

A TEXT COMPRESSION SCHEME THAT ALLOWS FAST SEARCHING DIRECTLY IN THE COMPRESSED FILE

Udi Manber¹

Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

March 1993 (Revised September 1994 and May 1996)

To Appear in ACM TOIS

ABSTRACT. A new text compression scheme is presented in this paper. The main purpose of this scheme is to speed up string matching by searching the compressed file directly. The scheme requires no modification of the string-matching algorithm, which is used as a black box; any string-matching procedure can be used. Instead, the *pattern* is modified; only the outcome of the matching of the modified pattern against the compressed file is decompressed. Since the compressed file is smaller than the original file, the search is faster both in terms of I/O time and processing time than a search in the original file. For typical text files, we achieve about 30% reduction of space and slightly less of search time. A 30% space saving is not competitive with good text compression schemes, and thus should not be used where space is the predominant concern. The intended applications of this scheme are files that are searched often, such as catalogs, bibliographic files, and address books. Such files are typically not compressed, but with this scheme they can remain compressed indefinitely, saving space while allowing faster search at the same time. A particular application to an information retrieval system that we developed is also discussed.

1. Introduction

Text compression is typically used to save storage or communication costs. We suggest an additional purpose in this paper. By reducing the size of a text file in a special way, we reduce the time it takes to search through it. Our scheme improves the speed of string matching and we also save space in the process. The savings are not spectacular, but they are in a sense for free. Files that are usually not compressed because they are often read, can now be compressed and at the same time the speed of searching is improved. The improvement is independent of current technology, because it comes from the fact that the compressed files are smaller than the original and therefore less work is done. The same improvement will hold for faster CPU or I/O. Another important advantage of our scheme is that it is independent of the actual string-matching program. There is no need to modify the search program in any way; we only add a

¹ Supported in part by NSF grants CCR-9002351 and CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. Part of this work was done while the author was visiting the University of Washington.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

preprocessing step and a postprocessing step. As will be discussed in section 3.1, in our implementation the name of the search program is part of the input to the search. This makes it much easier for programs that handle text to adapt to handle our compressed text.

Searching in compressed files was studied, in a theoretical framework, by several authors. Eilam-Tsoreff and Vishkin [EV88] looked at run-length compression, Amir, Landau and Vishkin [ALV92] and Amir and Benson [AB92a, AB92b] looked at two-dimensional matching, and very recently Amir, Benson and Farach [ABF94] looked at LZW compression. While these authors have looked at some theoretical questions, the practical side has not been sufficiently addressed yet. It is not clear, for example, whether in practice the compressed search in [ABF94] will indeed be faster than a regular decompression followed by a fast search. As far as we know [Fa93], no implementation of these algorithms exist. In this paper we concentrate on practical issues in this area. In particular, we compare our algorithms to Boyer-Moore pattern matching, which is typically 5-7 times faster in practice (for natural language texts) than the Knuth-Morris-Pratt approach (which is taken in [ABF94]). Speeding up the fast Boyer-Moore searching is a real challenge (e.g., see [CGJLPR] and [HS91]). For this reason we also rejected Huffman's and other bit-level compression, because we want to keep the search on a byte level for efficiency.

Several schemes have been suggested for compression of full-text information-retrieval systems ([KBD89], [WBN91], [MZ92], [GM94]). Moffat and Zobel [MZ92] (see also [WMB94]), is the most promising of the three. They first build a regular inverted index and then use a Huffman-encoding type compression (word based) for the text and a suffix compression for the inverted index. A query first accesses the index, then led to the right place in the compressed text, which is then decompressed. Such a scheme achieves better compression than our scheme for large collection of files. Our scheme is much simpler and more convenient to use for small to medium scale files and especially for files that change frequently. It can also handle queries that match part of words. It is not suitable, however, for very large information bases. We will assume throughout the paper that the search is sequential. Sequential search occurs in many other applications, and in particular, it plays a key role in an information-retrieval system that we designed [MW94]. We discuss this in section 4. We also assume that we are dealing with English text and the ASCII character set.

We first present the idea behind the compression scheme, and then describe in detail the algorithms involved and their implementation. Although an NP-complete problem needs to be solved as part of the compression algorithm, we present a solution to it that turns out to be fast and effective in practice. We then discuss how the search is done and present experimental results for different search routines. We conclude with a discussion of possible applications.

2. The Compression Algorithm

The basis of the compression is a very simple pattern-substitution method that has been used by Snyderman and Hunt [SH70], Jewell [Je76], and most likely reinvented by many others. The goal is to substitute common pairs of characters with special symbols that are still encoded in one byte. A byte allows 256 possible encodings, but typical text uses much fewer. ASCII encoding, for example, uses only 128 characters, leaving us 128 additional encodings to be used for pairs. The compression consists of substituting each of the common pairs with the special byte allocated for it, and decompression is achieved by reversing this procedure. Both are very fast. This kind of compression is not as good as more modern

techniques such as the Lempel-Ziv based algorithms [ZL77, We84] or context-modeling algorithms [BCW90]. However, context-dependent techniques cannot perform string matching without keeping track of the compression mechanism at the same time, because the compression depends on previous text. Our goal is to perform the string matching directly on the compressed text so that it takes *less time* than a matching to the original text. If we performed additional processing to do decompression on the fly, it would take more time, since decompression is typically an order of magnitude slower than string matching. (We may save IO time by reading less with a better compression scheme, but the extra processing time will exceed the savings.) Speeding up string matching is a difficult goal because string matching is already extremely efficient; with Boyer-Moore filtering [BM77], the matching procedure skips many characters and only a fraction of the text is looked at. As we will show, a pattern-substitution method can be modified to allow a direct search in the compressed file. Essentially, the search for a given pattern will work by modifying the pattern, but not the search routine, and matching the modified pattern to the compressed file. There is, however, one major problem with this idea.

Suppose that the pattern we are looking for is the word *wood*. Suppose further that both *wo* and *od* belong to the list of common pairs. We'll use \boxed{ij} to denote the encoding of the pair *ij*, and \square to denote a blank. The pattern *wood* is thus translated into $\boxed{wo} \boxed{od}$, but that is not necessarily how the word *wood* appears in the compressed file. For example, the pairs $\square w$, *oo*, and $d\square$ could be common pairs as well, in which case it is possible that the word *wood* was compressed into $\boxed{\square w} \boxed{oo} \boxed{d\square}$, which are 3 totally different characters. And these are not the only two possibilities. The word *wood* could start a new line, it could be a part of a larger word (say, *deadwood* in which case the encoding could be $\boxed{dw} \boxed{oo} \boxed{d\square}$), or it could be followed by a period — $\boxed{\square w} \boxed{oo} \boxed{d.}$ Each of these combinations leads to a different encoding of the word *wood* in the compressed text.

The reason for the ambiguities in encoding *wood* is that, even if we know all the common pairs, we cannot determine ahead of time whether the first character *w* will belong to the end of a pair, the beginning of a pair, or to no pair; it depends on the text. The main idea of this algorithm is to devise a way to restrict the number of possible encodings for any string, and still achieve good compression. One possible approach would be to start the compression always from the beginning of a word. In other words, compress the words separately, and leave all other symbols intact. This will partially solve our problem, but it has two major weaknesses. First, it restricts the search to whole words. If the pattern is *compression* and the text has *decompression*, it may not be found. Second, blanks, commas, and periods are very common characters. Removing them from consideration for compression limits the compression rate significantly. We present a different approach.

We choose the common pairs in a special way, not just according to their frequency. The main constraint that we add is that no two chosen pairs can overlap. An equivalent way to pose this constraint is that no character can be a first character in a pair and a second character in a (possibly another) pair. The compression will be done in a “greedy” fashion, compressing a pair whenever it is possible to do so without looking ahead. Since we can determine, for each character of the pattern, whether it can be the start of a pair, we limit the ambiguities. As we will show, this leads to a speedy matching. But before we see the matching part, let's see how to enforce this constraint in the best possible way. To do that we abstract the problem.

2.1. Finding the Best Non-Overlapping Pairs

Let's assume that we first scan the text and find, for each possible pair of characters, the number of times it appears in the text. We denote by $f(ab)$ the frequency of the pair ab . We now construct a directed graph $G = (V, E)$, such that the vertices of G correspond to the different characters in the text, one vertex for each unique character, and the edges correspond to character pairs. The weight of each edge ab is $f(ab)$. We want to find the "best" edges (corresponding to pairs) such that no edges "overlap." More precisely, we want to partition the vertices into two sets V_1 and V_2 , such that the sum of the weights of the edges that go from V_1 to V_2 (these are the edges we choose) is maximized. We also need to restrict the number of edges we select (there is a limit on the number of unused byte combinations, which we assume is 127), but let's leave this aside for now. The sum of weights of the edges from V_1 to V_2 gives us the exact compression savings. Each of these edges corresponds to a pair that will be encoded with one byte, thus it contributes one byte of savings.

The formulation above is a very clean abstraction of our problem, but unfortunately, it is also a very difficult problem. It turns out that the problem is NP-complete even for unit weights and undirected graphs (by a reduction from the Bipartite Subgraph problem [GJ79, p. 196]). But, on the other hand, the graphs we are dealing with are not too large. The important edges correspond to common pairs and not too many characters appear in common pairs. In fact, the number of unique characters overall is quite limited. So, even though the problem for large graphs is intractable, it may be possible to solve it in practice for small graphs. We experimented with several approaches to solving this problem. The following simple heuristic seems to be the most effective. It is fast, and in all the cases we verified, it gave the optimal solution. More on the verification later. It is worth noting that standard methods of algorithm analysis would not be applicable here, because the size of the problem (e.g., the number of unique characters) is small and fixed. We believe that the only practical way to evaluate different approaches for such small-scale problems is to try them on typical inputs, which is what we did.

The main step of the algorithm is a local optimization heuristic that can be called a 1-OPT procedure. Given an initial partition of the vertices into V_1 and V_2 , we examine each vertex to see whether switching this vertex to the other set would improve the total. We continue with such switches until no more are possible (since each switch improves the total, this procedure converges). We also tried a 2-OPT heuristic, in which we attempt to switch 2 vertices at the same time. This is different from switching each one separately, because the edges connecting these two vertices count differently. The main question is how to select the initial partition. We tried several greedy methods, the most obvious one is to first pick the edges with maximum "benefit" of switching, but they did not work very well. We found that the simplest method is the most effective: use a random partition. We repeat the algorithm several times with different random partitions. The solutions were almost uniformly good, even without using 2-OPT. The best solution was typically found in 5-20 random trials, which led us to set the number of trials to 100 to be on the safe side. In fact, the *average* total compression over 100 single random trials was 98.6% of the best total of these 100 trials for one text, 96.1% of the best for the second text, and 96.9% for the third text. The standard deviation ranged from 2.3% to 4.7%. The *worst* total we found in these 300 random trials was 80% of the best. Running 1-OPT 100 times can be done reasonably fast, as we will show shortly.

The last problem is to limit the number of selected edges to the available unused byte combinations. We set this number to be 127, leaving 128 characters for the regular ASCII set plus one special delimiter

(with value 128). We need this delimiter for the (hopefully) rare cases of characters in the text with values of > 128 (which may occur with non-ASCII text). In these cases, the compressed file contains the delimiter followed by the non-ASCII character. We could have achieved a slightly better compression rate by figuring out the exact set of characters used by the text, and using all other unused byte combinations, but for simplicity, we decided not to do that. Also, for easier integration with the string matching program, we did not use the newline symbol in any of the pairs, leaving it always uncompressed.

We could not see any intelligent way to incorporate the limit on the number of pairs during the pairs selection. Instead, we simply find the best partition, ignoring the limit, and then take the best (highest weight) 127 edges from V_1 to V_2 . We also tried to limit the number of edges initially; for example, start with, say, the 400 highest-weight edges of the graph and find the best partition for them, and then take the 127 best of those. Or, alternatively, start with, say 30 vertices of highest degree. The differences in the total compression were usually small (2-3%), although the running time was, of course, improved because the graphs were smaller. However, there were added complications because if the graph is reduced too much, the total number of edges becomes less than the optimal 127. So we decided to stay with the original scheme. The algorithm, using only 1-OPT switches, is given in pseudo-code in Figure 1.

We leave it to the reader to fill in the details of the implementation. The graphs we handle are small (hundreds of edges at the most), and for English texts they will always be small (the size of the graph depends on the number of unique characters in the language, not the size of the text). Therefore, as long as x (the number of randomizations) is relatively small, the algorithm is quite fast. In particular, the running time of this algorithm is a small fraction of the running time of the entire compression algorithm (unless the

Algorithm Best_Non_Overlapping_Pairs(G : weighted graph)

```

repeat x times { x is a constant; we used 100 for ASCII texts }
  randomly assign each vertex to either  $V_1$  or  $V_2$  with equal probability;
  for each vertex  $v \in V$  do
    put  $v$  on the queue;
  loop until the queue is empty
    pop  $v$  from the queue;
    if switching  $v$  to the opposite set improves the sum of weights then
      switch  $v$ ;
      if switching  $v$  caused other vertices, not already on the
        queue, to prefer to switch then put them on the queue;
  store the best solution to date;
output the highest weight 127 edges from the best partition;

```

Figure 1: The algorithm to select the partition.

files are very small). Sample of compression savings, running times, and the percentage of the running times used to find the best pairs are given in Figure 2 for 5 different texts: Roget's Thesaurus, the 1992 CIA world facts, and King James' Bible from Project Gutenberg; a collection of articles from the Wall Street Journals, and the archives of the Telecom Digest, a discussion group about telecommunication. The running times are in seconds on a DEC 5000/240.

To verify that the solutions obtained from our algorithm are not too far from optimal, we also implemented a deterministic algorithm that guarantees the best solution. This algorithm is exponential, but by using binary reflected Gray codes [BER76] it was possible to run experiments for up to 30 vertices, which is not too far from real data. In all our tests the solution found by the random algorithm was indeed the optimal. We also compared the best 127 edges obtained from the random algorithm on the whole graph (usually about 40-60 vertices) vs. the best 127 edges from the optimal solution for a "good" induced subgraph of 30 vertices, and those solutions were within 1-2% apart.

Another interesting experiment was done with random text. The compression rates of this algorithm for random text of uniform distribution are quite predictable. Suppose that the text consists of S characters each with equal probability, and that it is large enough. A partition of the vertices into equal-size sets would yield $(S/2)^2$ edges from the first set to the second set. (If the text is large all possible pairs will most likely appear). All edges occur with the same probability, so, by the law of large numbers, one would expect that the weight of the selected edges will be only slightly above the weight of any other set of $(S/2)^2$ edges, which is one fourth of the total number of edges. The best partition will be better than the average, but again, if the text is large enough, the variance will be small. Therefore, if about one fourth of the edges can be chosen, we expect the savings to be close to 25%.² Since we can use only 127 edges, the saving would be reduced by a factor of $127/(S/2)^2$. For example, for $S = 30$ we get approximately 14%

Text	size original	compression savings	compression time	uncompression time	% of time for pairs
Roget Thesaurus	1.38	31%	10.3	0.9	14.6%
CIA world facts	2.42	28%	11.4	1.7	13.2%
King James' Bible	4.85	30%	13.3	3.5	9.0%
Wall Street J.	15.84	33%	23.9	11.2	6.7%
Telecom archives	69.21	28%	104.0	52.1	1.9%

Figure 2: Statistics for the compression and decompression algorithms.

Files sizes are given in Mbytes; running times are given in seconds.

² At some point we used the UNIX old *rand* procedure to generate random text and obtained a perfect compression, for our algorithm, of 50%. The reason was that the old *rand* procedure generates "random" numbers that go from odd to even to odd with no exception. Our heuristic was good enough (it uses much better random numbers) to catch the best partition, which was, of course, all the odd characters in one set and the even in the other. All edges were then used. This is another example of the risk of using bad random number generators, or as in our case, copying from old code.

savings for random text (which we verified by experiments) compared to about 30% for natural text. Just another indication that text is not random.

2.2. Some Additional Implementation Issues

A couple of other points are worth mentioning. The compression algorithm is a two-stage algorithm; the first stage reads the file and computes the best pairs, and the second stage reads the file again and performs the compression. To improve this process, we use only a sample of the file in the first stage, usually the first 1 million characters. We keep that part in memory for the next stage, so it is not too much of an overhead. It is possible to use "standard" pairs that are computed on a large sample of text. This will save time for the compression, but for most cases, it is probably not cost effective, as it leads to a less effective compression. For example, we compressed the version of the Bible we had using the best pairs computed from the Wall Street Journal text and obtained a compression that required 3.9% more space. We also did it the other way around and found the Wall Street Journal to be much more sensitive: it required 6.0% more space. We support an option that reads the pairs from another compressed file rather than computing them for the given file, because it allows one to concatenate to a compressed file. If file1 is a compressed file and we want to add to it file2, we need only to compress file2 using file1's pairs and concatenate; there is no need to decompress, to concatenate, and to compress again. We can do that in the middle of the file as well. It is therefore possible to adapt a text editor to read and edit a compressed file directly.

Our compressed files save about 30% space. If searching is not needed, there are much better compression algorithms, such as UNIX *compress*, based on the Lempel-Ziv-Welch algorithm [We84], which can achieve 50-60% reduction. We tested whether compressing our compressed files further with UNIX *compress* lead to the same compression rates as compressing the original file, and found that we got *better* compression rates. Typically using both programs together (provided that our scheme is used first) improves the compression by an extra 6%. For example, the Wall Street Journal text we had was compressed to 6.96MB with *compress*, and to 6.55MB using our scheme in addition to *compress*. This is so because *compress* adds strings to its dictionary one character at a time, and since our scheme shortens strings, they are added to the dictionary earlier.

3. Searching Compressed Files

3.1. The Search Algorithm

The search program consists of five parts. In the first part the common pairs used in the compression are read (they are stored at the beginning of the compressed file) and two translation tables are constructed. One table maps all new characters (identified by having a value of ≥ 129) to their original pairs of characters. The second table is the inverse; it maps pairs of original characters into new characters. In the second part, the pattern is translated (compressed) using the inverse table. This translation is unique as we showed in the previous section, except possibly for the first and last character of the pattern. If either the first character belongs to V_2 in the partition or the last character belongs to V_1 , then they are removed from the pattern for now. The third part is to use the string-matching program — as a black box — to search for the compressed pattern in the compressed file and output all matches. This is done by creating a new process

and using UNIX's pipe facilities³ to divert the output to the fourth part, in which the output of the match is decompressed using the translation table. If either the first or last character of the pattern were removed in the second part, then the output must be further filtered. We do that by creating another process and running the same string-matching program on the (decompressed) output of the fourth part. This can be done more efficiently during the search because we know the exact location of the match, but since in most cases the output size is small, it makes little difference. (Adding it to the search would require an access to the source code of the search procedure, which we do not assume.) An example of a search is given in Figure 3.

The five parts may seem like a lot of work, but in fact only the third part is substantial. The first part depends on the common pairs and there are at most 127 of them. The second part depends on the pattern, which is typically very small. The fourth and fifth parts depend on the size of the output, which again is typically very small. The third part therefore dominates the running time of the program.

One of the main features of our scheme is that it can work with almost any string-matching program (assuming a simple string as a pattern; we do not support regular expressions). The only requirement is that the program can handle any character encoded in a byte (i.e., that it is 8-bit clean). We encountered no problem using any (UNIX) string-matching program. We believe that this is a very important design

command: `cgrep pattern input_file`

First part: read the list of common words;
for example, `pa te` and `n` may be common pairs.

Second part: translate *pattern* into `pa t te r`;
(Notice that we removed the last character n, because it is a first character in a common pair.)

Third part: search for `pa t te r` in `input_file` using your favorite string-matching program.

Fourth part: decompress what you find above; for example,
`pa t te r n ap pe a rs h e re` — pattern appears here

Fifth part: run the string matching again on the output to filter a possible match to only *patter* (recall that we removed the last n).

Figure 3: An example of the search.

³ It is possible, of course to use the string-matching procedure directly as a procedure in the search rather than to create a new process. We chose to create another process to make the whole program more general by allowing any string-matching program.

decision. The modularity allows using our compression without having to modify existing software. This makes the whole scheme much more reliable, general, and convenient. In addition, we mark a compressed file with a distinct signature at its beginning. If our search program does not see this signature it reverts to the original search procedure. So, one can use our program as a default on all files and get the speedup whenever the file is compressed. (A move to a two-byte representation of text will most probably create even more unused byte combinations so our scheme will still work effectively; finding the best non-overlapping pairs may be more of a problem.)

3.2. Experiments

We used texts of varying sizes (these texts were also used in Figure 1), and several search programs. We present the results for three large texts and two representative search programs. The first program is `fgrep`, which does not use Boyer-Moore filtering, and the second is our own `agrep` [WM92], which does use it and is much faster as a result. We selected 100 random words from the dictionary (the effectiveness of a Boyer-Moore search depends on the pattern), and ran 100 searches for each combination of search and text. We give the average running times in Figure 4. All experiments were run on a DECstation 5000/240 running Ultrix and times were user plus system times obtained from the UNIX time routines (which are not very precise), and given in seconds.

4. Applications

Our scheme can be used for many applications that require searching. We highlight two possible applications of this scheme. In [MW94] we present a design of a multi-level information retrieval system called *Glimpse* (available from <http://glimpse.cs.arizona.edu/>). We cannot describe the system in detail here, but the main idea behind it is to partition the information space (e.g., a personal file system) into blocks, and build an index that is similar to an inverted index but much much smaller (typically 2-4% of the total size, which is an order of magnitude smaller than the best full-text indexes). The index contains all distinct words, but for each word there are pointers to only the blocks in which it appears (rather than to its exact location as in an inverted index). Searching for a word consists of finding all its blocks and then searching those blocks sequentially. Sequential search is fast enough for medium-size files, therefore this search strategy, if implemented carefully, is reasonably fast. Its main advantage is the tiny space occupied by the index. Having a small index leads to fast indexing and incremental indexing. Using the compression described here, one can keep many files compressed and still be able to perform the sequential search.

Text	compression savings	original file		compressed file		improvements	
		fgrep	agrep	fgrep	agrep	fgrep	agrep
Bible	30%	7.5	1.4	5.4	1.0	28%	29%
WSJ	33%	24.9	5.0	17.4	3.8	30%	24%
Telecom	28%	109.2	21.2	80.0	16.4	27%	23%

Figure 4: Running times for `agrep` and `fgrep` for various texts.

That will save time and space. In particular, the index itself can be compressed.

Another application is in filtering information. For example, the information may consist of some newsfeed and all articles containing certain patterns are supposed to be extracted. With our scheme the newsfeed can come in compressed, resulting in faster filtering. (Although we discussed searching single patterns here, our scheme will work for searching multiple patterns with either the Aho-Corasick algorithm used in `fgrep` [AC75] or with `agrep`'s algorithm [WM94].)

5. Future Work

A natural improvement attempt to our compression scheme is to consider more than just two characters at a time. This is typical of fixed dictionary schemes (e.g., [BCW90], [RZ92]). The problem is to find some restrictions on the common strings that will prevent overlaps and will be easy to maintain.

The same approach may also be used for approximate matching, although the problem is more difficult. An error in the original text may be transformed into two errors in the compressed text (e.g., by changing a common pair to a pair that is not common), and two errors in the original text may be transformed into one error in the compressed text (e.g., by substituting one common pair for another). Therefore, we cannot simply match the compressed pattern to the compressed file. However, there are efficient techniques for reducing an approximate matching problem to a different exact matching problem. A simple example [WM92] is to divide the pattern into $k + 1$ parts such that any k errors will leave at least one part intact. An exact search is performed for all parts, and the output, which is hopefully much smaller, is then filtered appropriately. The scheme presented in this paper can be used for the exact matching part.

Recently, as part of the Glimpse package [MW94], we implemented a different scheme that achieves better compression (45-50%) and faster search, but requires a much larger dictionary [GM94]. It essentially translates many words into short representations and uses a fixed dictionary for the translation. A search first translates the pattern (assuming it is a complete word) into its encoding and then searches for the encoding directly. Patterns that are not in the dictionary are stored verbatim and the search for those turns out to be even faster (in the best case we achieved 4 fold speedup for unsuccessful search). The main problem with this approach is portability: unless everyone uses the same dictionary, the dictionary has to accompany the text when it is sent to another location.

Acknowledgements

Thanks to Dan Hirschberg, Richard Ladner, Martin Tompa, and Sun Wu for helpful discussions, and to Jan Sanislo for help in improving the code. We also thank an anonymous referee for numerous suggestions that significantly improved the manuscript.

References

- [AB92a] Amir, A, and G. Benson, "Two-dimensional periodicity and its application," *Proc. of the 3rd Symp. on Discrete Algorithms*, Orlando Florida (January 1992), pp. 440–452.

- [AB92b] Amir, A, and G. Benson, “Efficient two dimensional compressed matching,” *Proc. of the Data Compression Conference*, Snowbird Utah (March 1992), pp. 279–288.
- [ABF94] Amir, A, G. Benson, and M. Farach, “Let sleeping files lie: pattern matching in Z-compressed files,” *Proc. of the 5rd Symp. on Discrete Algorithms*, (January 1994), to appear.
- [AC75] Aho, A. V., and M. J. Corasick, “Efficient string matching: an aid to bibliographic search”, *Communications of the ACM*, **18** (June 1975), pp. 333–340.
- [BER76] Bitner J. R., G. Erlich, and E. M. Reingold, “Efficient generation of the binary reflected Gray code and its applications,” *Communications of the ACM*, **19** (September 1976), pp. 517–521.
- [BCW90] Bell, T. G., J. G. Cleary, and I. H. Witten, *Text Compression*, Prentice-Hall, Englewood Cliffs, NJ (1990).
- [BM77] Boyer R. S., and J. S. Moore, “A fast string searching algorithm,” *Communications of the ACM*, **20** (October 1977), pp. 762–772.
- [CGJLPR] Crochemore, M., A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, “Speeding up two string-matching algorithms,” *Algorithmica*, **12** (4/5) (October 1994), pp. 247–267.
- [EV88] Eilam-Tsoreff T., and U. Vishkin, “Matching patterns in a string subject to multilinear transformations,” *Proc. of the Int. Workshop on Sequences, Combinatorics, Compression, Security, and Transmission*, Salerno, Italy (June 1988).
- [Fa93] Farach M., private communication (October 1993).
- [GJ79] Garey M. R., and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, CA, 1979.
- [GM94] B. Gopal, and U. Manber, “A Fixed-Dictionary Approach to Fast Searching in Compressed Files,” manuscript.
- [HS91] Hume A., and D. Sunday, “Fast string searching,” *Software — Practice and Experience* **21** (November 1991), pp. 1221–1248.
- [Je76] Jewell G. C., “Text compaction for information retrieval systems,” *IEEE SMC Newsletter*, **5** (February 1976).
- [KBD89] Klein, S.T., A. Bookstein, and S. Deerwester, “Storing text retrieval systems on CD-ROM: compression and encryption considerations,” *ACM Trans. on Information Systems*, **7** (July 1989), pp. 230–245.
- [MW94] Manber U. and S. Wu, “GLIMPSE: A Tool to Search Through Entire File Systems,” *Usenix Winter 1994 Technical Conference*, San Francisco (January 1994), pp. 23–32.
- [MZ92] Moffat A., and J. Zobel, “Coding for compression in full-text retrieval systems,” *Proc. IEEE Data Compression Conference* (1992), pp. 72–81.
- [RZ92] Revuz D. and M. Zipstein, “DZ, A text compression algorithm for natural languages,” *Third Annual Symp. on Combinatorial Pattern Matching*, Tucson, AZ (1992), pp. 193–204.
- [SH70] Snyderman, M. and B. Hunt, “The myriad virtues of text compression,” *Datamation*, **16** (1970), pp. 36–40.

- [WBN91] Witten, I. H., T. C. Bell, and C. G. Nevill, “Models for compression in full-text retrieval systems,” *Proc. of the Data Compression Conference*, Snowbird, Utah (April 1991), pp. 23–32.
- [We84] Welch, T. A., “A technique for high-performance data compression,” *IEEE Computer*, **17** (June 1984), pp. 8–19.
- [WM92] Wu S., and U. Manber, “Fast Text Searching Allowing Errors,” *Communications of the ACM* **35** (October 1992), pp. 83–91.
- [WM94] Wu S., and U. Manber, “A Fast Algorithm for Multi-Pattern Searching” to appear in *Software, Practice & Experience*.
- [WMB94] Witten, I. H., A. Moffat, and T. C. Bell, *Managing Gigabytes*, Van Nostrand Reinhold, New York, 1994.
- [ZL77] Ziv, J. and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. on Information Theory*, **IT-23** (May 1977). pp. 337–343.

About the author

Udi Manber is a professor of computer science at the University of Arizona, where he has been since 1987. His research interests include computer networks and the World Wide Web, Software tools, especially search and resource discovery tools, design of algorithms, and pattern matching. He is the author of “Introduction to Algorithms - A Creative Approach,” and over 50 technical articles. Among the tools he developed or co-developed are Agrep, Glimpse, Harvest, GlimpseHTTP, Siff, Warmlist, and NetShell. He received three best-paper awards, two patents, and one PYI.