Sequential Coding Algorithms: A Survey and Cost Analysis

JOHN B. ANDERSON, SENIOR MEMBER, IEEE, AND SESHADRI MOHAN, MEMBER, IEEE

Abstract—**The cost of a number of sequential coding search algorithms is analyzed in a systematic manner. These algorithms search code trees, and find use in data compression, error correction, and maximum likelihood sequence estimation. The cost function is made up of the size of and number of accesses to storage. It is found that algorithms that utilize sorting are much more expensive to use than those that do not; metric-first searching regimes are less efficient than breadth-first or depth-first regimes. Cost functions are evaluated using experimental data obtained from data compression and error correction studies.**

I. INTRODUCTION

CODES with a tree structure find wide use in data compression and error correction. They are implicit as well in the equalization of band-limited nonlinear communication channels by sequence estimation. Generally, a tree code is a code whose words may be graphed on a perfectly regular tree structure having a group of σ symbols or a piece of a waveform on each branch and b branches out of each node. Each code word corresponds to a path through the tree made up of these branches. In channel transmission, successive data symbols guide the path level by level deeper into the tree, and a decoder attempts to find this path in noise. In data compression, successive data are compared to branches level by level in an attempt to find a codeword path close to the data. In equalization, digital data symbols presented to a channel cause a set of responses that can be organized in a tree structure.

Any usable tree code is a trellis code. A practical tree code is generated by a finite-state machine, and as a result a pattern of tree nodes can be merged with other nodes to form a trellis structure. Whether the code is viewed as a tree or a trellis, it is generally impractical to view and weigh all the branches in a code, so a search algorithm must be employed which considers some but not all in a predetermined fashion.

The efficiency of these "sequential coding" algorithms has traditionally been measured by the number of branches searched for a given level of performance. It has become increasingly clear that this measure does not indicate the true consumption of resources. In this paper we survey the different types of algorithms that are possible and explain their features. We also develop more accurate measures of cost based on access cycles and storage size, and use these to compare in a systematic way many of the procedures that are in use.

The Viterbi algorithm is not a selective search but an exhaustive one, the most efficient one possible given the finite-state machine description of the code generation or intersymbol interference mechanism. The algorithm has been used with great success in certain channel coding problems where simple codes give acceptable performance. But in other channel problems and in analog-to-digital conversion of sources like speech, codes with many states are required, and at the same time only a few code paths need to be pursued. In sequence estimation, the effect of heavy band limiting is to create a highly complex state description, so that in intersymbol interference rejection and in similar problems like adaptive reception over the HF channel, an effective Viterbi algorithm must be very complex. For all of these problems, a properly designed search algorithm will be effective. Search procedures have proven useful in other applications, including identification of image contours [19], and text and speech recognition [21] and digitizing of images [20].

Code searching schemes may be classified as sorting or nonsorting, and as depth-first, breadth-first, or metric-first, where the "metric" is some measure of fidelity or likelihood. A number of schemes are summarized in Table I. Among algorithms which sort, the well-known stack algorithm (see [1] or [3] for channel decoding or [2] for source encoding) extends code tree paths in a purely metric-first manner, meaning that the next path extended is always the one with the best metric among those presently stored. Sorting is used to single out the best path. The usual method is an ordering procedure, but we shall also analyze a merging procedure which has significantly lower cost. A purely breadth-first algorithm that sorts is the M-algorithm. This algorithm views all branches at once that it will ever view at a given depth, then sorts out and drops paths ending in certain branches before continuing on. Another sorting scheme that is not purely either metric or breadth-first is the bucket algorithm [1].

A second class of algorithms does not sort; that is, paths are never compared with one another. The simplest such method is the single stack algorithm, a purely depth-first method suggested by Gallager [4]. This scheme simply pursues a path until its metric falls below a discard criterion, and at any one time it stores the identity of only one path. A direct implementation is a single push-down stack. (It should be mentioned that the more widely known "stack algorithm" cited above in fact contains no stack, but only a "list.") A familiar variation of the single stack method is the Fano algorithm; the peculiar character of this search stems from its method of computing the discard criterion. A more sophisticated nonsorting procedure is to set aside certain good paths for later attention as they appear in the depth-first search. This method stores a number of paths, but they are known to be good ones; an example is the two-cycle algorithm [5]. Uddenfeldt and Zetterberg [7] have discussed a depth-limited exhaustive search.

The usual measure of efficiency for code searching algorithms has been the *node computation*, the number of branches visited during the progress of the scheme divided by the branches released as output, for a given level of source encoder fidelity or decoder probability of error. As the algorithms have come into more use, however, it has become clear that this is not a sufficient measure. Several authors [8], [15] have found that stack algorithm source encoding is no faster than other methods, even though it has the least node computation of any known method. Experience with M-algorithm hardware speech encoders [10] shows that this method is efficient despite a poorer node computation. In sequential channel decoding and in sequence estimation, the situation is similarly confused. The Viterbi algorithm finds wide use despite its exhaustive nature, even though supposedly more efficient schemes exist. What seems to be missing here is a factor to ac-

Paper approved by the Editor for Signal Processing and Communication Electronics of the IEEE Communications Society for publication after presentation at the International Conference on Communications, Boston, MA, June 1979. Manuscript received October 8, 1982; revised June 10, 1983. This work was supported by the National Research Council of Canada under Grant A8828.

J. B. Anderson is with the Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY 12181.

S. Mohan is with the Department of Electrical and Computer Engineering, Clarkson College of Technology, Potsdam, NY 13676.

0090-6778/84/0200-0169$01.00

TABLE I
SEARCH RATIONALES FOR CERTAIN SELECTIVE SEARCH ALGORITHMS

	Metric-First	Breadth-First	Depth-First
Sorting	Stack Alg. Merge Alg. Bucket Alg. (roughly) --Haccoon's Alg. (both)-- xx←	M-Alg. Mult. Stack Alg.	→xx
Non-Sorting		Simmons-Wittke --2-Cycle Alg. (both)--	Single Stack Alg. Fano Alg.

count for the size and complexity of the required information structures, in addition to the intensity of their use.

II. A Definition of Algorithm Cost

A more realistic measure of algorithm cost can be based on the number of storage elements in a scheme and the number of accesses to them. The *space complexity* of an algorithm is the size of resources that must be reserved for its use, while the *time complexity* counts the number of accesses to this resource. The product of these two, the *space-time complexity*, and the sum of these two are both useful measures of overall cost.

A space–time product cost measure assumes that storage blocks "wear out" after a certain number of accesses and that the cost of blocks is proportional to their speed, assumptions that are roughly true for physical devices. Parallel processing is of no benefit under this measure, since there is no gain in trading space for time. A second measure of cost, more suited to software implementations, is the space + time complexity. The sum of space and time, this measure stresses more the opportunity cost foregone by assigning resources to a user. Different constants are often placed before the two components, but these will have no asymptotic significance. We shall list results for both measures, but go into detail only for the product measure.

A perhaps more traditional measure of complexity for sorting methods is the number of comparisons, but this measure does not account for both space and time, and as mentioned previously, not all code search algorithms sort.

Other measures of coding algorithm cost could be proposed than the space and time cost of storage blocks, but this kind of measure relates closely to the nature of such algorithms. Code search algorithms basically move in and out of storage data about code tree paths. Other tasks, such as computing metrics, checking for ambiguous output symbols, and generating code word letters, form a constant multiplier on the cost of storage access. The major determinants of cost remain the storage size and the pattern of accesses called for by the steps of the algorithm.

A simple building block for search algorithms is the random-access memory (RAM), although many algorithms relate more naturally to another structure, the push-down stack. Building algorithms out of push-down stacks instead of RAM's leads to the same asymptotic complexity in all the cases we discuss except the M-algorithm.

Three variables dominate the asymptotic cost of the algorithms we analyze, the length of path an algorithm can retain, L, the number of paths it can retain, S, and the expected node computation already defined, $E[C]$. We assume that L and S are finite and fixed in value. It is important to realize that algorithms differ significantly when these are so constrained; when the number of paths exceeds S, for instance, some mechanism must delete excess paths, and whenever a path exceeds length L, its oldest branch must be checked to ensure that it is consistent with other paths kept to this depth. These routines may change the asymptotic cost.

TABLE II
ASYMPTOTIC COST OF CERTAIN ALGORITHMS IN THE LIMIT OF INTENSIVE SEARCHING, PER OUTPUT SYMBOL RELEASED. SPACE IN b-ARY SYMBOLS, TIME IN ACCESSES TO STORAGE. C = BRANCHES VISITED/OUTPUT SYMBOL RELEASED, L = LENGTH OF RETAINED PATHS, S = NUMBER OF RETAINED PATHS.

Algorithm	(Space)(Time)	(Space)+(Time)
Single Stack	$\sim LE[C_{SS}]$	$\sim L+E[C_{SS}]$
Fano	$\sim LE[C_{FA}]$	$\sim L+E[C_{FA}]$
2-Cycle	$\sim LE[C_{2C}]$	$\sim LL_{pd}+E[C_{2C}]$
Stack	$\sim LS^2E[C_{SA}]$	$\sim LS+SE[C_{SA}]$
Merge	$\sim L(S^{4/3}E[C_{SA}]+S^2)$	$\sim LS+(S+S^{2/3}E[C_{SA}])$
Bucket	$\sim L(SE[C_{SA}]+S^2)+H$	$\sim LS+(S+E[C_{SA}])+H$
M-	$\sim LS^2$	$\sim LS+S$

We turn now to a space and time cost analysis of a number of algorithms. For clarity, we emphasize sequential source encoding schemes throughout, although the analysis applies as well to channel decoding and sequence estimation. The algorithms chosen for exposition are those which differ from each other in fundamental ways, or which demonstrate a principle in pure form. Often, a variation or a scheme combining several principles will be most effective in applications. Results are summarized in Table II. It is clear that there are dramatic differences among the algorithms; all depend linearly on L, but some, like the single stack algorithm, have no dependence on S, while others range as high as S^2. As has been shown in earlier work, there are also wide variations in the node computation $E[C]$ at a given S, L, and performance level. The M-algorithm has no separate dependence on $E[C]$ because C depends only on S.

It is not our intention to optimize over the choice of the three major cost factors, or over the many lesser factors and parameter settings, but only to establish the cost functions. An accurate optimization is an immense task.

III. Basic Features of Searching Algorithms

It will be convenient first to define features which are common to all algorithms.

The aim of the search is to find a path with distortion or likelihood metric as good as possible. Searching begins at a root node and continues until some path reaches depth L. The algorithm then decides once and for all which first branch to release as output; the end node of this branch becomes the new root node. Searching resumes until some path again reaches total length L branches. The procedure continues indefinitely in this "incremental" fashion, releasing some branch at depth l and accepting a new data group at depth $l + L$. An older attitude toward searching is the "block" search, in which an

L-branch path is released all at once and the search begins anew at the path's end node. We give no separate analysis for this alternative, although most of our conclusions apply.

Paths are described by *path maps* made up of b-ary symbols, $\{0, \cdots, b-1\}$, one for each branch. The code word letters on a path's terminal branch are somehow computable from its path map. Associated with each map is a metric μ, either a likelihood of the path or a measure of its distortion, and sometimes an indication of the path's length or pointers to other storage locations. For source encoding, the metric is given by $\mu(z^{\sigma l}, \hat{z}^{\sigma l}) = \sigma l D^* - d(z^{\sigma l}, \hat{z}^{\sigma l})$, where $d(z^{\sigma l}, \hat{z}^{\sigma l})$ is an additive distortion measure, z and $\hat{z}$ are vectors of source data and code word letters of length σl, and D^* is a target distortion the algorithm hopes to achieve. For channel decoding and sequence estimation another constant, called the bias, replaces D^*, and $d(\ ,\ )$ becomes the negative of the log likelihood of the path.

The following actions are performed by all algorithms and will be denoted throughout by the italicized expressions.

Extend Path: The algorithm extends a path one branch forward, "viewing" the branch. Viewing includes calculating the code word symbols on the branch, fetching the input data group corresponding to its depth (source symbols to be encoded or channel symbols to be decoded), calculating the metric increment for the branch, and forming the new metric total for the path. Branches are viewed either singly or in groups of b, depending on the algorithm. In the former case, only a single branch, say the zeroth is viewed during the first visit to the original path's end node; if there is a later visit the first will be viewed, and so on until all b branches are viewed. Other algorithms *extend b paths,* and view all b at once.

Ambiguity Check: The algorithm checks all eldest path map symbols to determine if they are consistent with the symbol released as output. If a symbol is not, the path must be deleted. Ambiguity checks are necessary for two reasons. If a path fails the check but is kept in storage, it may later be released as output even though its antecedent does not match earlier output. The encoder and decoder will then not develop the same path. Ambiguity checks also prevent the accidental storage of two paths with the same symbols. If two path maps once differ (as they do initially), they can become identical only when the differing symbols are dropped and this is forewarned by the check. Actually, this clogging of storage can be detected in more timely fashion by checking the subset of path symbols that determine the code generator state. Two paths with these subsets identical lead to identical subtrees. But this method requires comparison of a subset of symbols rather than a single symbol. Hang and Woods [22] have studied the effect of identical and nearly identical paths in analog source coding.

A separate ambiguity check is unnecessary in nonsorting algorithms, which store only a single path.

Delete Path: The algorithm deletes an entire path map. A deletion must occur whenever the number of paths stored exceeds S, and whenever a path fails an ambiguity check.

Release Output Symbol: The algorithm releases as output the earliest symbol of the best path map it has. In sorting algorithms, this triggers an ambiguity check to make certain that all path maps have the same symbol at this depth.

In nonsorting algorithms the possibility exists that no path satisfies the constraints of the algorithm, an event we call *algorithm failure.* The cost of recovering from this event is small. The easiest method is to move one branch forward of the root node, declare the branch's end node to be the new root node, and start again.

The ambiguity check has been the source of some debate among those who have used search algorithms. It can be argued that its possibly significant cost is not worth a small performance gain. The authors, however, have calculated that with some channel codes and with small S and L, failure to check for path ambiguity can lead to large performance loss when the channel SNR is high. When the SNR is low, on the other hand, ignoring the ambiguity check can actually improve performance.

IV. Nonsorting Algorithms

The distinguishing feature of nonsorting schemes is that they store only a single path, extending or backtracking along the path in response to the value of the path metric.

A. The Single Stack Algorithm

This algorithm proceeds depth-first directly through the code tree until the path metric falls below a discard criterion B; the search then backtracks to the first untried branch and proceeds depth-first again. The symbols of the path map are stored in a push-down stack, and in the steady-state operation of the algorithm, an output path map symbol is forced out the bottom whenever a depth is visited for the first time.

By convention we assume that on first visiting a node, branch 0 out of the node is viewed, on the second visit branch 1, and so on until all branches are viewed, at which time the search must backtrack. An example of this routine appears in Fig. 1 for $b = 2$; X's indicate a path that has fallen below the discard criterion. It is more straightforward to think of the discard criterion B as a constant, although Davis and Hellman [11] show that B probably must be a function of the input data.

Using a single push-down stack and a comparison to a discard criterion, the single stack algorithm is the simplest of all search algorithms.

Algorithm SS (The Single Stack Algorithm):

SS0) Insert root node into stack. Set $i \leftarrow 0$.

SS1) (View branch.) *Extend path* whose map is in the stack, viewing branch i out of its end node.

SS2) (Check discard criterion.) If $\mu > B$, go to SS3; else go to SS4.

SS3) (Advance forward.) Stack new branch symbol i and *release output symbol* at stack bottom if not null. Set $i \leftarrow 0$. Go back to SS1.

SS4) (Switch to sister branch.) Set $i \leftarrow i + 1$.

SS5) (More sisters?) If $i < b$, go back to SS1.

SS6) (Stack empty?) If stack empty, declare *algorithm failure.*

SS7) (Backtrack.) Pop stack. Set $i \leftarrow$ popped symbol. Go back to SS4.

A flow chart appears in Fig. 2.

The space cost of the single stack algorithm is L b-ary symbols (plus a small overhead for side registers and code letter generation). The time cost is overbounded by two accesses per branch viewed, since the algorithm passes through SS1 once per branch and SS7 at most once. The space-time product cost is thus

$$\cong LE[C_{SS}] \quad \text{access-symbols/output branch} \tag{1}$$

where C_{SS} denotes the node computation of the single stack algorithm, and here and throughout $X \cong Y$ means that $\log X / \log Y$ tends asymptotically to 1.

The Fano algorithm is a variant of the single stack procedure in which the discard criterion varies up and down as a function of the path metric. Searching proceeds depth-first until a path falls below B, but B is raised in increments whenever possible during visits to new nodes, and is lowered when necessary during returns to previously visited nodes. The cost of the Fano algorithm is again of the form $LE[C_{FA}]$, but it is not clear in particular applications which of C_{FA} and C_{SS} is the larger. The opportunistic changing of B undoubtedly reduces the set of nodes visited, but the algorithm may visit certain of the nodes many times. Nonetheless, if C_{FA} can be measured, then the form (1) gives the total cost.

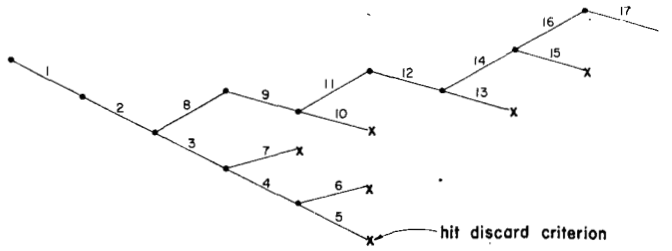

Fig. 1. Push-down stack search, $b = 2$. Downward branch (zeroth) taken first, then upward branch (first); numbers show order of visiting, X means path metric hits discard criterion.

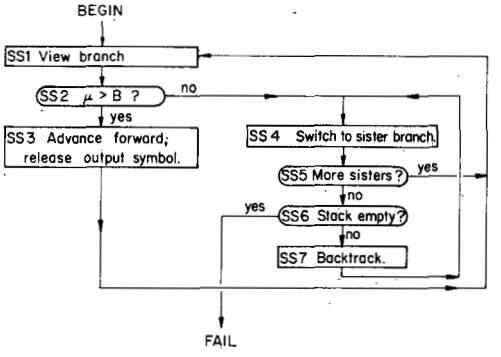

Fig. 2. Single stack algorithm.

B. The Two-Cycle Algorithm

Algorithms similar to the single stack procedure search depth-first, backtracking only when the path falls below the discard criterion. Another method which uses the same stack structure is to search *all* paths lying above the criterion and having length $l \leqslant L$. Of these, only a much smaller set of "good" paths, say those with metric $\mu \geqslant A$, are saved for later attention. We have called this procedure the two-cycle algorithm [5].

The two-cycle algorithm illustrates a nonsorting search that is not purely depth first. Such a mixed-regime search may be better in a given application. Table II lists some complexity data for this algorithm.

V. Sorting Algorithms

Sorting schemes compare paths on the basis of metric in order to decide which to extend and which to delete. These algorithms view fewer branches than nonsorting algorithms, but the cost of sorting is often very high.

A. The Stack Algorithm

As mentioned before, the stack algorithm is based not on a stack, but on a list of code tree paths. The usual view is that this is an ordered list; the next path extended is always the best in terms of μ, and sufficient worst paths are deleted to keep the list at length S paths. An alternate view is that new paths are simply appended, and that the list is probed for its best entry prior to each extension and for its worst entry prior to each deletion. The cost is similar in either case, and we shall take as the defining attributes for the stack algorithm simply a single list and metric-first extensions and deletions.

Since a path's metric indicates the likelihood that the best path in the code tree lies ahead of it, it comes as no surprise that this metric-first procedure appears to find a path at a given metric level with the least node computation of any scheme (see [1] or [2]). Despite this, the space–time cost of the stack algorithm seems to exceed that of any other scheme.

In the stack algorithm's list, paths vary in length. Once the

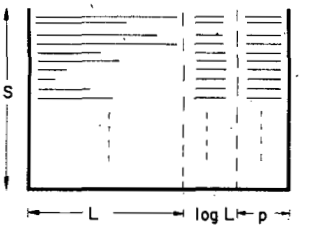

Fig. 3. Example of stack algorithm list, showing paths, length indicators, and metrics. The top path is about to penetrate a new depth, causing an ambiguity check; the fourth path will be deleted if its earliest symbol does not pass.

algorithm reaches a steady state, an ambiguity check must be performed whenever the length of a path in storage exceeds L. One can show that this occurs whenever a tree depth is reached for the first time. Fig. 3 shows a list data structure in which each entry consists of three subwords, a path metric (precision p), an indication of path length, and a path map. The path maps are left justified, and to find an end node, the length subword must be consulted. All paths end on the left at a point L branches before the deepest tree penetration; during an ambiguity check all these earliest branch symbols must be checked to see if they agree with the symbol released as output.

Algorithm SA (The Stack Algorithm):

SA0) Obtain root node.

SA1) (New penetration?) If the next extension will visit a depth for the first time, continue to SA2; else go to SA3.

SA2) (Ambiguity check.) Perform *ambiguity check* and *release output* symbol.

SA3) (View branches.) *Extend b paths* from the best path in the list. *Delete* this path.

SA4) (Reorder list.) Order into the list the b new paths.

SA5) (Delete paths.) *Delete paths,* worst first, until S remain. Go back to SA1.

A flow chart appears in Fig. 4.

Most of the cost of the algorithm resides in step SA4. This reorder step can be done simply by reading one RAM into another until the place appears for the new path; the new path is then written and the reading continues until the second RAM is full. An alternative that requires less storage is to have just one RAM like Fig. 3, plus a list of pointers of size S log S; to reorder the list, only these pointers are manipulated. The ordering may also be done by push-down stacks. All three methods have the same asymptotic space cost, $S(L + \log_b L + p)$ b-ary symbols.

The time cost is an average S accesses per branch viewed in all three cases, although the cost of an access may be "cheaper" in one of the implementations. SA1 and SA3 form a constant overhead, and the ambiguity check SA2 occurs only once with each depth penetration rather than each path extension, and so has a lower order cost.

Total space–time complexity is thus about $S^2(L + \log L + p)E[C_{SA}]$, or since p is small, $\cong L\, S^2 E[C_{SA}]$ asymptotically, where C_{SA} is the node computation of the stack algorithm.

Unless $E[C_{SA}]$ is very small, the added S^2 factor will make this cost much larger than that of the nonsorting algorithms. All research studies on the stack algorithm have reported computational difficulties in its use. Jelinek [1] suggests a chained storage scheme for the path maps which would eliminate the blank regions in Fig. 3, but it does not change the asymptotic cost [17]. He also suggests alternative algorithms, a combining of the Fano and stack algorithms [1, pp. 682 ff.], and a bucket algorithm. The latter is a basically new scheme to which we return in Section V-C.

B. The Merge Algorithm

The cost of the stack algorithm can be greatly reduced while still retaining the strictly metric-first feature if two ordered

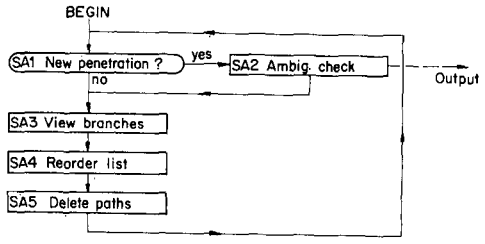

Fig. 4. The stack algorithm.

lists are jointly maintained. The strategy will be to conduct a metric-first search with a short auxiliary list and occasionally merge the list into the main list. We call such a procedure the merge algorithm.

Consider a main list of size S entries and an auxiliary list of size T. A succession of T new paths are searched metric first just as in the stack algorithm, requiring about $T^2/2$ comparisons between paths. After this, the two lists are merged. The merge operation requires $S + T$ comparisons [12, ch. 5], and the worst T paths are deleted in the process. Per branch viewed, the number of comparisons is about $S/T + 2 + T/2$, and minimizing this expression over the choice of T, we get an optimal auxiliary list size of about $\sqrt{2S}$. Furthermore, the number of comparisons per branch is only $\sqrt{2S} + 2$, instead of the S required by the stack algorithm. Total storage increases by the fraction $\sqrt{2S/S}$, an asymptotically insignificant factor.

To be sure that it proceeds metric-first, the algorithm must extend the best path at the top of either list, and this will require a single comparison per path extended. The ambiguity check must also reference both lists. Both of these provisos make the above estimates slight overbounds. Otherwise the merge algorithm is identical to the stack algorithm.

Algorithm MG (The Merge Algorithm):

MG0) Obtain root node.

MG1) (New penetration?) If the next extension will visit a depth for the first time, continue to MG2; else go to MG3.

MG2) (Ambiguity check.) Perform *ambiguity check* on both lists and *release output symbol.*

MG3) (View branches.) Compare the top paths in the two lists and *extend b paths* from the better. *Delete* this path.

MG4) (Reorder aux. list.) Order into the auxiliary list the b new paths.

MG4a) (Aux. list full?) If the auxiliary list cannot accept another b paths, continue to MG5; else go back to MG1.

MG5) (Merge and delete.) Merge the auxiliary and main lists into the main list, and *delete paths* that overflow. Go back to MG1.

In computing the cost of algorithm MG, define a merge cycle to include the branch viewings required to fill the auxiliary list. During a cycle, MG3 and MG4 cause about T accesses to a storage of size LS and about $T + T^2/2$ accesses to a size LT storage, where we have idealized the entry length as L; at the cycle's end, MG5 causes S accesses to LS, T to LT, and $S + T$ to $L(S + T)$. The total space-time product is $L(T^3/2 + 4ST + 4T^2 + 2S^2)$ per cycle. Dividing by T gives a tight overbound to the cost per branch viewed, since more than T branches will be viewed if some nodes are drawn from the auxiliary list.

Minimizing this expression, we get an optimal list size of about $T = (\sqrt{2S})^{2/3}$ for large S. The minimized cost is about $(2.38)LS^{4/3}$ per branch viewed. As an example consider a stack algorithm with $S = 100$. Its cost per branch viewed is on the order of 20 000 L, while the merge algorithm, by adding a second list of length 27, reduces this cost to 1100 L. It is interesting to observe that the comparisons-based optimization specifies a second list of only 16.

Thus far not considered is the ambiguity check (MG2), which costs $L(S^2 + T^2)$ per branch released. Inserting this term gives a total product cost of about

$$(L + \log L + p)[(2.38)S^{4/3}E[C_{SA}] + (S + (\sqrt{2}S)^{2/3})^2]$$

access-symbols/branch released (2)

which is asymptotically $L(S^{4/3}\ E[C_{SA}] + S^2)$. We have used the fact that the node computation will be close to that of the stack algorithm. It is no longer clear that the cost of ambiguity checks is insignificant, although it is likely that it is, since $E[C_{SA}]$ is probably of order S or larger.

Many variations have been proposed on the basic theme of side lists. The multiple stack algorithm of Chevillat and Costello [13] utilizes a succession of side lists to reduce sorting in the main list, although the search is not strictly metric-first and merging is not used between lists. Haccoun and Ferguson [14] propose extending K paths at once rather than one, before each ordering. The idea of merging seems to have originated with Mohan [9]. An extension of this idea leads to the metric-first procedure having the least possible asymptotic cost [17].

C. The Bucket Algorithm

A reduction in cost is easy to obtain by resort to schemes that are not strictly metric-first. Jelinek [1] proposed that the range of the metric be quantized into segments called "buckets" and incoming paths be sorted only into the correct bucket. He called this procedure the bucket algorithm.

The bucket approach has a number of subtle variations which have markedly different asymptotic cost. Suppose there are kS buckets, where k is a small but fixed fraction and each bucket is of fixed size; then the metric range for each bucket continuously varies, and the number of comparisons required to find the proper bucket for a path is asymptotically still of order S, just as it was for the stack algorithm. An alternative is to let the buckets vary in size instead of metric range. In what follows we show that such an algorithm exists whose time cost for insertion is overbounded by a constant, regardless of S.

Both these variants are only roughly metric-first. A pure metric-first algorithm with buckets can be obtained by ordering just the top bucket, as several authors have proposed. But even if the fraction of buckets k tends to zero as $S \uparrow \infty$, one can show that comparisons of at least order $S^{1/3}$ are required [9]. With moderate, practical storage S, the issues are less clear, and modification to the basic algorithm should be useful. Two principles are clear, however: bucket size should not be fixed and whether the algorithm is perfectly or only roughly metric first has a critical asymptotic effect.

We find it useful to view the bucket procedure as a *hashed search* for the best path. Our object is to draw attention away from sorting, which the procedure does not really do, and toward a definition of a hashing function $h(\ )$, which is the real crux of the algorithm. It is best to construe sorting as perfect sorting, a much more costly procedure.

Paths are places in buckets according to the function $h(\ )$, which maps the metric μ to an integer β in the set $\{1, \cdots, Q\}$, where β denotes which one of Q buckets. Paths are extended from the best bucket and continuously deleted from the worst, to maintain a total of S paths. $h(\ )$ identifies the quantization segment of μ. Little is known about the optimal definition of $h(\ )$ as S grows very large; it is doubtful that h is simply a uniform quantization, and it seems likely that the storage must be rehashed (i.e., h redefined) from time to time to obtain optimal performance. The cost of the latter may not be insignificant [12, sect. 6.4].

Since the bucket algorithm cost will depend critically on the cost of maintaining the buckets, we investigate now a construction that does this. To realize the advantage of the al-

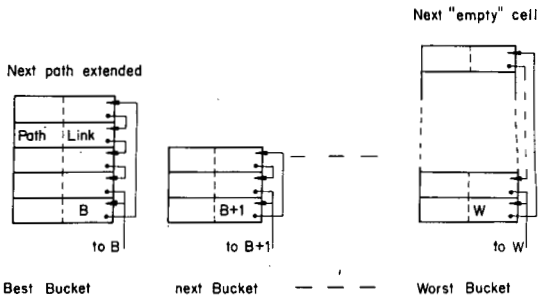

Fig. 5. Chained storage of buckets for bucket algorithm. W = worst bucket, B = best bucket. An empty bucket contains only the bottom cell.

gorithm, the construction must insert a path with a fixed number of accesses, independent of S.

Let the storage consist of cells containing a path (map, length, μ) and a link address to another cell. Buckets are constructed as shown in Fig. 5. Cells $\{1, \cdots, Q\}$ are bucket heads and are permanently reserved in storage; their links point to the most recently added cell in the bucket. The other cells each link to the cell which arrived before them. Path extensions take place always from the best bucket, and new cells are obtained from the worst bucket, which functions as a list of unoccupied (or expendable) cells.

We can now redefine some of the basic operations of III in terms of this chaining regime. The variables BSTBKT and WSTBKT give the location of the best and worst bucket heads.

Extend Path:

i) Find top of best bucket by accessing cell BSTBKT.

ii) Remove top path, form extensions, and *add path(s)* to buckets.

iii) Link top cell to empty cell list.

iv) Link cell BSTBKT to next-to-top cell.

v) If best bucket empty, BSTBKT ← BSTBKT + 1 (repeat as necessary).

Add Path:

i) Compute $h(\mu)$; if $h(\mu) >$ WSTBKT, WSTBKT ← h.

ii) Find top of bucket $h(\mu)$ by accessing cell $h(\mu)$.

iii) *Obtain empty cell.*

iv) Replace link field of cell $h(\mu)$ with address of this cell.

v) Place new path in this cell; place old top cell address in link field.

Obtain Empty Cell:

i) Access bucket WSTBKT; if empty, WSTBKT ← WSTBKT − 1 (repeat as necessary).

ii) Take cell pointed to by link field of cell WSTBKT; replace cell WSTBKT link field with address of next-to-top cell.

Ambiguity Check:

i) View each path.

ii) Link ambiguous paths to worst bucket, using *Add Path* iv and v.

Algorithm BA (The Bucket Algorithm):

BA0) Place root node in bucket 1. Set BSTBKT ← 1 and WSTBKT ← Q. Link all cells except the first Q to bucket Q.

BA1) (New penetration?) If the next generation will visit a depth for the first time, continue to BA2; else go to BA3.

BA2) (Ambiguity check.) Perform *ambiguity check* and *release output symbol.*

BA3) (View branches.) *Extend b paths* from top path of best bucket. *Delete* this path.

BA4) (Update buckets.) *Add paths* to proper buckets, using $h(\ )$. Go back to BA1.

A flow chart of algorithm BA will be similar to Fig. 4, except that there is no step SA5 to delete paths.

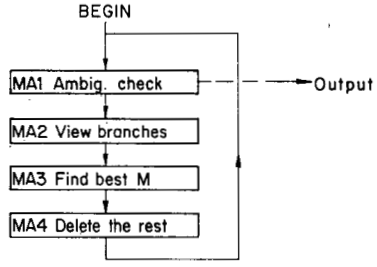

Fig. 6. The M-algorithm.

The cost of algorithm BA is

time cost: k' accesses/branch viewed, plus

$k''S$ accesses/branch released, for ambig. ch.

space cost: $S(L + \log L + p + \log_b S)$ b-ary symbols.

We have assumed that Q is small compared to S. The $\log_b S$ term accounts for the link field; k' depends on the precise sequence of reads and writes in BA3 and BA4, but is close to 5, and k'' is less than 2. Asymptotically, one gets

$$\cong LS(E[C_{SA}] + S) + H \quad \text{access symbols/branch released} \tag{3}$$

for the total space–time cost, assuming the node computation is close to that of the stack algorithm. A factor H takes account of the hash function cost. We have deleted the space cost term $\log_b S$ in (3) on the assumption that L dominates $\log_b S$. Of the three metric-first procedures we have studied, the bucket algorithm has the lowest order dependence on S—only S instead of $S^{4/3}$ or S^2.

D. The M-Algorithm

We conclude with breadth-first sorting algorithms. In general, such a procedure views all the branches at depth l that it will ever view, deletes paths according to some criterion, and then moves on to the next depth. The M-algorithm deletes all paths except a fixed number M. Breadth-first searches are synchronous (that is, all paths have the same length) and they are effective at low intensities of searching; they are thus good candidates for practical application [10], [16].

In its specific operation, the M-algorithm moves forward by extending the M paths it has retained to form bM new paths. All the terminal branches are compared to the input data corresponding to this depth, metrics computed, and the $(b-1)M$ poorest paths deleted. The heart of the algorithm is the sorting procedure which deletes these paths.

Algorithm MA (The M-Algorithm):

MA0) Obtain root node.

MA1) (Ambiguity check.) Perform *ambiguity check* and *release output symbol.*

MA2) (View branches.) *Extend b paths* from each retained path; save these in a list.

MA3) (Find best M.) Order the list to find the best M paths.

MA4) (Delete the rest.) *Delete* the remaining paths. Go back to MA1.

Observe that MA1 may reduce the number of retained paths below M, so that MA4 may delete fewer than $(b-1)M$. A flow chart appears in Fig. 6.

We are indebted to a reviewer for observing that the time cost of the M-algorithm is only of order S. Let $V_t(n)$ denote the number of comparisons required to find the tth largest of

n elements. Then algorithms exist [12, pp. 216-217] for which

$$E[V_t(n)] = n + t + f(n) \quad \text{where } \lim_{n\to\infty} f(n)/n = 0.$$

Using such an algorithm, one can find the Mth best path out of bM paths with only

$$E[V_M(bM)] = bM + M + f(bM)$$

comparisons. Once the metric of the Mth best path is known, one can choose the M best paths with at most bM comparisons.

The M-algorithm's time cost is then (with $S = M$), $(2b + 1)S + f(bS)$ comparisons per branch released. Hence, the asymptotic space-time product cost of the M-algorithm is

$$\cong LS^2 \text{ access symbols/branch released.}$$

Another breadth-first procedure has been suggested by Simmons and Wittke [18] for channel decoding. Their method deletes all paths at a given depth whose distance from the received sequence exceeds some constant c_0. Thus, a varying number of paths are retained at each level but the procedure is still purely breadth-first. Excellent results, free of erasures, were reported for an application to phase modulation codes.

VI. Some Implications

To choose the best algorithm for a given situation, one must determine the combination of L, S, and the node computation that optimizes the cost function for the desired encoding distortion or error probability. This is a most difficult task. Experimentation has been done, and some conclusions are possible. The results generally favor algorithms that do not sort, and point away from metric-first searching.

We first consider sequential source encoding. Table III reproduces simulations done by the authors for the stack, M-, two-cycle, and single stack algorithms used with random tree codes. The source is the binary letter, i.i.d. source with "Hamming" distortion measure (unit penalty for mismatched letters, zero penalty otherwise), the encoder rate is 1/2 output bit/source bit, and all the algorithms achieve an average distortion of about 0.125, 15 percent above the distortion-rate function. The formulas from Table II are used to evaluate cost, and because of the asymptotic nature of these and the uncertainties associated with simulation, only orders of magnitude are significant in the results. Still, it is clear the two nonsorting algorithms have greatly reduced cost compared to the stack or M-algorithms under either the space-time or space-plus-time cost evaluation. Even the merge and bucket algorithms (borrowing the same L, S, and $E[C]$ as the stack algorithm) fall well short, and in fact perform only as well as the M-algorithm.

Turning now to sequential channel decoding, we consider the case of rate 1/2 convolutional codes and a binary symmetric channel with crossover probabilities in the range 0.02-0.045. The 0.045 error rate corresponds to $R_{comp} = 1/2$; for higher error rates than this, the node computation lacks a first moment. Jelinek [1, pp. 680 ff.] reports $E[C_{SA}]$ increasing from 2 up to about 4 through this error range, while Chevillat and Costello [13, p. 1469] report an $E[C]$ of about 2.8 at error probability 0.025 for their multiple stack algorithm. The storage factor S reported by both lies in the range 1000-3000. The Fano algorithm, on the other hand, requires up to seven times as much node computation as the stack schemes. Nonetheless, if we assume that all algorithms have about the same L, the cost evaluation of the Fano algorithm is less by a factor of 10^3.

It is of interest to compare the cost of these methods with that of the Viterbi algorithm, using a code which achieves about the same decoded error probability (10^{-5} – 10^{-4}). The space complexity of this algorithm is about Lb^K, where K is the constraint length of the code, while b accesses to this storage are required per branch released as output. Using hard decisions, a K of at least 8 is needed to achieve the above error range at a crossover probability of 0.025 [13, p. 1469]. $K = 4$ is sufficient for soft decisions. At $K = 8$, the Viterbi algorithm space-time cost is on the order of 512 L, versus only about 3 L for the Fano algorithm; a similar difference holds for space-plus-time cost. The Viterbi algorithm is still much cheaper than the sorting procedures. For decoded error rates below 10^{-5}, Viterbi coding becomes very costly, but it does have the advantages of being synchronous and erasure free.

TABLE III
EVALUATION OF COST FOR CERTAIN ALGORITHMS, TAKEN FROM EXPERIMENTAL DATA. BINARY i.i.d. SOURCE WITH HAMMING DISTORTION, RATE 1/2 OUTPUT BIT PER INPUT BIT, ENCODED DISTORTION 0.125 (SHANNON LIMIT = 0.110).

	Branches Viewed E[C]	Paths Stored S	Space•Time Cost	Space+Time Cost
Stack	200	>500	10^{10}**	200K
M-	500	250	7×10^7	51K
2-Cycle	1000	1*	200K	8500
Single Stack	1500	1	300K	1700

L = 200-300, all cases (200 used for cost).

* About 150 paths of average length 50 were kept in the save stack.

** 2×10^8 with Merge Alg.; 7×10^7+ h with Bucket Alg.

Erasure of output data caused by noise-induced computational overload can occur in many decoding search algorithms and thwarting this problem is a factor in their design. Our cost measures make no explicit mention of erasures, but to the extent that erasures stem from exhaustion of resources, the estimates of Table II indicate susceptibility to erasures.

Research into the application of search procedures to sequence estimation for band-limited channels has only just begun. There should be interesting results here, since the Viterbi algorithm estimator would seem to have limited applicability to severely band-limited channels.

References

[1] F. Jelinek, "A fast sequential decoding algorithm using a stack," *IBM J. Res. Develop.*, vol. 13, pp. 675–685, Nov. 1975.

[2] J. B. Anderson, "A stack algorithm for source coding with a fidelity criterion," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 211–226, Mar. 1974.

[3] K. Sh. Zigangirov, "Some sequential decoding procedures," *Probl. Peredach. Inform.*, vol. 2, no. 4, pp. 13–25, 1966.

[4] R. G. Gallager, "Tree encoding for symmetric sources with a distortion measure," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 65–76, Jan. 1974.

[5] J. B. Anderson and F. Jelinek, "A 2-cycle algorithm for source coding with a fidelity criterion," *IEEE Trans. Inform. Theory*, vol. IT-19, pp. 77–92, Jan. 1973.

[6] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimal decoding algorithm," *IEEE Trans. Inform. Theory*, vol. IT-13, pp. 260–269, Apr. 1967.

[7] J. Uddenfeldt and L. H. Zetterberg, "Algorithms for delayed encoding in delta modulation with speech-like signals," *IEEE Trans. Commun.*, vol. COM-24, pp. 652–658, June 1976; see also "Adaptive delta modulation with delayed decision," *IEEE Trans. Commun.*, vol. COM-22, pp. 1195–1198, Sept. 1974.

[8] R. J. Dick, T. Berger, and F. Jelinek, "Tree encoding of Gaussian

sources," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 332–336, May 1974.
[9] S. Mohan, "Analyses and cost evaluation of code tree search algorithms," Ph.D. dissertation, McMaster Univ., Hamilton, Ont., Canada, Nov. 1979.
[10] J. B. Anderson and C.-W. P. Ho, "Architecture and construction of a hardware sequential encoder for speech," *IEEE Trans. Commun.*, vol. COM-25, pp. 703–707, July 1977.
[11] C. R. Davis and M. E. Hellman, "On tree coding with a fidelity criterion," *IEEE Trans. Inform. Theory*, vol. IT-21, pp. 373–378, July 1975.
[12] D. E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
[13] P. R. Chevillat and D. J. Costello, Jr., "A multiple stack algorithm for erasure-free decoding of convolutional codes," *IEEE Trans. Commun.*, vol. COM-25, pp. 1460–1470, Dec. 1977.
[14] D. Haccoun and M. J. Ferguson, "Generalized stack algorithms for decoding convolutional codes," *IEEE Trans. Inform. Theory*, vol. IT-21, pp. 638–651, Nov. 1975.
[15] S. Mohan and J. B. Anderson, "Speech encoding by a stack algorithm," *IEEE Trans. Commun.*, vol. COM-28, pp. 825–830, June 1980.
[16] S. G. Wilson and S. Husain, "Adaptive tree encoding of speech at 8000 bits/s with a frequency-weighted fidelity criterion," *IEEE Trans. Commun.*, vol. COM-27, pp. 165–170, Jan. 1979.
[17] S. Mohan and J. B. Anderson, "Computationally optimal metric-first code tree search algorithms," to be published.
[18] S. J. Simmons and P. H. Wittke, "Low complexity decoders for bandwidth efficient digital phase modulations," presented at the 11th Queen's Symp. Commun. Signal Processing, Queen's Univ., Kingston, Ont. Canada, May 30–June 2, 1982; also available as S. J. Simmons, M.Sc. thesis, Dep. Elec. Eng., Queen's Univ., 1982.
[19] G. P. Ashkar and J. W. Modestino, "The contour extraction problem with biomedical applications," *Comput. Graphics Image Processing*, vol. 7, pp. 331–355, 1978.
[20] J. W. Modestino, V. Bhaskaran, and J. B. Anderson, "Tree encoding of images in the presence of channel errors," *IEEE Trans. Inform. Theory*, vol. IT-27, pp. 677–697, Nov. 1981.
[21] F. Jelinek, L. R. Bahl, and R. L. Mercer, "Design of a linguistic statistical decoder for the recognition of continuous speech," *IEEE Trans. Inform. Theory*, vol. IT-21, pp. 250–256, May 1975.
[22] H.-M. Hang and J. W. Woods, "Near merging of paths in suboptimal tree searching," *IEEE Trans. Inform. Theory*, to be published.

John B. Anderson (M'72–SM'82) was born in Canandaigua, NY, in 1945. He received the B.S., M.S., and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1967, 1969, and 1972, respectively.

From 1972 to 1980 he was a member of the Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ont., Canada, except for the 1978–1979 academic year, when he served as Visiting Associated Professor at the Department of Electrical Engineering and Computer Science, University of California at Berkeley. At McMaster, he was a founding member of its Communications Research Laboratory. Since 1981 he has been an Associate Professor in the Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY. His research interests include the properties of searching and coding algorithms, methods of bandwidth-efficient coding, and the practical application of these to transmission of data, speech, and pictures. He has authored a number of publications in these areas and served as consultant to a number of organizations in different countries.

Dr. Anderson was a member of the IEEE Information Theory Group Board of Governors from 1980 to 1982. He coordinated liaison between the Board and the IEEE Communications Society from 1979 to 1981, and served as Co-Chairman of the 1983 IEEE International Symposium on Information Theory, St. Jovite, P.Q., Canada. He has been a Guest Editor of the IEEE TRANSACTIONS ON COMMUNICATIONS and currently serves as Associate Editor for Coding Techniques for the IEEE TRANSACTIONS ON INFORMATION THEORY.

Seshadri Mohan (S'76–M'80) received the B.E. (honours) degree from the University of Madras, Madras, India, in 1972, the M.Tech. degree from the Indian Institute of Technology, Kanpur, India, in 1974, and the Ph.D. degree from McMaster University, Hamilton, Ont., Canada, in 1979.

From 1974 to 1975 he was an Assistant Systems Analyst and Programmer with Tata Consultancy Services, India. In 1979, he joined Bell Laboratories, Holmdel, NJ, as a member of the Technical Staff, where he worked on switched network performance measurement and characterization. In 1980, he joined the Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI, as an Assistant Professor. Since September 1983, he has been an Associate Professor of Electrical and Computer Engineering with Clarkson College of Technology, Potsdam, NY. His present research interests include the design and architecture of sequential coding algorithms and the application of these algorithms to speech coding and data transmission.

Dr. Mohan is a member of the Association for Computing Machinery.