FALSE if this sequence is not longer than cnt.
*****/

```

(continued on next page)

FILE COMPARISONS

Listing One (Listing continued, text begins on page 28.)

```
static BOOLEAN chk_hashes (line1, line2, cnt)

LINEINF      *line1; /* (I) Line information for #1. */
LINEINF      *line2; /* (I) Line information for #2. */
register int  cnt;   /* (I) Count to try to exceed. */
{
    register int  n; /* Count of consecutive matches. */

    for (n = 1; n <= cnt &&
         ((++line1)->hash == (++line2)->hash); n++)
    {
        ;
    }

    return (n > cnt);
}

/*****
** cnt_matches counts the number of consecutive matching
** lines of text.
**
** RETURNS: number of consecutive matching lines.
*****/

static int cnt_matches (s1, s2, most)

char      **s1; /* (I) Starting line in file #1. */
char      **s2; /* (I) Starting line in file #2. */
register int most; /* (I) Most matching lines possible. */
{
    register int  n; /* Count of consecutive matches. */

    /* Count the consecutive matches. */
    for (n = 0; n < most && strcmp (*s1++, *s2++) == ALIKE;
         n++)
    {
        ;
    }

    return (n);
}

/*****
** rpt_seq reports a matching sequence of lines.
*****/

static void rpt_seq (m1, m2, cnt)

int  m1; /* (I) Location of matching sequence in #1. */
int  m2; /* (I) Location of matching sequence in #2. */
int  cnt; /* (I) Number of lines in matching sequence. */
{
    fprintf (stdout,
             "Matched %5d lines: (%5d - %5d) and (%5d - %5d)\n",
             cnt, m1, m1 + cnt - 1, m2, m2 + cnt - 1);
}

```

End Listing