[*Mechanical Translation*, vol.5, no.2, November 1958; pp. 74-83]

The Storage Problem†

William S. Cooper, Massachusetts Institute of Technology, Cambridge, Massachusetts

The bulkiness of linguistic reference data, contrasted with the limited capacity of existing random-access memory units, has aroused interest in means of conserving storage space. A dictionary, for example, can be considerably compressed, yet at the same time virtually all of its usefulness can be retained. Various approaches to compression are described and evaluated. One of them is singled out for extensive treatment. This approach allows considerable compression of the "argument" part of each dictionary entry, yet it introduces no chance of lookup error, provided the item to be looked up is indeed in the dictionary.

The Storage Problem

A DIGITAL COMPUTER can be used to process a staggering quantity of data. Data that is to be processed needs not tax the memory of the computer, since it can be dealt with a little at a time, and then disposed of. Sometimes, however, the processing itself requires a large store of reference data, and such data must remain accessible throughout the processing — and preferably in the most efficient memory medium available. The mechanical translation process falls into this class; it is inevitable that dictionary or glossary information of some kind must be stored in quantity for reference. Other long tables of linguistic data may also be found useful for translation. The proportion of this reference data that can be stored in the high-speed memory units depends partly on the capacity of the units, and partly on the cleverness of the programmer.

The capacity of most high-speed, random-access memory units which are presently in use for MT experiments is small compared with linguists' needs. Without sophisticated packing techniques, even the information in a small pocket dictionary could hardly be fitted into the high-speed storage of these computers. Special arrangements of the dictionary help (for example, maintenance of a short subdictionary of the most common words in high-speed storage), but it is still necessary to be frugal with memory space. Large capacity, high-speed storage units are being developed, and these should eventually ease the problem, but meantime stop-gap techniques for stretching the effective capacity of existing storage facilities are needed.

The programmer is thus faced with the task of shrinking the dictionary to a minimum volume, without substantially impairing its usefulness. The obvious approach is to attempt to code the data in question into a form that is more compact, but that retains all the original information. An example would be the following rule: "For English, delete every 'u' that follows a 'q'. " Note that this coding process is reversible, for the more compact, coded form may be expanded back to its original form by the rule: "Insert a 'u' after every 'q'."

However, the formulation of rules as simple as the foregoing is highly empirical. Furthermore, simple rules rarely provide a useful degree of contraction. On the other hand, more complex coding operations lead to the ridiculous situation in which storage space equalling that required by the dictionary is needed to encode the material to be looked up or read out. So such recoding approaches, at least at present, seem rather unrewarding.

---

† This work was supported in part by the U. S. Army (Signal Corps), the U. S. Air Force (Office of Scientific Research, Air Research and Development Command), and the U.S.Navy (Office of Naval Research); and in part by the National Science Foundation.

1. M.M. Astrahan, "The role of large memory in scientific communications," Research and Engineering (Datamation) 4, 34-39 (Nov.-Dec. 1958).

Argument Compression

A more practical approach is to settle for the compression of only part of each entry. The name "argument compression" derives from the viewpoint that a dictionary can be considered as a function. If X symbolizes the word or phrase to be looked up, the dictionary specifies the value of F(X). For example, a French-English dictionary might yield the function value F(X) = "n.,boy" if the argument X = "garçon" were looked up. An entry in the dictionary is thought of as the pair [X, F(X)] for some particular X. Argument compression is confined to whittling down the length of X for every entry.

Although argument compression is a compromise measure, it is nevertheless a very useful one. Certainly in applications where the arguments are long and the function values short, it is most valuable. But even when both X and F(X) are long, argument compression paves the way for some very convenient arrangements. The components of an entry [X, F(X)] may be separated physically in storage, so long as an indication of the location of F(X) is obtained by finding X. (The indication could be the machine address of F(X), which would be stored along with X; or perhaps the location of F(X) could be made derivable from the machine address of X.) In particular, the compressed X's could be kept in core storage, for example, and the uncompressed F(X)'s relegated to tape. In many circumstances, the greater facility with which lookup operations can be performed might recommend this arrangement. Furthermore, a useful element of F(X), such as a part-of-speech tag, might be allowed to accompany X in high-speed storage. If each F(X) comprises several words, it might be practical to list on tape all words appearing in at least one F(X); then F(X) could be indicated by serial numbers accompanying X in core storage. These examples point to the variety of factors that may make argument compression worth while.

Argument compression is unlike the reversible encoding process previously described. All that is required of an argument compression process is that it leave the arguments sufficiently intact to allow one of the entries to be singled out as the correct one. Consequently, a wide variety of devices is available. These devices can be divided into methods that compress each argument individually and methods that compress each argument in a manner dictated by the arguments of neighboring entries.

Suppose that every argument has N characters, or fewer; the first type of device compresses by discarding information from each argument in some ad hoc manner, so that the remainder has the desired length of N' characters. The truncation of every argument after its N^{th} character would be a crude example. Equally unsophisticated would be the removal of some arbitrary portion of each argument, say, every third character. A little better is the system that replaces each argument by its "check sum," which is merely the sum of its characters when the characters are regarded as digits in some number system. In binary computers, arguments must, of course, lie in binary form. One can capitalize on this by forming a "logical check sum"; each argument can be divided into sections of length N', and the logical sum or product of the sections taken. More complicated schemes can be devised at will. In all instances, the X to be looked up must be mutilated in the same fashion as were the entry arguments and then looked up by an ordinary search routine.

In general, automatic dictionaries are susceptible to two kinds of error:

Error 1. When X is indeed in the dictionary, either no value or a mistaken value of F(X) is yielded by the lookup program.

Error 2. When X is not in the dictionary, an F(X) is assigned to it anyway and is, therefore, extraneous.

The compression devices described in the preceding paragraph introduce the possibility of both kinds of error, the reason being that there is no guarantee against two or more different arguments being compressed down to the same form. However, the probability of this happening is surprisingly low[2] if the desired length N' is large enough and if the system of compression is sufficiently "random." If the instances of two arguments being compressed into the same form are few enough, Error 1 can be eliminated by listing the problematic arguments separately in the computer and by checking X against the exceptions list before it is looked up. And there is always the resort of trying slightly modified compression schemes until one that introduces a low error risk is found.

---

2. D.Panov, "Concerning the problem of machine translation of languages, " Publication of The Academy of Sciences of the U.S. S. R., pp. 9-10, 1956.

Such systems have a special advantage: if N' is set equal to or less than the length of a machine address, and every argument can be compressed to length N', then each F(X), or an indication of the location of F(X), can be stored in the register whose address equals the compressed form of X. Not only is the storing of X avoided completely, but the lookup is immediate and involves no trial-and-error system. When data from short dictionaries or subdictionaries is to be stored in a machine featuring multiple address instructions, this arrangement may be ideal.

The second type of device for argument compression depends on some special ordering of the dictionary entries. Then only the relationships between the arguments of succeeding entries need be stored. Here is an instance where the relationships between arguments are so simple that they are known a priori: A table of the cube roots of the positive integers may be stored merely by storing the ascending values of the cube roots in successive registers; the z^{th} register then contains $3\sqrt{z}$, and arguments may be dispensed with.

Unfortunately, dictionary arguments are not as tightly interrelated as numerical arguments usually are. But the imposition of some ordering — say, alphabetic — immediately creates redundancy in the left-hand columns of a list. For example, the following eight words might be found as arguments of consecutive entries in a French-English dictionary:

garçon
garçonnier
garde
gardon
garer
gargantuesque
gargariser
garnir

Only the underlined part of each word differs from its upstairs neighbor. It has been suggested[3] that certain redundant parts of each entry could be deleted and replaced by an indication of the number of letters to be brought down from the preceding entry. For example, this dictionary segment could be stored as:

0garçon
6nier
3de
4on
3er
3gantuesque
5riser
3nir

This representation has the advantage of being reversible, for the dictionary arguments could be reconstructed in full. Neither Error 1 nor Error 2 would occur. The disadvantage of the representation is that the compressed forms are of unequal length, some of them still being very long.

It is a striking and apparently little-known fact that if a word is known to be in the list, it is unnecessary to store anything but the following list, which consists of an indication of the number of letters to be brought down and the first letter of the remainder of each word:

--
6n
3d
4o
3e
3g
5r
3n

Furthermore, if the list is based on the equivalent binary spelling of words rather than on their alphabetic spelling, it is necessary to store only the number of binary digits to be brought down from the preceding entry — the first digit in the remainder is always a one.

The rest of this paper develops the idea and describes the way a word can be looked up in such a list. We call this system "constituent compression." It has the following features:

a) There is no risk of Error 1.

b) It compresses to a high degree. In a binary machine it can shrink an N-bit word down to as few as $N' = \log_2 N$ bits.

c) The lookup method is fairly complicated and slow, although perhaps no more so than the alternative that would be forced by longer arguments . Provision for looking up several words at one time makes the lookup program more efficient.

d) In applications where an Error 2 is possible, the probability of such can be lowered at the cost of retaining, somewhere in the computer, more information from the original argument list.

3. W.N.Locke and A.D.Booth (editors), Machine Translation of Languages, (The Technology Press of M.I.T. and John Wiley and Sons, Inc., New York, May 1955), Chap. 5, "Some problems of the 'word'," by W. E. Bull, C. Africa and D. Teichroew.

Terminology of Constituent Compression

An argument in a dictionary is a string of alphabetic characters, but we must endow it with numerical properties. It is possible to identify each character with a digit in the number system with radix r, where r is at least as large as the number of different characters to be dealt with. But since the argument must certainly become a series of digits when it is placed in storage, it is probably more natural to regard the coded string as the character string. In this case, the radix r would simply be the base of the computer, e.g., r = 2 for binary computers.

Imagine that the arguments are arranged in a vertical list. Append leading zeros to the shorter arguments until all have a common length of N characters. If there are M arguments all told, the list resembles an MxN matrix having the augmented argument A as its typical row:

$$(1)\qquad \begin{aligned} A_1 &= a_{1,1} \ldots a_{1,n} \ldots a_{1,N} \\ A_m &= a_{m,1} \ldots a_{m,n} \ldots a_{m,N} \\ A_M &= a_{M,1} \ldots a_{M,n} \ldots a_{M,N} \end{aligned}$$

The lower-case a's are individual characters which are considered as digits, and a row A is a single number. Our ordering restriction requires that

$$(2)\qquad A_i < A_{i+1} < \ldots < A_j < \ldots < A_{k-1} < A_k$$

under the convention $1 \le i < j < k \le M$.

Next in some number system with radix s (usually s=r), we form a strictly decreasing series of N non-negative integers:

$$(3)\qquad b_1 > b_2 > \ldots > b_n > \ldots > b_{N-1} > b_N$$

When some $a_{m,n}$ from (1) is written after the corresponding b_n from (3), the combination is called a constituent of A_m, and might be denoted $b_n a_{m,n}$ where the conjunction denotes "write end to end" rather than "multiply." When it is not desirable to specify a particular n, C_m denotes any one of the N constituents of A_m. Every constituent can be read as a number in some system with radix as large as the greater of r and s. The expression $C_i \in A_i$ is to be read "C_i is a constituent of A_i." Likewise, $C_i \in A_j$ may be read "C_i is equal to some constituent of A_j." $C_i \notin A_j$ means "C_i is not equal to any constituent of A_j."

For illustration, suppose all characters are decimal digits and r=s=10. Form series (3) from the decreasing integers N through 1. For A_m = 009408, the six consituents of A_m are displayed: $b_1 a_{m,1}$ = 60; $b_2 a_{m,2}$ = 50; $b_3 a_{m,3}$ = 49; $b_4 a_{m,4}$ = 34; $b_5 a_{m,5}$ = 20; $b_6 a_{m,6}$ = 18. These constituents also might have been denoted C_m. The following are true statements: $20 \in A_m$; $18 \in 009408$; $35 \notin A_m$.

A zero constituent is a constituent $C_m = b_n a_{m,n}$ in which $a_{m,n} = 0$. The zero constituents of our example are 60, 50, and 20. The rest are its non-zero constituents.

The distinguishing constituent of A_j with respect to A_i is a function of the two variables, A_j and A_i. But instead of notating it $C(A_j, A_i)$, we write C_j^i, which is intended to suggest that the value of the function is a constituent of A_j. It is defined as the largest constituent of A_j for which no identical constituent may be found in A_i. More precisely, the following two statements hold:

a) $C_j^i \notin A_i$.

b) If $C_j > C_j^i$ for some C_j, then $C_j \in A_i$.

For example, if A_j = 009408 and A_i = 009266, it is found that C_j^i = 34, and that C_i^j = 32, when (3) is chosen as before.

Under our convention $A_j > A_i$, $a_{m,n} \neq 0$ in any $C_j^i = b_n a_{m,n}$. In the binary number system, $a_{m,n} = 1$ in a distinguishing constituent;

	ARGUMENT MATRIX	CONSTITUENT LIST
A_1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 1 1 1 1 0 0 1	8
A_2	0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 1 1 1 0 1 0 1 0 1	10
A_3	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 0 1 **1** 0	1
A_4	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 **1** 0 0 0 0 1	5
A_5	0 0 0 0 0 **1** 0 1 1 0 1 1 0 0 0 1 0 0 0 1 1 1 1 1	18
A_6	0 0 **1** 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 0 0 1 1 0 1	21
A_7	0 0 1 1 1 **1** 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 1 1 0	18
A_8	0 0 1 1 1 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 1 1 **1**	0
A_9	0 0 1 1 1 1 0 1 0 1 0 0 0 1 0 0 **1** 1 0 0 1 1 0 0	7
A_{10}	0 0 1 1 1 1 0 1 0 1 0 0 0 1 **1** 0 1 1 1 1 0 0 0 0	9
A_{11}	**1** 1 0 1 1 1 1 0 0 1 0 0 0 1 0 1 1 1 1 0 1 1 0 1	23
A_{12}	1 1 0 1 1 1 1 **1** 1 1 0 0 0 0 0 0 1 0 0 1 1 0 0 1	16

Fig. 1 A binary example of an argument matrix of form (1) with the corresponding constituent list to be stored in its place.

hence it is implicit, and may be omitted. In binary, C_j^i takes the form b_n, which implicitly records the fact that A_j resembles A_i up through its $n\text{-}1^{th}$ bit, and that the n^{th} bit of A_j is different and is a one-bit. We are most interested in distinguishing constituents of the special form C_j^{j-1}; these point out the leftmost digits that differ from their upstairs neighbors in the A_j's of (1). For illustration, these crucial digits are written inside boxes in the binary matrix in Figure 1; a row $A_0 = 0$ is assumed, to provide a definition for C_1^0. The other features of Figure 1 will be touched upon in following sections.

The constituent list, which serves as the substitute for (1) in storage, is a list of M constituents. Its m^{th} member is the distinguishing constituent C_m^{m-1}. This rule defines the entire list except for its first member, C_1^0, which may be taken to be any non-zero constituent of A_1. Figure 1 includes a constituent list. It is written in decimal instead of binary form for readability. To form the list the series (3) was taken to be the integers N-1 through 0. With this choice, the orders of the boxed bits in the argument matrix make up the constituent list. Therefore a statement such as "$C_i^{i-1} \epsilon X$" could be interpreted, "The bit of X of order C_i^{i-1} is a one-bit," in a binary machine.

Lookup Procedure

Assume that X is in the dictionary. This means $X = A_j$ for some value of j. Just what do we seek when we "look up X"? The usual answer would be that the A_m of the argument matrix (1) must be pointed out as an identical match for X. However, under a compression system, the matrix (1) is never actually stored as it stands, so we must be satisfied

with determining the position m of the A_m which would have matched X if (1) had been accessible in its entirety. Thus, in theory, we seek the value of m for which $F[X] = F[A_m]$. For programming purposes, it might be more convenient to handle $F[A_m]$, or its machine address, than to handle m itself.

If (1) could be stored, a slow but sure procedure for looking up X would be to compare X with $A_1, \ldots, A_m, \ldots, A_M$ in turn, and during the process to take note of the position m for which $X = A_m$. But (1) is to be compressed into a constituent list, with the result that direct comparisons are impossible. Therefore, as will be seen later, we have the complication that not just one, but probably many, positions will be noted as the list is swept. Each one is merely a nominee for the desired value of m. After the list has been swept, the choice of the nominees is easily made; the correct one is simply the largest value of m – that is, the value most recently nominated. More precisely, the search must be designed to fulfill two conditions:

Condition I: If $X = A_j$, position j must be nominated.

Condition II: If $X = A_j$, no position m can be nominated if $m > j$.

After such a j is known, the mission is completed, except possibly for gaining access to $F[A_j]$.

The nominator is a block of storage that is set aside for the recording of nominees. If each nominee obliterates the previous nominee while being stored over it, the nominator will automatically display the correct nominee as the search ends.

The carrier is another block of storage that is set aside for bookkeeping purposes. Its contents are excerpts from the constituent list, and change constantly as the search proceeds.

The X to be looked up must, of course, be accessible during the search. If the programmer decides to decompose X into its constituent form, only the non-zero constituents need be available to the search.

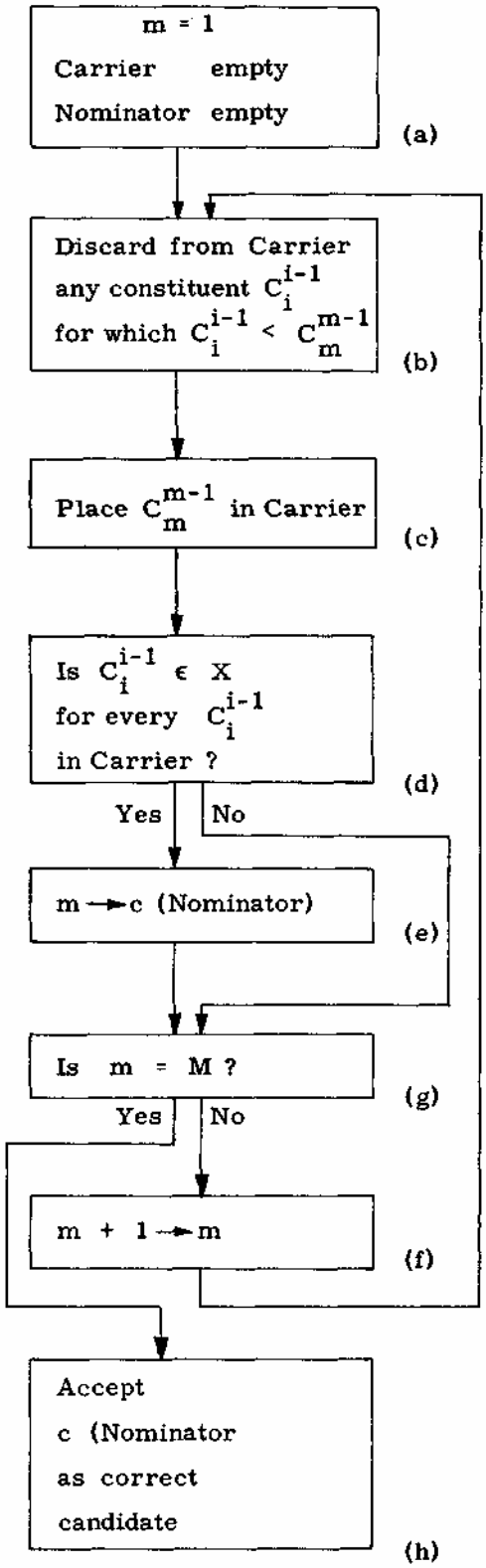

Fig. 2. In this flow diagram, the carrier must be designed to hold as many as N constituents.

The problem: look up X =

0 0 1 1 1 1 0 1 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 0.

CYCLE	CONTENTS OF CARRIER	NOMINATOR
m=1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0	-
m=2	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	2
m=3	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0	2
m=4	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0	2
m=5	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	5
m=6	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	6
m=7	0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	7
m=8	0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	7
m=9	0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0	9
m=10	0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	9
m=11	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	9
m=12	1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	9

The solution: X = A_9

Fig. 3. Contents of carrier and nominator as search of Fig. 2 proceeds down the constituent list of Fig. 1.

There seem to be at least two approaches to performing the search. The first uses a carrier that is equipped to record as many as N constituents at a time. In the second, the carrier contains at most one constituent at a time. The approaches are most easily described and distinguished by means of flow diagrams. They will be discussed in the following two sections.

Search Using a Multiconstituent Carrier

Figure 2 illustrates how a search might proceed. Given the initial conditions of box (a), the loop is traversed M times, one cycle for each successive position m. Boxes (b) and (c) may be regarded as maintenance rules for the carrier, to bring it up to date with m. Box (d) makes the crucial decision of whether or not to nominate the current value of m. An arrow should be interpreted as "replaces, " and c(z) means "contents of z."

A special format for the carrier may be helpful. Let the carrier be simply an N-digit register in the computer:

$$(4) \qquad d_1 d_2 \ldots d_n \ldots d_{N-1} d_N$$

At box (a), every d_n is set equal to zero. In order to place a constituent $C_m^{m-1} = b_n a_{m,n}$ in the carrier, set d_n at the value of $a_{m,n}$ To remove it, set $d_n = 0$ once again. It can be shown that no two constituents need ever share the same d_n in the carrier. The format for the carrier described by (4) allows boxes (b), (c) and possibly (d) to be executed efficiently with shifting operations, especially if the sequence (3) is judiciously chosen so that its members dictate the amount of shift. Also, with format (4), the question of box (d) may be rephrased into a weaker form: "Is each $d_n \leq x$?" where x_n is the n^{th} digit of X.

In a binary machine, format (4) for the carrier may be exploited further. The question of box (d) becomes, "Is $x_n = 1$ for every n for which $d_n = 1$?" Logical operations give a fast answer.

Figure 3 illustrates the problem of looking up X= 001 111 010 100 010 011 001 100 by using only the constituent list in Figure 1. Each line of Figure 3 shows the state of the search after the main cycle of Figure 2 has been performed. The special format (4) has been used to display the contents of the carrier. In place of a value of m, either $F(A_m)$ or its machine address could have been stored in the nominator.

Search Using a Single-Constituent Carrier

If the test of box (d) in Figure 2 remains unwieldy in spite of attempted streamlining, a different approach is needed. Figure 4 displays a search method in which the carrier is never required to carry more than one constituent at a time. Therefore special formats for the carrier need not be devised. Figure 5 illustrates the same problem as did Figure 3. This time, however, the flow diagram of Figure 4 was used for its solution.

Explanation of the Procedures

The lookup procedures of Figure 2 and Figure 4 work on the same principle. Since the binary case is the most easily visualized, we will take as our illustration the argument matrix of Figure 1. Dotted horizontal lines extend from above the boxed one-bits to the right edge of the matrix. Because the list is ordered in ascending magnitude, two little theorems may be proved:

Theorem I: Starting at each boxed one-bit, a "chain" of 1's extends downward until a dotted line is reached (or possibly farther).

Theorem II: Starting just above each boxed one-bit, a chain of zeros extends upward until a dotted line is reached (or possibly farther).

By using the information in the constituent lists, a "cross-sectional" view of the chain of 1's of Theorem I is reconstructed in the carrier for each position m. The search of Figure 2 reconstructs cross-sections of all of these chains (as is apparent in Figure 3), whereas the search of Figure 4 keeps track only of one chain at a time. In either search, every position m is

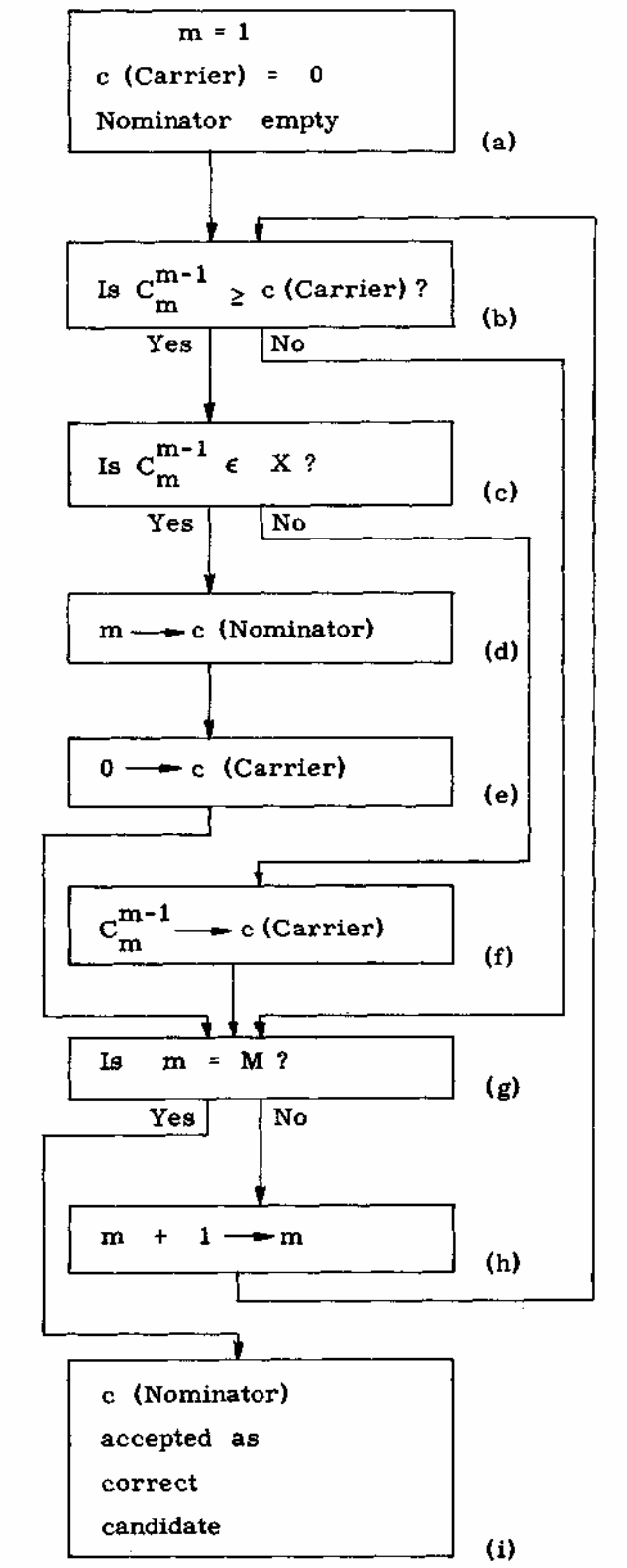

Fig. 4. In this flow diagram, the carrier holds only one constituent at a time.

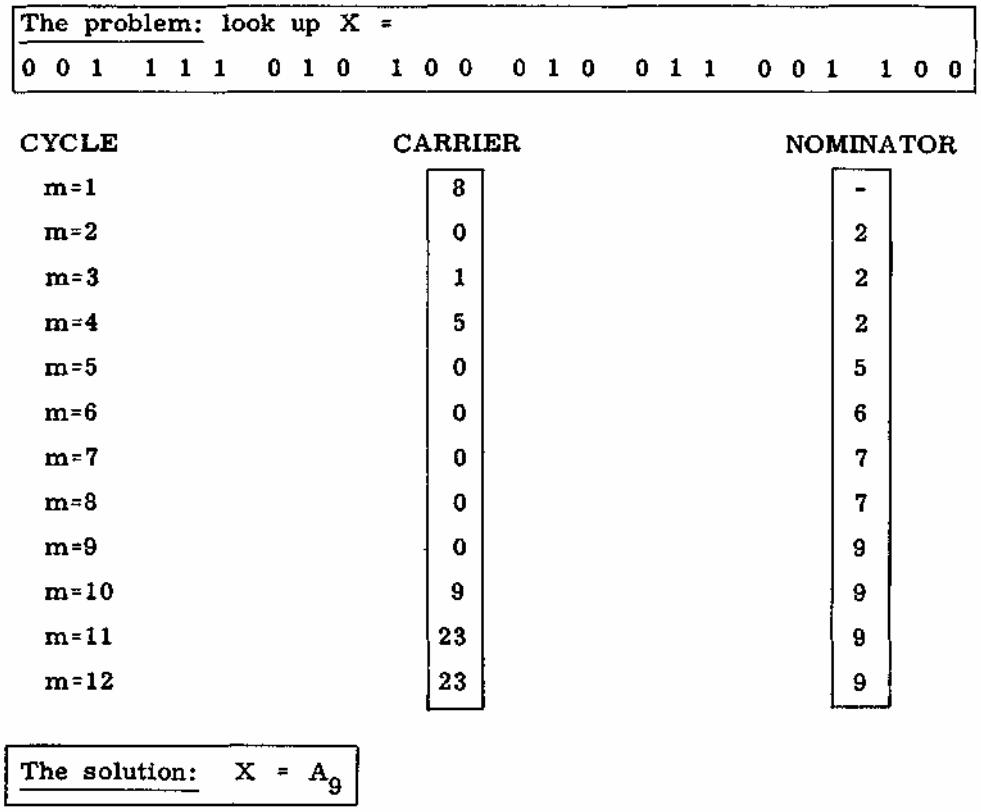

Fig. 5. Contents of carrier and nominator as search of Fig. 4 proceeds down the constituent list of Fig. 1.

automatically nominated, unless the one-bit of some chain in the cross-section for cycle m corresponds to a zero-bit in X.

We know that $X = A_j$ for some j. Condition I, given previously, requires that position $m = j$ be nominated. This will certainly happen, since the 1-chains in the cross-section at $m = j$ are by construction those that are known to pass through A_j. Therefore Condition I is fulfilled, and j is nominated.

Condition II requires that no position $m = k > j$ be nominated. For any position $k > j$, consider the leftmost 1-chain that passes through A_k but that does not extend as far up as A_j. (The search of Figure 2 keeps track of all chains, and hence of this one, whereas the search of Figure 4 is especially designed to keep track only of this chain.) For this particular chain, no dotted lines lie between it and A_j, so Theorem II shows that the one-bit in this chain which lies in A_k corresponds to a zero-bit in A_j above. Position k is not nominated, and Condition II holds. Conditions I and II are necessary and sufficient for the scheme to work.

A rigorous proof is possible. The C_j^i notation is convenient for proofs when $r \neq 2$. The basic stepping stones are theorems that generalize the foregoing theorems. They are stated here in case the reader might enjoy proving them.

Theorem I:

Given matrix (1) under constraint (2). If $C_{i+1}^{i}, C_{i+2}^{i+1}, C_{i+3}^{i+2}, \ldots, C_{j-1}^{j-2}, C_{j}^{j-1} < C_i$ for some C_i, then $C_i \in A_j$ for this C_i.

Theorem II:

Given matrix (1) under constraint (2). If $C_{j+1}^{j}, C_{j+2}^{j+1}, \ldots, C_{k-2}^{k-3}, C_{k-1}^{k-2} < C_k$ for some C_k, and if this $C_k \notin A_{k-1}$, then $C_k \notin A_j$ for this C_k.

Additional Possibilities

So far, it has been assumed that the main cycles of a search must be performed M times. However, a prior knowledge of the approximate position of X down the list is often obtainable; perhaps X is known to match some A_m between A_{m_1} and A_{m_2}. In this case only the relevant range need be searched, so parameters m_1 and m_2 could be inserted in boxes (a) and (g), in either search. If no prior knowledge of the position of X is obtainable, some search time still can be salvaged. Notice that if the largest non-zero constituent of X is exceeded in magnitude by C_m^{m-1}, then neither position m nor any position beyond it is nominable. This statement supplies a stop rule that gives warning when further searching is pointless.

In a scheme that fulfills Condition II, it is plain that if $X_{y_1} > X_{y_2}$, and if position m is a nominee with regard to X_{y_1}, then m cannot be the correct nominee with regard to X_{y_2}. This observation suggests that the X's be ordered according to magnitude prior to the search, the subscripts being assigned so that

$$X_1 < X_2 < \ldots < X_y < \ldots < X_{Y-1} < X_Y. \tag{5}$$

Then, at any given position m, the X's need be examined in turn only until m is stored as a nominee for some X_y. We now have another stop rule that assures us that the remaining X's may be ignored at position m.

An elaborate but efficient program utilizes both of the preceding stop rules: as m increases, a rising floor value of y is determinable from the first rule, whereas the second rule determines a ceiling value of y at each cycle. Only those X's of (5) carrying subscripts between the floor and ceiling values of y need be considered during any given cycle.

Throughout the discussion, we have assumed that $X = A_j$ for some argument A_j; that is that X is indeed to be found in the dictionary. If we leave the system as it stands, an error of the type described previously as Error 2 is certain to occur whenever a word not contained in the dictionary is looked up. For some special applications, the situation could never arise. With a large enough dictionary, it might arise seldom enough to make the errors forgiveable. Otherwise, it would be necessary to supplement the constituent list with further information about the arguments. A few of the rightmost columns of matrix (1) could be stored, in addition to the constituent list, thereby supplying a few "check digits" for each argument. In order to use the information, the check digits from A_m would be compared against the corresponding digits in X at some stage before $F(A_m)$ could be accepted officially as the correct nominee. The extra information needed might reclaim much of the space saved by compression, but on the other hand, one is free to relegate the check information to a slower storage medium, perhaps along with the F(X)'s. If this sort of error check were programmed, the risk of an occurrence of Error 2 could be reduced to negligible proportions.

I am indebted to V.H.Yngve, K.C.Knowlton, F.C.Helwig, and M. M. Jones for their suggestions and criticism.