SECTION 6. A FASTER COMPRESSOR

A2 has very fast expansion with a small storage requirement, but, even though compression has constant amortized time, it is 5 times slower than expansion. **A1** and **A2** are most appropriate in applications where compression speed is not critical and where the performance of the expander needs to be optimized, such as the mass release of software on floppy disks. However, in applications such as file archiving, faster compression is needed. For this reason, we have developed the **B1** and **B2** methods described here, which use the same encodings as **A1** and **A2**, respectively, but compute window displacement differently. Copy codewords are restricted to start at the beginning of the yth previous codeword or literal character emitted; they can no longer address every earlier character, but only those where literal characters occurred or copy codewords started; we refer to displacements computed this way as "compressed displacements" throughout. Copy length is still measured in characters, like **A1**. By inserting this level of indirection during window access, compression speed typically triples, though expansion and the rate of adaptation are somewhat slower.

With "compressed displacements," suffix pointers and update propagation are unnecessary and a simpler PATRICIA tree can be used for the dictionary. Entries are made in the tree only on codeword boundaries, and this can be done in linear time by starting at the root on each iteration. It is useful to create an array of permanent nodes for all characters at depth 1. Since copy codewords of length 1 are never issued, it doesn't matter that some permanent nodes don't correspond to any window character. Each iteration begins by indexing into this node array with the next character. Then hash table lookups and arc character comparisons are used to descend deeper, as in **A1**. The new window position is written into nodes passed on the way down, so update propagation is unnecessary.

In short, the complications necessary to achieve constant average time per source character with **A2** are eliminated. However, one new complication is introduced. In the worst case, the 16,384 window positions of **B2** could require millions of characters, so we impose a limit of 12 x 16,384 characters; if the full window exceeds this limit, leaves for the oldest window positions are purged from the tree.

Because of slower adaptation, **B2** usually compresses slightly less than **A2** on small files. But on text and program source files, it surpasses **A2** by 6% to 8% asymptotically; the crossover from lower compression to higher occurs after about 70,000 characters! **A2** codewords find all the near-term context, while **B2** is restricted to start on previous codeword boundaries but can consequently reach further back in the file. This gives **B2** an advantage on files with a natural word structure, such as text, and a disadvantage on files where nearby context is especially important, such as scanned images.

We also tried variations where the tree is updated more frequently than on every codeword boundary and literal character. All variations up to and including **A2** can be implemented within the general framework of this method, if speed is not an issue. For example, we found that about 1% higher compression can be achieved by inserting another compressed position between the two characters represented by each length 2 copy codeword and another 0.5% by also inserting compressed positions after each character represented by length 3 copy codewords. However, because these changes slow compression and expansion we haven't used them.

SECTION 7. IMPROVING THE COMPRESSION RATIO

In section 6 we considered ways to speed up compression at the cost of slower adaptation and expansion. In this section we will explore the other direction: improving the compression ratio with a slight cost to the running time of the algorithm.

When a string occurs frequently in a file, all the methods we have considered so far waste space in their encoding; when they are encoding the repeating string, they are capable of specifying the copy displacement to multiple previous occurrences of the string, yet only one string needs to be copied. By contrast, the data structures we have used do not waste space. The repeating strings share a common path near the root. If we base the copy codewords directly on the data structure of the dictionary, we can improve the compression ratio significantly. (This brings us closer to the second style of Ziv and Lempel's textual substitution work [ZL 78] [MW 84] [J 85], where a dictionary is maintained by both the compressor and expander. However, since we still use a window and an explicit copy length coding, it is natural to view this as a modification of our earlier compressors, in the style of Ziv and Lempel's first textual substitution work.)

The **C2** method uses the same PATRICIA tree data structures as **B2** to store its dictionary. Thus it takes two pieces of information to specify a word in the dictionary: a node, and a location along the arc between the node and its parent (since PATRICIA tree arcs may correspond to strings with more than one character). We will distinguish two cases for a copy: if the arc is at a leaf of the tree, then we will use a *LeafCopy* codeword, while if the arc is internal to the tree will use a *NodeCopy* codeword. Essentially, those strings appearing two or more times in the window are coded with *NodeCopies*, avoiding the redundancy of **A2** or **B2** in these cases.

The **C2** encoding begins with a single prefix bit that is 0 for a *NodeCopy*, 1 for a *LeafCopy* or *Literal*.

For *NodeCopy* codewords, the prefix is followed by a node number in $[0..maxNodeNo]$, where *maxNodeNo* is the largest node number used since initialization; for most files tested, *maxNodeNo* is about 50% the number of leaves. Following the node number, a displacement along the arc from the node to its parent is encoded; for most *NodeCopy* codewords the incoming arc is of length 1, so no length field is required. If a length field is required, 0 denotes a match exactly at the node, 1 a displacement 1 down the arc from the parent node, etc. Rarely is the length field longer than one or two bits because the arc lengths are usually short, so all possible displacements can be enumerated with only a few bits. For both the node number and the incoming arc displacement, the trick described in Section 5 is used to eliminate wasted states in the field; that is, if v values must be encoded, then the smaller values are encoded with $\lfloor \log_2 v \rfloor$ bits and larger values with $\lceil \log_2 v \rceil$ bits.

LeafCopies are coded with unary progressions like those of **A2** or **B2**. A $(1, 1, 11)$ progression is used to specify the distance of the longest match down the leaf arc from its parent node, with 0 denoting a literal; this progression leads to a maximum copy length of 4094 bytes. Since another literal never occurs immediately after a literal of less than maximum literal length, the *LeafCopy* arc distance progression is shifted down by 1 when the preceding codeword was a literal (i.e., arc displacement 1 is coded as 0, 2 as 1, etc.) On a cross section of files from the data sets discussed later, distance down the leaf arc was highly skewed, with about half the arc displacements occurring one character down the leaf arc. Because of this probability spike at 1 and the rapid drop off at larger distances, the average length field is small. Following the length field, the window position is coded by gradually phasing in a $(10, 2, 14)$ unary progression exactly like **B2**'s.

Literals are coded by first coding a *LeafCopy* arc displacement of 0 and then using a $(0, 1, 5)$ unary progression for the literal length exactly like **B2**.

Unlike **A2** and **B2**, the expander for **C2** must maintain a dictionary tree exactly like the compressor's tree to permit decoding. Notice that this is not as onerous as it might seem. During compression, the algorithm must search the tree downwards (root towards leaves) to find the longest match, and this requires a hash table access at each node. By contrast, the expander is told which node was matched, and it can recover the length and window position of the match from the node. No hash table is required, but the encoding is restricted: a copy codeword must always represent the longest match found in the tree. In particular, the superior heuristic used by **B2** to choose between Literal and Copy codewords must be discarded; instead, when the longest match is of length 2 or more, a copy codeword must always be produced. With this restriction, the expander can reconstruct the tree during decoding simply by hanging each new leaf from the node or arc indicated by the *NodeCopy* or *LeafCopy* codeword, or in the case of literals, by hanging the leaf from the permanent depth 1 node for each literal character.

SECTION 8. EMPIRICAL STUDIES

In this section, we compare the five compression methods we have developed with other one-pass, adaptive methods. For most other methods, we do not have well-tuned implementations and report only compression results. For implementations we have tuned for efficiency, speed is also estimated (for our 3 MIP, 16-bit word size, 8 megabyte workstations). The execution times used to determine speed include the time to open, read, and write files on the local disk (which has a relatively slow, maximum transfer rate of 5 megabits per second); the speed is computed by dividing the uncompressed file size by the execution time for a large file.

We tested file types important in our working environment. Each number in the table below is the sum of the compressed file sizes for all files in the group divided by the sum of the original file sizes. Charts 1–3 show the dependency of compression on file size for all of the compression methods tested on the source code (SC) data set. The gray area in these charts shows the distribution of file sizes in the data set, and the numbers next to the labels are the total compression ratios, duplicating the SC column in the table below.

DATA SETS

SC Source Code. All 8-bit Ascii source files from which the boot file for our programming environment is built. Files include some English comments, and a densely-coded collection of formatting information at the end of each file reduces compressibility. The files themselves are written in the Cedar language. (1185 files, average size 11 Kbytes, total size 13.4 Mbytes)

TM Technical Memoranda. All files from a directory where computer science technical memoranda and reports are filed, excluding those containing images. These files are 8-bit Ascii text with densely-coded formatting information at the end (like the source code). (134 files, average size 22 Kbytes, total size 2.9 Mbytes)

NS News Service. One file for each work day of a week from a major wire service; these files are 8-bit Ascii with no formatting information. Using textual substitution methods, these do not compress as well as the technical memoranda of the previous study group, even though they are much larger and should be less impacted by startup transient; inspection suggests that the larger vocabulary and extensive use of proper names might be responsible for this. (5 files, average size 459 Kbytes, total size 2.3 Mbytes)

	Text			Binary		Fonts		Images		
Method	SC	TM	NS	CC	BF	SF	RCF	SNI	SCI	BI
H0	.732	.612	.590	.780	.752	.626	.756	.397	.845	.148
H1	.401	.424	.467	.540	.573	.380	.597	.181	.510	.101
KG	.751	.625	.595	.804	.756	.637	.767	.415	.850	.205
V	.749	.624	.595	.802	.756	.637	.766	.414	.850	.205
CW	.369	.358	.326	.768	.544	.516	.649	.233	.608	.106
MW1	.508	.470	.487	.770	.626	.558	.705	.259	.728	.117
MW2	.458	.449	.458	.784	.594	.526	.692	.270	.774	.117
UW	.521	.476	.442	.796	.638	.561	.728	.255	.697	.118
BSTW	.426	.434	.465	-	.684	--	.581	—	—	—
A1	.430	.461	.520	.741	.608	.502	.657	.351	.766	.215
A2	.366	.395	.436	.676	.538	.460	.588	.259	.709	.123
B1	.449	.458	.501	.753	.616	.505	.676	.349	.777	.213
B2	.372	.403	.410	.681	.547	.459	.603	.255	.714	.117
C2	.360	.376	.375	.668	.527	.445	.578	.238	.662	.105

Table 1. Comparison of Compression Methods.

CC Compiled Code. The compiled-code files produced from the SC data set. Each file contains several different regions: symbol names, pointers to the symbols, statement boundaries and source positions for the debugger, and executable code. Because each region is small and the regions have different characteristics, these files severely test an adaptive compressor. (1220 files, average size 13 Kbytes, total size 16.5 Mbytes)

BF Boot File. The boot file for our programming environment, basically a core image and memory map. (1 file, size 525 Kbytes)

SF Spline Fonts. Spline-described character fonts used to generate the bitmaps for character sets at a variety of resolutions. (94 files, average size 39 Kbytes, total size 3.6 Mbytes)

RCF Run-coded Fonts. High-resolution character fonts, where the original bitmaps have been replaced by a run-coded representation. (68 files, average size 47 Kbytes, total size 3.2 Mbytes)

SNI Synthetic Images. All 8 bit/pixel synthetic image files from the directory of an imaging researcher. The 44 files are the red, green, and blue color separations for 12 color images, 2 of which also have an extra file to encode background transparency; in addition, there are 6 other grey scale images. (44 files, average size 328 Kbytes, total size 14.4 Mbytes)

SCI Scanned Images. The red separations for all 8 bit/pixel scanned-in color images from the directory of an imaging researcher. The low-order one or two bits of each pixel are probably noise, reducing compressibility. (12 files, average size 683 Kbytes, total size 8.2 Mbytes)

BI Binary Images. CCITT standard images used to evaluate binary facsimile compression methods. Each file consists of a 148-byte header followed by a binary scan of 1 page (1728 pixels/scan line x 2376 scan lines/page). Some images have blocks of zeros more than 30,000 bytes long. Because these files are composed of 1-bit rather than 8-bit items, the general-purpose compressors do worse than they otherwise might. (8 files, average size 513 Kbytes, total size 4.1 Mbytes)

The special-purpose CCITT 1D and 2D compression methods reported in [HR 80] achieve, respectively, .112 and .064 compression ratios on these standard images when the extraneous end-of-line codes required by the facsimile standard are removed and when the extraneous 148-byte header is removed. The special-purpose CCITT 2D result is significantly more compact than any general purpose method we tested, and only **CW** and **C2** surpassed the 1D result.

MEASUREMENTS AND COMPRESSION METHODS

H0 and H1. These are entropy calculations made on a per file basis according to:

$$H_0 = -\sum_{i=0}^{n-1} P(x = c_i) \log_2 P(x = c_i). \tag{7}$$

$$H_1 = -\sum_{i,j=0}^{n-1} P(x = c_i)\, P(y = c_j | x = c_i) \log_2 P(y = c_j | x = c_i). \tag{8}$$

where x is a random symbol of the source, xy is a randomly chosen pair of adjacent source characters, and c_i ranges over all possible symbols. Because of the small file size, the curves in charts 1 to 3 drop off to the left. In theory, this small sampling problem can be corrected according to [B 59], but we have found it difficult to estimate the total character set size in order to apply these corrections. Nevertheless, chart 1 shows that **H0** is a good estimator for how well a memoryless (zero-order) compressor can do when file size is a large multiple of 256 bytes and **H1** bounds the compression for a first-order Markov method. (None of our files were large enough for **H1** to be an accurate estimator.)

KG and V. These adaptive methods maintain a Huffman tree based on the frequency of characters seen so far in a file. The compressor and expander have roughly equal performance. The theory behind the **KG** approach appears in [G 78] and [K 85]. The similar **V** method, discussed in [V 85], should get better compression during the startup transient at the expense of being about 18% slower. It is also possible to bound the performance of Vitter's scheme closely to that of a fixed non-adaptive compressor. Except on the highly compressible CCITT images, these methods achieve compression slightly worse than **H0**, as expected. But because of bit quantization, the compression of the CCITT images is poor—arithmetic coding would compress close to **H0** even on these highly compressible sources.

CW Based on [CW 84], this method gathers higher-order statistics than **KG** or **V** above (which we ran only on zero-order statistics). The method that Cleary and Witten describe keeps statistics to some order o and encodes each new character based on the context of the o preceding characters. (We've used $o = 3$, because any higher order exhausts storage on most of our data sets.) If the new character has never before appeared in the same context, then an escape mechanism is used to back down to smaller contexts to encode the character using those statistics. (We've used their escape mechanism A with exclusion of counts from higher-order contexts.) Because of high event probabilities in some higher-ordered contexts and the possibility of multiple escapes before a character is encoded, the fractional bit loss of Huffman encoding is a concern, so [CW 84] uses arithmetic encoding. We have used the arithmetic encoder in [WNC 87].

As Table 1 shows, **CW** achieves excellent compression. Its chief drawbacks are its space and time performance. Its space requirement can grow in proportion to file size; for example, statistics for $o = 3$ on random input could require a tree with 256^4 leaves, though English text requires much less. The space (and consequently time) performance of **CW** degrades dramatically on "more random" data sets like SNI and SCI. A practical implementation would have to limit storage somehow.

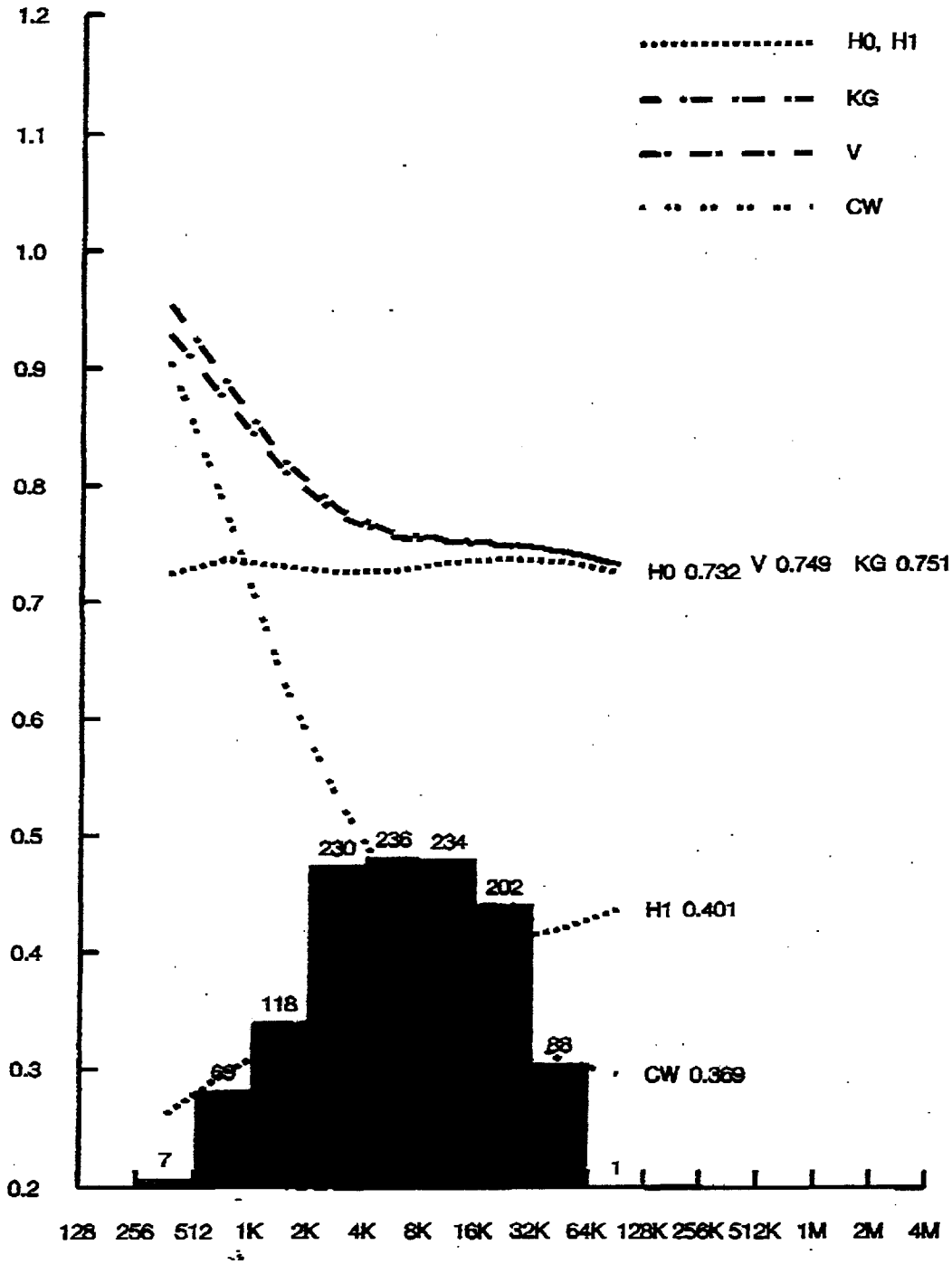

Chart 1. Compression vs. File Size, Data Set SC

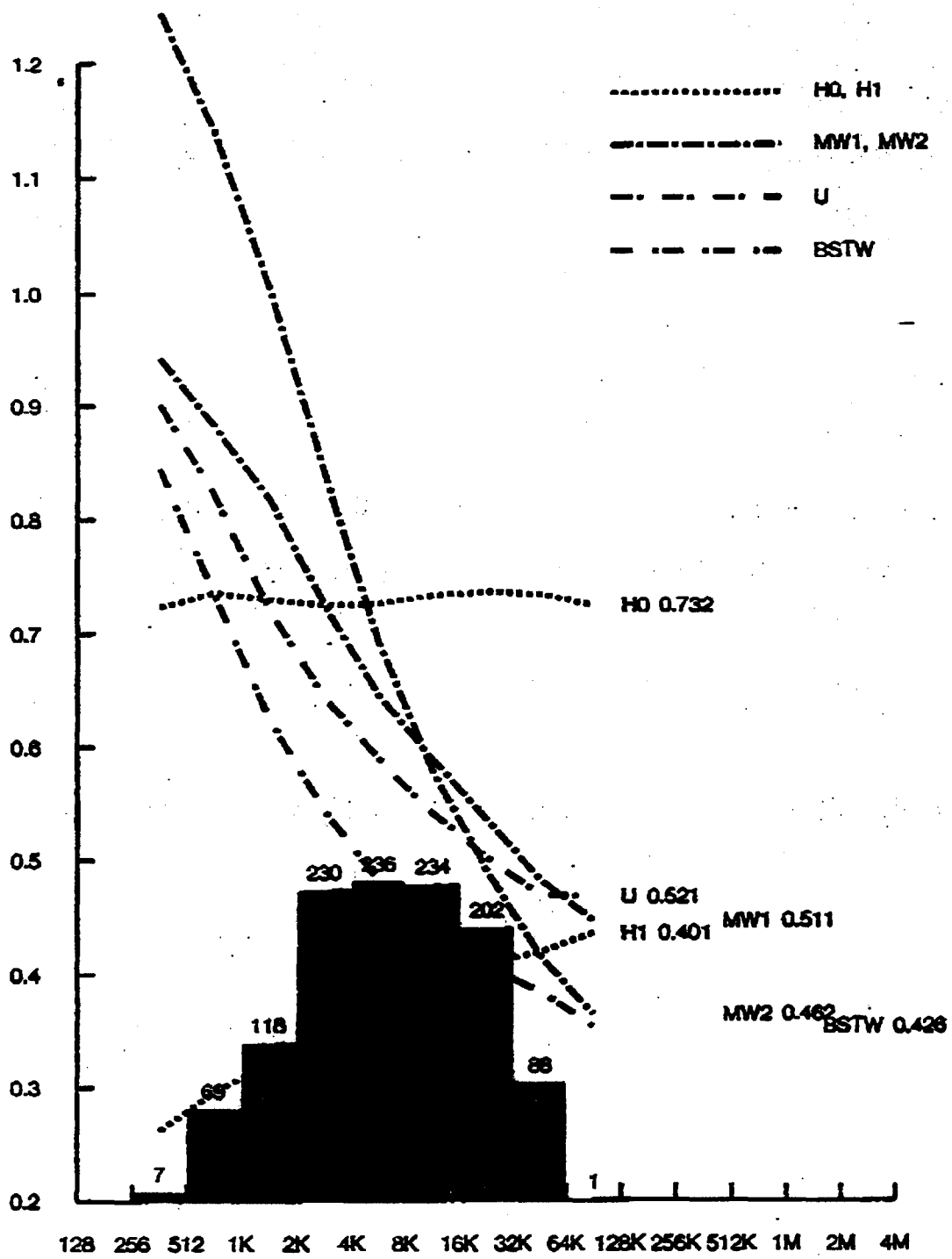

Chart 2. Compression vs. File Size, Data Set SC

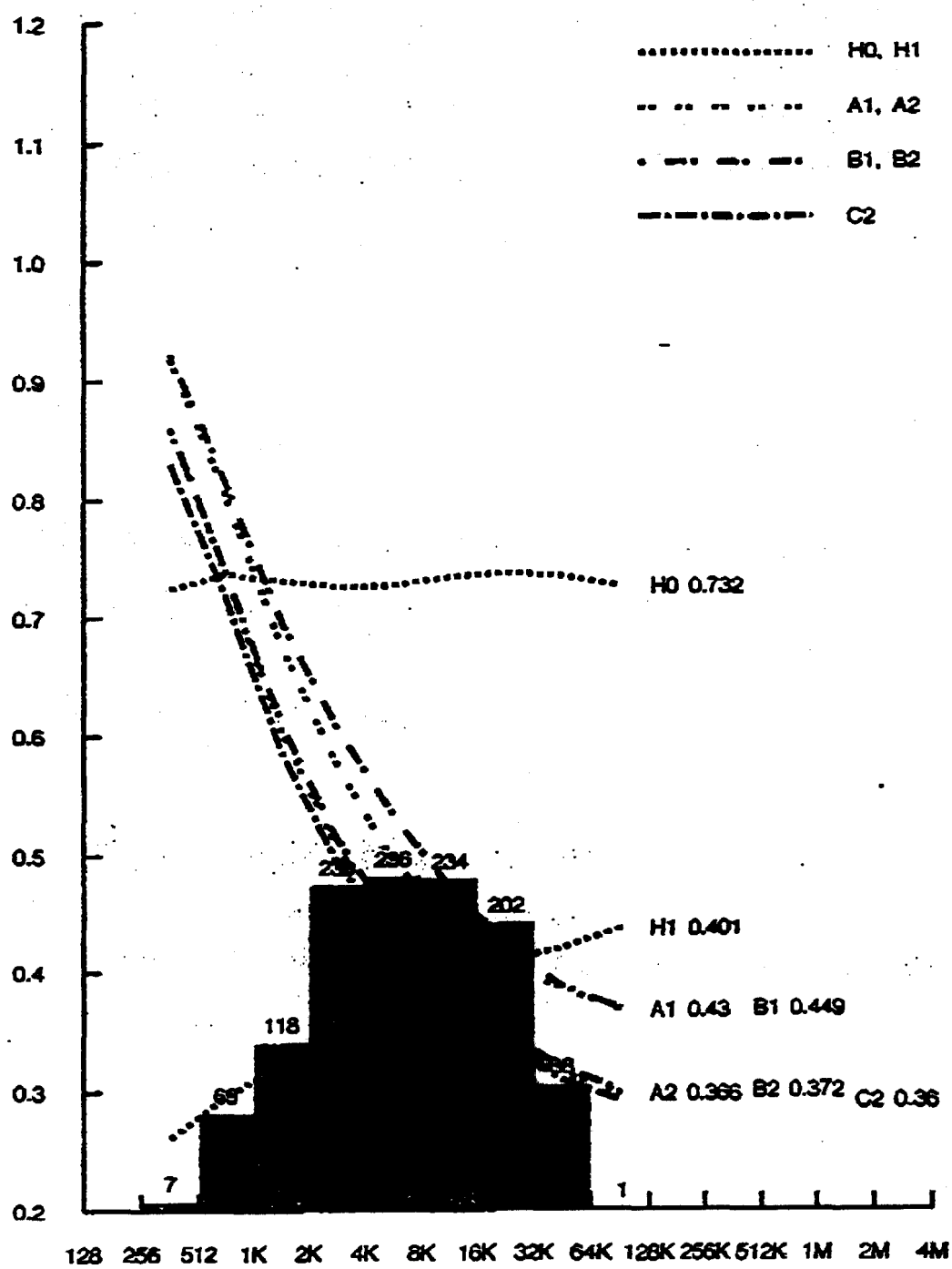

Chart 3. Compression vs. File Size, Data Set SC

Even on English, Bell, Cleary, and Witten estimate that Moffat's tuned implementation of **CW** is 3 times slower compressing and 5 times slower expanding than **C2** [BCW 88].

MW1. This method, described in [MW 84], is related to the second style of Lempel-Ziv compression, alluded to in the introduction. It uses a Trie data structure and 12-bit codes. Initially (and always) the dictionary contains 256 one-character strings. New material is encoded by finding the longest match in the dictionary, outputting the associated code, and then inserting a new dictionary entry that is the longest match plus one more character. After the dictionary has filled, each iteration reclaims an old code from among dictionary leaves, following a LRU discipline, and reuses that code for the new dictionary entry. The expander works the same way. **MW1** is simple to implement and is balanced in performance, with good speed both compressing and expanding (250,000 bits/sec and 310,000 bits/sec respectively). The original method used 12-bit codes throughout for simplicity and efficiency. However, our implementation starts by using 9-bit codewords, increasing to 10, 11, and finally to 12 bits as the dictionary grows to its maximum size; this saves up to 352 bytes in the compressed file size. On text and source code, Miller and Wegman determined that the 12-bit codeword size is close to optimal for this method.

MW2. One drawback of **MW1** is the slow rate of buildup of dictionary entries. If, for example, the word *abcdefghi* appears frequently in a document, then *ab* will be in the dictionary after the first occurrence, *abc* after the second, and so on, with the full word present only after 8 occurrences (assuming no help from similar words in the document). **A1** below, for example, would be able to copy the whole word *abcdefghi* after the first occurrence, but it pays a penalty for the quick response by having a length field in its copy codeword. The idea of **MW2** is to build dictionary entries faster by combining adjacent codewords of the **MW1** scheme. Longer words like *abcdefghi* are built up at an exponential rather than linear rate. The chief disadvantage of **MW2** is its increased complexity and slow execution. Our implementation follows the description in [MW 84] and uses an upper limit of 4096 dictionary entries (or 12-bit codewords). We did not implement the 9–12 bit phase-in that was used in **MW1** so the size-dependent charts underestimate **MW2**'s potential performance on small files.

UW. This is the Compress utility found in the Berkeley 4.3 Unix, which modifies a method described in a paper by Welch [W 84]; the authors of this method are S. Thomas, J. McKie, S. Davies, K. Turkowski, J. Woods, and J. Orost. It builds its dictionary like **MW1**, gradually expanding the codeword size from 9 bits initially up to 16 bits. The dictionary is frozen after 65,536 entries, but if the compression ratio drops significantly, the dictionary is discarded and rebuilt from scratch. We used this compressor remotely on a VAX-785, so it is difficult to compare its running time and implementation difficulties with the other methods we implemented. Nevertheless, because it does not use the LRU collection of codes, it should be faster than **MW1**. However, it has a larger total storage requirement and gets worse compression than **MW1** on most data sets studied.

BSTW. This method first partitions the input into alphanumeric and non-alphanumeric "words," so it is specialized for text, though we were able to run it on some other kinds of data as well. The core of the compressor is a move-to-front heuristic. Within each class, the most recently seen words are kept on a list (we have used list size 256). If the next input word is already in the word list, then the compressor simply encodes the position of the word in the list and then moves the word to the front of the list. The move-to-front heuristic means that frequently used words will be near the front of the list, so they can be encoded with fewer bits. If the next word in the input stream is not on the word list, then the new word is added to the front of the list, while another word is removed from the end of the list, and the new word must be compressed character-by-character.

Since the empirical results in [BSTW 85] do not actually give an encoding for the positions of words in the list or for the characters in new words that are output, we have taken the liberty of

using the V compressor as a subroutine to generate these encodings adaptively. (There are actually four copies of Vitter's algorithm running, one to encode positions and one to encode characters in each of two partitions.) Using an adaptive Huffman is slow; a fixed encoding would run faster, but we expect that a fixed encoding would slightly reduce compression on larger files while slightly improving compression on small files. We could not run BSTW for all of the data sets, since the parsing mechanism assumes human-readable text and long "words" appear in the other data sets. When the unreadable input parsed well, as in the case of run-coded fonts, the compression was very good.

A1. This is our basic method described earlier. It has a fast and simple expander (560,000 bits/sec) with a small storage requirement (10,000 bytes). However, the compressor is much slower and larger (73,000 bits/sec, 145,000 bytes using scan-from-leaf and update-to-root). The encoding has a maximum compression to 1/8 = 12.5% of the original file size because the best it can do is copy 16 characters with a 16-bit codeword.

Caveat: As we mentioned above, the running times reported include the file system overhead for a relatively slow disk. To provide a baseline, we timed a file copy without compression and obtained a rate of 760,000 bits per second. Thus, some of the faster expansion rates we report are severely limited by the disk. For example, we estimate that without disk overhead the A1 expander would be about twice as fast. On the other hand, removing disk overhead would hardly affect the compression speed of A1.

A2. This method, discussed in Section 5, enlarges the window to 16,384 characters and uses variable-width unary-coded copy and literal codewords to significantly increase compression. The running time and storage requirements are 410,000 bits/sec and 21,000 bytes for expansion and 60,000 bits/sec and 630,000 bytes for compression (using suffix pointers and percolated update).

B1. This method, discussed in Section 6, uses the A1 encoding but triples compression speed by updating the tree only at codeword boundaries and literal characters. The running time and storage requirements are 470,000 bits/sec and 45,000 bytes for expansion and 230,000 bits/sec and 187,000 bytes for compression.

B2. This method, discussed in Section 6, uses the same encoding as A2 but triples compression speed by updating the tree only at codeword boundaries and literal characters. The compressor and expander run at 170,000 and 380,000 bits/sec, respectively, and have storage requirements of 792,000 and 262,000 bytes.

C2. This method, discussed in Section 7, uses the same data structures as B2 but a more powerful encoding based directly upon the structure of the dictionary tree. Compression is about the same and expansion about 25% slower than B2; the compressor uses about the same storage as B2, but the expander uses more (about 529,000 bytes).

Table 1 highlights some differences between textual substitution methods like C2 and statistical methods like CW. (Time and space performance differences have been discussed earlier.) There are several data sets where these methods differ dramatically. On NS, CW is significantly better than C2. We believe that this is because NS shows great diversity in vocabulary: a property that is troublesome for textual substitution, since it cannot copy new words easily from elsewhere in the document, but this property is benign for CW, since new words are likely to follow the existing English statistics. On CC, for example, C2 is significantly better than CW. We believe that this is because CC contains several radically different parts, e.g. symbol tables, and compiled code. C2 is able to adjust to dramatic shifts within a file, due to literal codewords and copy addressing that favors nearby context, while CW has no easy way to rapidly diminish the effect of older statistics.

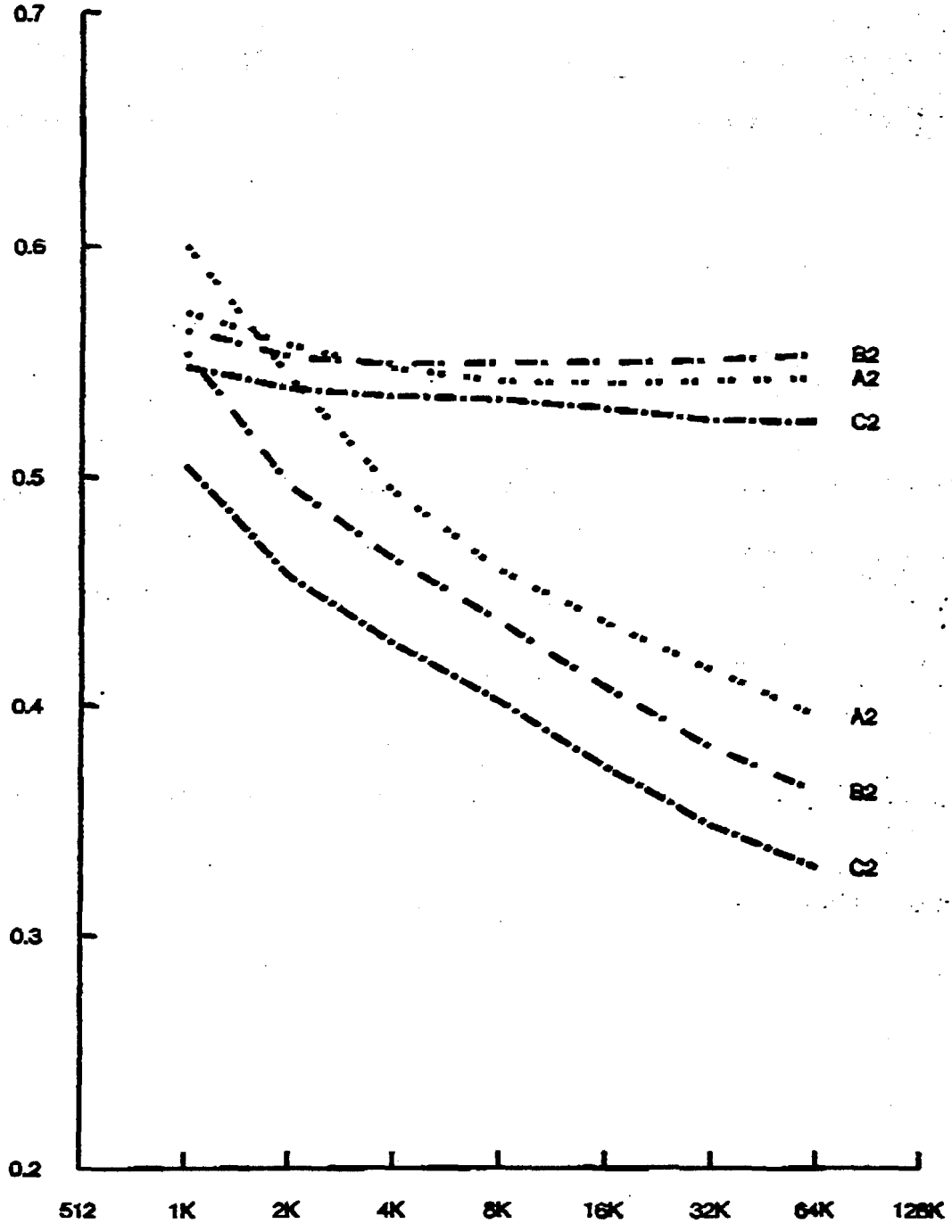

Chart 4. Compression vs. Window Size, Data Set NS (bottom) Data Set BF (top)

For all of our methods, **A2**, **B2**, and **C2**, window size is a significant consideration because it determines storage requirements and affects compression ratios. Chart 4 shows compression as a function of window size for the NS data set (concatenated into a single file to avoid start-up effects), and for the BF boot file. These two data sets were typical of the bimodal behavior we observed in our other data sets: large human-readable files benefit greatly from increasing window size, while other test groups show little improvement beyond a window size of 4K.

CONCLUSIONS

We have described several practical methods for lossless data compression and developed data structures to support them. These methods are strongly adaptive in the sense that they adapt not only during startup but also to context changes occurring later. They are suitable for most high speed applications because they make only one pass over source data, use only a constant amount of storage, and have constant amortized execution time per character.

Our empirical studies point to several broad generalizations. First, based on the **H0** and **H1** theoretical limits, textual substitution via **A2**, **B2**, or **C2** surpasses memoryless or first-order Markov methods applied on a character-by-character basis on half the data sets. On the other half, even the **CW** third-order method can't achieve the **H1** bound. This suggests that, to surpass textual substitution for general purpose compression, any Markov method must be at least second-order, and to date, all such methods have poor space and time performance.

Secondly, the methods we've developed adapt rapidly during startup and at transitions in the middle of files. One reason for rapid adaptation is the use of smaller representations for displacements to recent positions in the window. Another reason is the inclusion of multi-character literal codewords. Together the literals and short displacements allow our methods to perform well on short files, files with major internal shifts of vocabulary or statistical properties, and files with bursts of poorly compressing material—all properties of a significant number of files in our environment.

Thirdly, it appears that the displacement-and-length approach to textual substitution is especially effective on small files. On 11,000-byte program source files, for example, **A2** and **B2** were over 20% more compact than textual substitution methods which did not use a length field (**UW**, **MW1**, and **MW2**). This is not surprising because the particular advantage of the length field in copy codewords is rapid adaptation on small files. However, even on the largest files tested, **A2** and **B2** usually achieved significantly higher compression. Only on images did other methods compete with them; our most powerful method, **C2**, achieved higher compression than any other textual substitution method we tested on all data sets. The effect of a length field is to greatly expand dictionary size with little or no increase in storage or processing time; our results suggest that textual substitution methods that use a length field will work better than those which do not.

Fourthly, studies of **A2**, **B2**, and **C2** using different window sizes showed that, for human-readable input (e.g. English, source code), each doubling of window size improves the the compression ratio by roughly 6% (for details see Chart 4). Furthermore, the data structures supporting these methods scale well: running time is independent of window size, and memory usage grows linearly with window size. Thus increasing window size is an easy way to improve the compression ratio for large files of human-readable input. For other types of input the window size can be reduced to 4096 without significantly impacting compression.

Going beyond these empirical results, an important practical consideration is the trade-off among speed, storage, and degree of compression; speed and storage have to be considered for both

compression and expansion. Of our own methods, **A2** has very fast expansion with a minimal storage requirement. Its weakness is slow compression, which is seven times slower than expansion, even though the suffix tree data structure with amortized update uses constant amortized time per character. However, in applications which can afford relatively slow compression, **A2** is excellent; for example, **A2** would be good for mass distribution of software on floppy disks or for overnight compression of files on a file server. Furthermore, if the parallel matching in the compression side of **A2** were supported with VLSI, the result would be a fast, powerful method requiring minimal storage for both compressing and expanding.

B2 provides nearly three times faster compression than **A2** but has somewhat slower expansion and adaptation. Thus, **B2** is well suited for communication and archiving applications.

A1 and **B1** do not compress as well as **A2** and **B2**, respectively, but because of their two-codeword, byte-aligned encodings they are better choices for applications where simplicity or speed is critical. (For example, J. Gasbarro has designed and implemented an expansion method like **A1** to improve the bandwidth of a VLSI circuit tester [G 88].)

C2 achieves significantly higher compression than **B2**, but its expander is somewhat slower and has a larger storage requirement. In the compression study reported in Section 8, **C2** achieved the highest compression of all methods tested on 6 of the 10 data sets.

We believe that our implementations and empirical results demonstrate the value of window-based textual substitution. Together the **A**, **B** and **C** methods offer good options that can be chosen according to resource requirements.

ACKNOWLEDGEMENTS.

We would like to thank Jim Gasbarro, John Larson, Bill Marsh, Dick Sweet, Ian Witten, and the anonymous referees for helpful comments and assistance.

REFERENCES

[A 63] Norman Abramson
Information Theory and Coding
McGraw-Hill, 1963

[B 59] G. P. Basharin
On a Statistical Estimate for the Entropy of a Sequence of Independent Random Variables
Theory Probability Appl. 4:333-336, 1959.

[B 86] Timothy C. Bell
Better OPM/L Text Compression
IEEE Transactions on Communications, COM-34(12):1176–1182, 1986.

[BCW 88] Timothy C. Bell, John G. Cleary, and Ian H. Witten
Text Compression
In press with Prentice Hall, 1988.

[BSTW 85] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei
A Locally Adaptive Data Compression Scheme
Communications of the ACM 29(4):320–330, 1985.

[CW 84] John G. Cleary and Ian H. Witten
Data Compression Using Adaptive Coding and Partial String Matching
IEEE Transactions on Communications, COM-32(4):396–402, 1984.

[CH 86] G. V. Cormack and R. N. S. Horspool
Data Compression Using Dynamic Markov Modelling
The Computer Journal, 30(6): 541–550, 1987.

[DBR 66] G. B. Dantzig, W. O. Blattner, and M. R. Rao
Finding a Cycle in a Graph with Minimum Cost to Time Ratio
with Application to a Ship Routing Problem
P. Rosenstiehl, ed.
Theory of Graphs Gordon and Breach, 1966.

[E 75] P. Elias
Universal Codeword Sets and Representations of the Integers.
IEEE Transactions on Information Theory, IT-21(2):194–203, 1975.

[ER 78] S. Even and M. Rodeh
Economical Encoding of Commas Between Strings
Communications of the ACM, 21:315–317, 1978.

[G 78] R. G. Gallager
Variations on a Theme by Huffman
IEEE Transactions on Information Theory IT-24(6):668–674, 1978.

[G 88] J. Gasbarro
An Architecture for High-Performance VLSI Testers
Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 1988.

[G 66] Solomon W. Golomb
Run-Length Encodings
IEEE Transactions of Information Theory IT-12:399–401, 1966.

[G 80] M. Guazzo
A General Minimum Redundancy Source-coding Algorithm
IEEE Transactions on Information Theory IT-26(1):15–25, 1980.

[GH 82] Gu Guoan and John Hobby
Using String Matching to Compress Chinese Characters
Stanford Technical Report STAN-CS-82-914, 1982.

[HC 86] R. Nigel Horspool and Gordon V. Cormack
Dynamic Markov Modelling—A Prediction Technique
Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 700–707, 1986.

[H 51] D. A. Huffman
A Method for the Construction of Minimum-Redundancy Codes
Proceedings of the I.R.E 40:1098–1101, 1952.

[HR 80] Roy Hunter and A. Harry Robinson
International Digital Facsimile Coding Standards
Proceedings of the IEEE 68(7):854–867, 1980.

[J 85] Matti Jakobsson
Compression of Character Strings By an Adaptive Dictionary
BIT 25: 593–603, 1985.

[J 81] Christopher B. Jones
An Efficient Coding System for Long Source Sequences
IEEE Transactions on Information Theory IT-27(3):280–291, 1981.

[KBG 87] M. Kempf, R. Bayer and U. Güntzer
Time Optimal Left to Right Construction of Position Trees
Acta Informatica, 24:461–474, 1987.

[K 75] Donald E. Knuth
The Art of Computer Programming, Volume 3: Sorting and Searching
Addison-Wesley, second printing, 1975.

[K 85] Donald E. Knuth
Dynamic Huffman Coding
Journal of Algorithms 6:163–180, 1985.

[L 83] Glen G. Langdon, Jr.
A Note on the Ziv-Lempel Model for Compressing Individual Sequences
IEEE Transactions on Information Theory IT-29(2):284–287, 1983.

[LR 81] Glen G. Langdon, Jr. and Jorma Rissanen
Compression of Black-White Images with Arithmetic Coding
IEEE Transactions on Communications COM-29(6):858–867, 1981.

[LR 83] Glen G. Langdon, Jr. and Jorma Rissanen
A Double Adaptive File Compression Algorithm
IEEE Transcations on Communications COM-31(11):1253–1255, 1983.

[L 76] Eugene L. Lawler
Combinatorial Optimization: Networks and Matroids
Holt, Rinehart and Winston, 1976.

[M 76] Edward M. McCreight
A Space-Economical Suffix Tree Construction Algorithm
Journal of the Association for Computing Machinery 23(2):262–272, 1976.

[MW 84] Victor S. Miller and Mark N. Wegman
Variations on a theme by Ziv and Lempel
IBM Research Report RC 10630 (# 47798), 1984
Combinatorial Algorithms on Words, NATO ASI Series F, 12:131–140, 1985.

[M 68] Donald R. Morrison
PATRICIA Practical Algortihm To Retrieve Information Coded in Alphanumeric
Journal of the Association for Computing Machinery 15(4): 514-534, 1968.

[P 76] Richard Clark Pasco
Source Coding Algorithms for Fast Data Compression
Ph.D Dissertation, Dept. of Electrical Engineering, Stanford University, 1976.

[RL 79] Jorma Rissanen and Glen G. Langdon, Jr.
Arithmetic Coding
IBM Journal of Research and Development 23(2):149–162, 1979.

[RL 81] Jorma Rissanen and Glen G. Langdon, Jr.
Universal Modeling and Coding
IEEE Transactions on Information Theory IT-27(1):12–23, 1981.

[RPE 81] Michael Rodeh, Vaughan R. Pratt, and Shimon Even
Linear Algorithm for Data Compression via String Matching
Journal of the Association for Computing Machinery 28(1):16–24, 1981.

[S 48] C. E. Shannon
A Mathematical Theory of Communication
The Bell System Technical Journal 27(3):379–423 and 27(4): 623–656, 1948.

[SS 82] James A. Storer and Thomas G. Szymanski
Data Compression via Textual Substitution
Journal of the Association for Computing Machinery 29(4):928–951, 1982.

[V 85] Jeffrey Scott Vitter
Design and Analysis of Dynamic Huffman Coding
Brown University Technical Report No. CS-85-13, 1985.

[W 73] Peter Weiner
Linear Pattern Matching Algorithms
Fourteenth Annual Symposium on Switching and Automata Theory, 1–11, 1973.

[W 84] Terry A. Welch
A Technique for High Performance Data Compression
IEEE Computer 17(6): 8–19, 1984.

[WNC 87] Ian H. Witten, Radford M. Neal, and John G. Cleary
Arithmetic Coding for Data Compression
Communications of the ACM, 30(6):520–540, 1987.

[Z 78] Jacob Ziv
Coding Theorems for Individual Sequences
IEEE Transactions on Information Theory IT-24(4):405–412, 1978.

[ZL 77] Jacob Ziv and Abraham Lempel
A Universal Algorithm for Sequential Data Compression
IEEE Transactions on Information Theory IT-23(3):337–343, 1977.

[ZL 78] Jacob Ziv and Abraham Lempel
Compression of Individual Sequences via Variable-Rate Coding
IEEE Transactions on Information Theory IT-24(5):530–536, 1978.

APPENDIX A. A PATHOLOGICAL EXAMPLE

We now show a string that has the F pattern of equation (6) of Section 3:

$$\begin{array}{cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \\ p_{10} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_2 & p_{10} & p_{10} & p_9 & \cdots \end{array} \tag{6}$$

Hereafter we will stop abstracting the string by its copy lengths. Capital letters are strings, small letters are single characters, and i, j, r, p, b are integers. The pathological string follows the pattern:

$$M_0 M_1 \ldots M_{r-1} M_0 M_1 \ldots M_{r-1} M_0 M_1 \ldots, \tag{9}$$

where the parameter r is chosen large enough so that one iteration exceeds the finite window (this prevents direct copying from the beginning of one M_0 to a subsequent M_0). Within each M_i we have groups,

$$M_i = G_{i0} G_{i1} G_{i2} \ldots G_{i(n/p-1)}, \tag{10}$$

and each group is:

$$G_{ij} = s_{jp} B_0 s_{(j+i)p} c_i s_{jp+1} B_1 s_{(j+i)p+1} c_i s_{jp+2} B_2 s_{(j+i)p+2} c_i s_{jp+3} \ldots B_{p-1} s_{(j+i)p+p-1} c_i. \tag{11}$$

We have introduced two more parameters: p is the number of minor blocks B_i, and n is the number of s characters. All of the s subscripts in the above formula are computed mod n. The groups skew so that, for example, the beginning of $G_{10} = s_1 B_1 s_{p+1} \ldots$ will not match entirely with the beginning of $G_{00} = s_1 B_1 s_1 \ldots$. It will, however, match in two parts: the prefix $s_1 B_1$ appears in both strings, and the suffix $G_{10} = \ldots B_1 s_{p+1} \ldots$ will match with the suffix of $G_{01} = \ldots B_1 s_{p+1}$. If, for example, B_1 has 9 characters, this gives two consecutive locations where a copy of size 10 is possible, in the pattern of equation 6.

It remains to create the match of length 2 at position 12 in equation (6). For this purpose, each of the c_i above are either e_i or o_i. They will always precede respectively even and odd numbered s_j, and match in pairs with their following s_j's. For example, the e_0 in $G_{00} = s_1 B_1 s_1 e_0 s_2 B_2 s_2 \ldots$ will match with s_2. The $e_0 s_2$ match is hidden in a minor block, segregated from the odd numbered s_i:

$$\begin{aligned} B_0 &= x e_0 s_0 e_0 s_2 e_0 s_4 \ldots e_0 s_{b-2} \\ B_1 &= x e_0 s_b e_0 s_{b+2} e_0 s_{b+4} \ldots e_0 s_{2b-2} \\ &\vdots \\ B_{n/b} &= x e_1 s_0 e_1 s_2 e_1 s_4 \ldots e_1 s_{b-2} \\ &\vdots \\ B_{p/2} &= x o_0 s_1 o_0 s_3 o_0 s_5 \ldots o_0 s_{b-1} \\ &\vdots \end{aligned} \tag{12}$$

This causes p and n to be related by:

$$pb = 2nr$$

In the case of our running example, where the finite window is size 4096 and the maximum copy length is 16 characters, an appropriate setting of the above parameters is:

$$r = 2, \quad b = 8, \quad p = 100, \quad n = 200 \tag{13}$$

We need to take some care that the heuristic does not find the optimal solution. It turns out that if we just start as in equation 9, then the first M_0 will not compress well, but the heuristic will start the behavior we are seeking in M_1. Asymptotically we achieve a worst case ratio of 4/5 between the optimal algorithm and the policy heuristic.

Appendix B: Computing a Unary-Based Variable-Length Encoding of the Integers

In Section 5 we defined a (start, step, stop) unary code of the integers as a string of n ones followed by a zero followed by a field of j bits, where j is in the arithmetic progression defined by (start, step, stop). This can be defined precisely by the following encoder:

```
EncodeVar: PROC [out: CARDINAL, start, step, last: CARDINAL] ~ {
    UNTIL out < Power2[start] DO
        PutBits[1, 1];
        out ← out - Power2[start];
        start ← start + step;
        ENDLOOP;
    IF start < last THEN PutBits[out, start + 1]  -- 0 followed by field of size "start"
    ELSE IF start > last THEN ERROR
    ELSE PutBits[out, start];  -- save a bit
    };


PutBits: PROC [out: CARD, bits: INTEGER] ~
    Output the binary encoding of "out" in a field of size "bits."
```

Notice that the encoder is able to save one bit in the last field size of the arithmetic progression.