# A FAST ALGORITHM FOR MULTI-PATTERN SEARCHING

Sun Wu

Department of Computer Science
Chung-Cheng University
Chia-Yi, Taiwan
sw@cs.ccu.edu.tw

Udi Manber[1]

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

May 1994

***SUMMARY***

A new algorithm to search for multiple patterns at the same time is presented. The algorithm is faster than previous algorithms and can support a very large number — tens of thousands — of patterns. Several applications of the multi-pattern matching problem are discussed. We argue that, in addition to previous applications that required such search, multi-pattern matching can be used in lieu of indexed or sorted data in some applications involving small to medium size datasets. Its advantage, of course, is that no additional search structure is needed.

**Keywords**: algorithms, merging, multiple patterns, searching, string matching.

[1] Supported in part by a National Science Foundation grants CCR-9002351 and CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052.

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.

# 1. Introduction

We solve the following multi-pattern matching problem in this paper: Let $P = \{p_1, p_2, ..., p_k\}$ be a set of patterns, which are strings of characters from a fixed alphabet $\Sigma$. Let $T = t_1, t_2, ..., t_N$ be a large text, again consisting of characters from $\Sigma$. The problem is to find all occurrences of all the patterns of $P$ in $T$. For example, the UNIX *fgrep* and *egrep* programs support multi-pattern matching through the -f option.

The multi-pattern matching problem has many applications. It is used in data filtering (also called data mining) to find selected patterns, for example, from a stream of newsfeed; it is used in security applications to detect certain suspicious keywords; it is used in searching for patterns that can have several forms such as dates; it is used in *glimpse* [MW94] to support Boolean queries by searching for all terms at the same time and then intersecting the results; and it is used in DNA searching by translating an approximate search to a search for a large number of exact patterns [AG+90]. There are, of course, many other applications.

Aho and Corasick [AC75] presented a linear-time algorithm for this problem, based on an automata approach. This algorithm serves as the basis for the UNIX tool *fgrep*. A linear-time algorithm is optimal in the worst case, but as the regular string-searching algorithm by Boyer and Moore [BM77] demonstrated, it is possible to actually skip a large portion of the text while searching, leading to faster than linear algorithms in the average case. Commentz-Walter [CW79] presented an algorithm for the multi-pattern matching problem that combines the Boyer-Moore technique with the Aho-Corasick algorithm. The Commentz-Walter algorithm is substantially faster than the Aho-Corasick algorithm in practice. Hume [Hu91] designed a tool called *gre* based on this algorithm, and version 2.0 of *fgrep* by the GNU project [Ha93] is using it. Baeza-Yates [Ba89] also gave an algorithm that combines the Boyer-Moore-Horspool algorithm [Ho80] (which is a slight variation of the classical Boyer-Moore algorithm) with the Aho-Corasick algorithm.

We present a different approach that also uses the ideas of Boyer and Moore. Our algorithm is quite simple, and the main engine of it is given later in the paper. An earlier version of this algorithm was part of the second version of *agrep* [WM92a, WM92b], although the algorithm has not been discussed in [WM92b] and only briefly in [WM92a]. The current version is used in *glimpse* [MW94]. The design of the algorithm concentrates on typical searches rather than on worst-case behavior. This allows us to make some engineering decisions that we believe are crucial to making the algorithm significantly faster than other algorithms in practice.

We start by describing the algorithm in detail. Section 3 contains a rough analysis of the expected running time, and experimental results comparing our algorithm to three others. The last section discusses applications of multi-pattern matching.

# 2. The Algorithm

## 2.1. Outline of the Algorithm

The basic idea of the Boyer-Moore string-matching algorithm [BM77] is as follows. Suppose that the pattern is of length $m$. We start by comparing the *last* character of the pattern against $t_m$, the $m$'th character of the text. If there is a mismatch (and in most texts the likelihood of a mismatch is much greater than the likelihood of a match), then we determine the rightmost occurrence of $t_m$ in the pattern and shift accordingly. For example, if $t_m$ does not appear in the pattern at all, then we can safely shift by $m$ characters and look next at $t_{2m}$; if $t_m$ matches only the 4th character of the pattern, then we can shift by $m-4$, and so on. In natural language texts, shifts of size $m$ or close to it will occur most of the time, leading to a very fast algorithm. We want to use the same idea for the multi-pattern matching problem. However, if there are many patterns, and we would like to support tens of thousands of patterns, chances are that most characters in the text match the last character of some pattern, so there would be few if any such shifts. We will show how to overcome this problem and keep the essence (and speed) of the Boyer-Moore algorithm.

The first stage is a preprocessing of the set of patterns. Applications that use a fixed set of patterns for many searches may benefit from saving the preprocessing results in a file (or even in memory). This step is quite efficient, however, and for most cases it can be done on the fly. Three tables are built in the preprocessing stage, a SHIFT table, a HASH table, and a PREFIX table. The SHIFT table is similar, but not exactly the same, to the regular shift table in a Boyer-Moore type algorithm. It is used to determine how many characters in the text can be shifted (skipped) when the text is scanned. The HASH and PREFIX tables are used when the shift value is 0. They are used to determine which pattern is a candidate for the match and to verify the match. Exact details are given next.

## 2.2. The Preprocessing Stage

The first thing we do is compute the minimum length of a pattern, call it $m$, and consider only the first $m$ characters of each pattern. In other words, we impose a requirement that all patterns have the same length. It turns out that this requirement is crucial to the efficiency of the algorithm. Notice that if one of the patterns is very short, say of length 2, then we can never shift by more than 2, so having short patterns inherently makes this approach less efficient.

Instead of looking at characters from the text one by one, we consider them in blocks of size $B$. Let $M$ be the total size of all patterns, $M = k*m$, and let $c$ be the size of the alphabet. As we show in Section 3.1, a good value of $B$ is in the order of $\log_c 2M$; in practice, we use either $B=2$ or $B=3$. The SHIFT table plays the same role as in the regular Boyer-Moore algorithm, except that it determines the shift based on the last $B$ characters rather than just one character. For example, if the string of $B$ characters in the text do not appear in any of the patterns, then we can shift by $m-B+1$. Let's assume for now that the SHIFT table contains an entry for each possible string of size $B$, so its size is $|\Sigma|^B$. (We will actually use a compressed table with several strings mapped into the same entry to save space.) Each string of size $B$ is mapped (using a hash function discussed later) to an integer used as an index to the SHIFT table. The values in the SHIFT table determine how far we can shift forward (skip) while we scan the text. Let $X = x_1 \cdots x_B$ be the $B$ characters in the text that we are currently scanning, and assume that $X$ is mapped into the $i$'th entry of *SHIFT*. There are two cases:

1. $X$ does not appear as a substring in any pattern of $P$

   In this case, we can clearly shift $m-B+1$ characters in the text. Any smaller shift will put the last $B$ characters of the text against a substring of one of the patterns which is a mismatch. We store in $SHIFT[i]$ the number $m-B+1$.

2. $X$ appears in some patterns

   In this case, we find the rightmost occurrence of $X$ in any of the patterns; let's assume that $X$ ends at position $q$ of $P_j$ and that $X$ does not end at any position greater than $q$ in any other pattern. We store $m-q$ in $SHIFT[i]$.

To compute the values of the SHIFT table, we consider each pattern $p_i = a_1 a_2 \cdots a_m$ separately. We map each substring of $p_i$ of size $B$ $a_{j-B+1} \cdots a_j$ into SHIFT, and set the corresponding value to the minimum of its current value (the initial value for all of them is $m-B+1$) and $m-j$ (the amount of shifting required to get to this substring).

The values in the SHIFT table are the largest possible safe values for shifts. Replacing any of the entries in the SHIFT table with a *smaller* value will make less shifts and will take more time, but it will still be safe: no match will be missed. So we can use a compressed table for SHIFT, mapping several different strings into the same entry as long as we set the minimal shift of all of them as the value. In agrep, we actually do both. When the number of patterns is small, we select $B=2$, and use an exact table for SHIFT; otherwise, we select $B=3$ and use a compressed table for SHIFT. In either case, the algorithm is oblivious to whether or not the SHIFT table is exact.

As long as the shift value is greater than 0, we can safely shift and continue the scan. This is what happens most of the time. (In a typical example, the shift value was zero 5% of the time for 100 patterns, 27% for 1000 patterns, and 53% for 5000 patterns.) Otherwise, it is possible that the current substring in the text matches some pattern in the pattern list. But which pattern? To avoid comparing the substring to every pattern in the pattern list, we use a hashing technique to minimize the number of patterns to be compared. We already computed a mapping of the $B$ characters into an integer that is used as an index to the SHIFT table. We use the exact same integer to index into another table, called HASH. The $i$'th entry of the HASH table, HASH[$i$], contains a pointer to a list of patterns whose last $B$ characters hash into $i$. The HASH table will typically be quite sparse, because it holds only the patterns whereas the SHIFT table holds all possible strings of size $B$. This is an inefficient use of memory, but it allows us to reuse the hash function (the mapping), thus saving a lot of time. (It is also possible to make the hash table a power of 2 fraction of the SHIFT table and take just the last bits of the hash function.)

Let $h$ be the hash value of the current suffix in the text and assume that $SHIFT[h]=0$. The value of $HASH[h]$ is a pointer $p$ that points into two separate tables at the same time: We keep a list of pointers to the patterns, PAT_POINT, sorted by the hash values of the last $B$ characters of each pattern. The pointer $p$ points to the *beginning* of the list of patterns whose hash value is $h$. To find the end of this list, we keep incrementing this pointer until it is equal to the value in $HASH[h+1]$ (because the whole list is sorted according to the hash values). So, for example, if $SHIFT[h] \neq 0$, then $HASH[h]$ = $HASH[h+1]$ (because no pattern has a suffix that hash to $h$). In addition, we keep a table called PREFIX, which will be described shortly.

Natural language texts are not random. For example, the suffixes ‘ion’ or ‘ing’ are very common in English texts. These suffixes will not only appear quite often in the text, but they are also likely to appear in several of the patterns. They will cause collisions in the HASH table; that is, all patterns with the same suffix will be mapped to the same entry in the HASH table. When we encounter such a suffix in the text, we will find that its SHIFT value is 0 (assuming it is a suffix of some patterns), and we will have to examine separately all the patterns with this suffix to see if they match the text. To speed up this process, we introduce yet another table, called PREFIX.

In addition to mapping the last $B$ characters of all patterns, we also map the *first* $B'$ characters of all patterns into the PREFIX table. (We found that $B' = 2$ is a good value.) When we find a SHIFT value of 0 and we go to the HASH table to determine if there is a match, we check the values in the PREFIX table. The HASH table not only contains, for each suffix, the list of all patterns with this suffix, but it also contains (hash values of) their prefixes. We compute the corresponding prefix in the text (by shifting $m - B'$ characters to the left) and use it to filter patterns whose suffix is the same but whose prefix is different. This is an effective filtering method because it is much less common to have different patterns that share the same prefix *and* the same suffix. It is also a good ‘balancing act’ in the sense that the extra work involved in computing the hash function of the prefixes is significant only if the SHIFT value is often 0, which occurs when there are many patterns and a higher likelihood of collisions.

The preprocessing stage may seem quite involved, but in practice it is done very quickly. In our implementation, we set the size of the 3 tables to $2^{15} = 32768$. Running the match algorithm on a near empty text file, which is a measure of the time it takes to do the preprocessing, took only 0.16 seconds for 10,000 patterns.

## 2.3. The Scanning Stage

We now describe the scanning stage in more detail and give a partial code for it. The main loop of the algorithm consists of the following steps:

1. compute a hash value $h$ based on the current $B$ characters from the text (starting with $t_{m-B+1} \cdots t_m$).
2. check the value of SHIFT[h]: if it is > 0, shift the text and go back to 1; otherwise, go to 3.
3. Compute the hash value of the prefix of the text (starting $m$ characters to the left of the current position); call it text_prefix.
4. Check for each $p$, $HASH[h] \leq p < HASH[h+1]$ whether PREFIX[p] = text_prefix. When they are equal, check the actual pattern (given by PAT_POINT[p]) against the text directly.

A partial C code for the main loop is given in Figure 1. The complete code is available by anonymous ftp from cs.arizona.edu as part of the glimpse code (check the file mgrep.c in directory agrep).

```
while (text <= textend) {
    h = (*text<<Hbits)+(*(text-1));  /* The hash function (we use Hbits=5) */
    if (Long) h = (h<<Hbits)+(*(text-2)); /* Long=1 when the number of patterns warrants B=3 */
    shift = SHIFT[h];  /* we use a SHIFT table of size 2^15 = 32768 */
    if (shift == 0) { /* ‘‘h = h & mask_hash’’ can be used here if HASH is smaller then SHIFT */
        text_prefix = (*(text-m+1)<<8) + *(text-m+2);
        p = HASH[h];
           p_end = HASH[h+1];
           while (p++ < p_end) { /* loop through all patterns that hash to the same value (h) */
                if(text_prefix != PREFIX[p]) continue;
                px = PAT_POINT[p];
                qx = text-m+1;
                while (*(px++) == *(qx++)); /* check the text against the pattern directly */
                if (*(px-1) == 0) { /* 0 indicates the end of a string */
                     report a match
                }
                shift = 1;
           }
      text += shift;
}
```

**Figure 1**: Partial code for the main loop.

# 3. Performance

## 3.1. A Rough Analysis of the Running Time

We present an estimate for the running time of this algorithm assuming that both the text and the patterns are random strings with uniform distribution. In practice, texts and patterns are not random, but this estimate gives a rough idea about the performance of the algorithm. We show that the expected running time is less than linear in the size of the text (but not by much).

Let $N$ be the size of the text, $P$ the number of patterns, $m$ the size of one pattern, $M = mP$ the total size of all patterns, and assume that $N \geq M$. Let $c$ be the size of the alphabet. We define the size of the block used to address the SHIFT table as $B$=$\log_c 2M$. The SHIFT table contains all possible strings of size $b$, so there are $c^b = c^{\lceil \log_c 2M \rceil} \leq 2Mc$ entries in the SHIFT table. The SHIFT table is constructed in time $O(M)$ because each substring of size $B$ of any pattern is considered once and it takes constant time on the average to consider it. We divide the scanning time into two cases. The first case if when the SHIFT value is non-zero; in this case a shift is applied and no more work needs to be done at that position in the text. The second, and more complicated, case is when the SHIFT value is 0; in this case we need to read more of the pattern and the text and consult the HASH table.

**Lemma 1:** The probability that a random string of size $B$ leads to a shift value of $i$, $0 \le i \le m - B + 1$, is $\le 1/2m$.

**Proof:** At most $P = M/m$ strings lead to a SHIFT value of $i$ for $0 \le i \le m - B + 1$. But the number of all possible strings of size $B$ is at least $2M$. So the probability of one random string to lead to a SHIFT value of $i$ is $\le 1/2m$. Notice that this is true for SHIFT value 0 as well. □

Lemma 1 implies that the expected value of a shift is $\ge m/2$. Since it takes $O(B)$ to compute one hash function, the total amount of work in the cases of non-zero shifts is $O(BN/m)$. The extra filtering by the prefixes makes the probability of false hits extremely small. More precisely, let's assume that $B' = B$. The probability that a given pattern has the same prefix and suffix as another pattern is $< 1/M$, which is insignificant. Therefore, the amount of work for the case of shift value 0 is also $O(B)$ unless there is actually a match (in which case we need to check the whole pattern taking time $O(m)$). Since shift value 0 occurs $< 1/2m$ of the time (by lemma 1), the expected total amount for this step is also $O(BN/m)$.

## 4. Experiments

In this section we present several experiments comparing our algorithm to existing algorithms and evaluating the effects of the number and size of patterns on the performance. Unless the patterns are very small or there are very few of them, our algorithm is significantly faster. All experiments were conducted on a Sun SparcStation 10 model 510, running Solaris. All times are elapsed times (on a lightly loaded system getting more than 90% of the CPU) given in seconds; each experiment was performed 10 times and the averages are given (the deviations were very small). The text file we used for all experiments was a collection of articles from the Wall Street Journal totaling 15.8MB. The patterns were words from the file (all patterns appeared in the text).

Table 1 compares our algorithm, labeled *agrep*, against four other search routines: the original egrep and fgrep, GNU-grep version 2.0 [Ha93], and *gre*, an older program written by Andrew Hume (which at the time was the only program that could handle large number of patterns). The patterns were words of sizes ranging from 5 to 15 with average size slightly above 6. The original egrep and fgrep could not handle (or took too long for) more than few hundreds patterns.

In the second experiment, we measured the running times of agrep for different number of patterns ranging from 1000 to 10,000. The running time is indeed improved once the number of patterns exceeds about 8,000. The reason for that is very simple; it is related more to the way greps work rather than to the specific algorithm. Agrep (and every other grep) outputs the lines that match the query. Once it is established that a line should be output, there is no need to search further in that line. Above 8,000, the number of patterns becomes so large, most lines are matched and matched early on. So less work is needed to match the rest of the lines. We present this as an example of misleading performance measures; we probably would not have thought about this effect if the numbers had not actually gone down.

| # of patterns | egrep | fgrep | GNU-grep | gre | agrep |
|---|---|---|---|---|---|
| 10 | 6.54 | 13.57 | 2.83 | 5.66 | 2.22 |
| 50 | 8.22 | 12.95 | 5.63 | 9.67 | 2.93 |
| 100 | 16.69 | 13.27 | 6.69 | 11.88 | 3.31 |
| 200 | 42.62 | 13.51 | 8.12 | 14.38 | 3.87 |
| 1000 | - | - | 12.18 | 23.14 | 5.79 |
| 2000 | - | - | 15.80 | 28.36 | 7.44 |
| 5000 | - | - | 21.82 | 38.09 | 11.61 |

**Table 1**: A comparison of different search routines on a 15.8MB text.

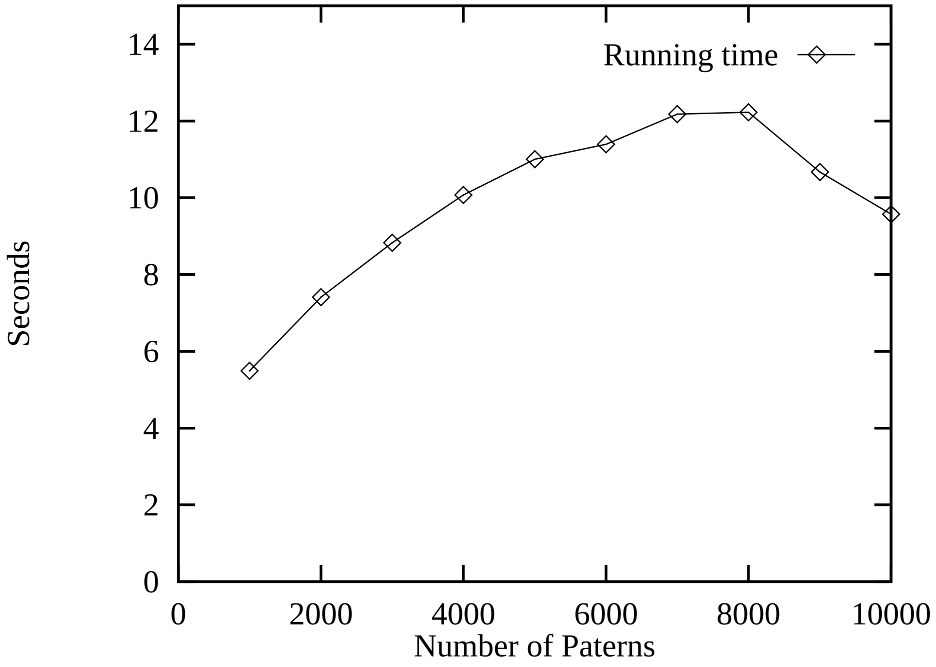

**Figure 2**: A comparison of running times for different number of patterns.

In the third experiment, we measured the effect of the sizes of the minimum pattern (denoted by $m$ in the discussion on the algorithm). The larger $m$ is the more chances of shifting there are, leading to less work. We used 100 patterns for each case, with the average size being typically 1 more than the minimal size. The graph matches quite well the curve of the function $1/(m-c)$, where $c$ is a small constant such that $m-c$ is the average shift value.

We also measured the time used for preprocessing a large set of patterns (by running agrep and GNU-grep on an empty text). The preprocessing for both programs are very fast up to 1000 patterns,

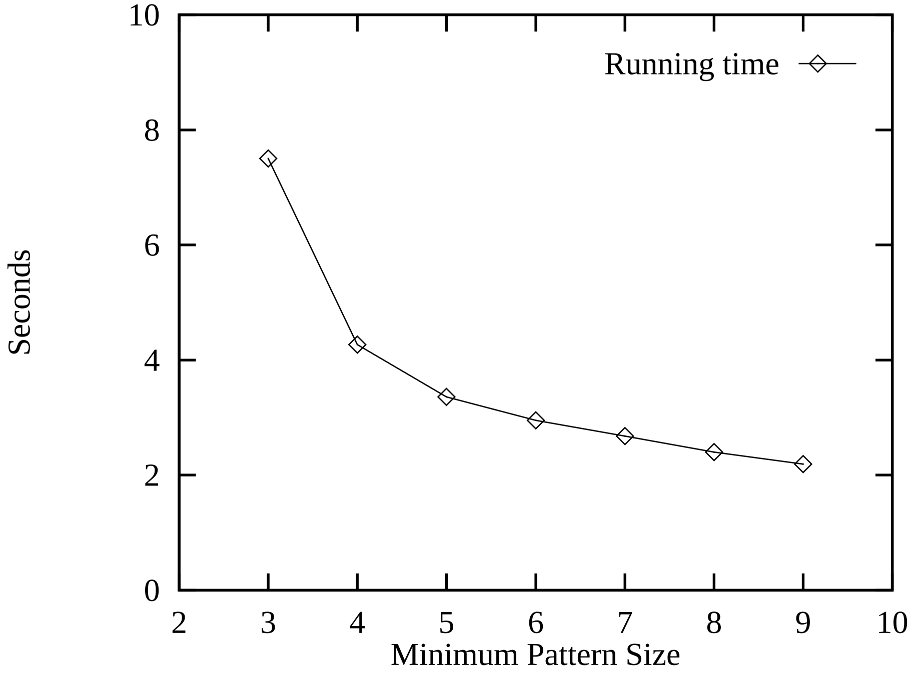

**Figure 3**: The effect of the minimum pattern length on the running time.

taking on the order of 0.1 second (once the patterns are in the cache). For more patterns, GNU-grep becomes slower (because of the added complexity of building tries); for 10,000 patterns, agrep takes about 0.17 seconds whereas GNU-grep requires about 0.90 seconds.

## 5. Additional Applications

In another project, to find all similar files in a large file system [Ma94], we needed a data structure to handle the following type of searches. We had a large set of (typically 100,000 to 500,000) small records, each identified by a unique integer. The main operation was to retrieve several records (typically 50-100, but sometimes as high as 1000) given their identifiers. The data structure needed to be stored on disk, and the operation above was triggered by the user. Searching for unique keywords is one of the most basic data structure problems and there are many options to handle it; the most common techniques use hashing or tree structures. But since we have to search many keywords at the same time, even with efficient hashing, most of the pages will be fetched anyway. And if the data is fetched from disk anyway, multi-pattern matching provides a very effective and simple solution. We stored the records as we obtained them without sorting them or providing any other structure, putting one record together with its identifier per line. Then we used *agrep* with the multi-pattern capabilities. The benefits of this approach are 1) no need for any additional space for the data structure, 2) no need for preprocessing or organizing the data structure (e.g., sorting), and 3) more flexible search; for example, the keywords can be any strings, Of course, if the total number of pages far exceeds the number of keywords that need to be searched, then other data structures will be better. But for small to medium size applications, this is a surprisingly good solution. We believe that it can be used in a variety of similar situations. Another example is intersection problems. The common method for finding all common records to two files is to sort both files and merge. With this approach, there is no need to sort, and the preprocessing is done only on the small file.

Another applications is a general ‘‘match-and-replace’’ utility, called *mar*.[2] Each pattern is associated with a replacement pattern. When a pattern is discovered it is replaced in the output by its replacement. One subtle detail in the algorithm is that we now must shift by the length of the matched pattern after it is discovered (instead of by 1) to avoid overlapping replacements. For example, if ‘war’ is replaced by ‘peace’ and ‘art’ is replaced by ‘science’, and we shift by 1 after seeing ‘war’, we will replace ‘wart’ by ‘peacescience’. (Of course, there is also an option to require that only complete words are replaced.) Since we can support thousands of patterns, *mar* can be used for wholesale translations (e.g., from English to American) very fast.

## References

[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.

[AG+90] Altschul S. F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, ‘‘Basic local alignment search tool,’’ *J. Molecular Biology* **15** (1990), pp. 403–410.

[Ba89] Baeza-Yates R. A., ‘‘Improved string searching,’’ *Software — Practice and Experience* **19** (1989), pp. 257–271.

[BM77] Boyer R. S., and J. S. Moore, ‘‘A fast string searching algorithm,’’ *Communications of the ACM* **20** (October 1977), pp. 762–772.

[CW79] Commentz-Walter, B, ‘‘A string matching algorithm fast on the average,’’ *Proc. 6th International Colloquium on Automata, Languages, and Programming* (1979), pp. 118–132.

[Ha93] Haertel, M., ‘‘Gnugrep-2.0,’’ Usenet archive *comp.sources.reviewed,* Volume 3 (July, 1993).

[Ho80] Horspool, N., ‘‘Practical Fast Searching in Strings,’’ *Software — Practice and Experience,* **10** (1980).

[Hu91] Hume A., personal communication (1991).

[Ma94] U. Manber, ‘‘Finding Similar Files in a Large File System,’’ *Usenix Winter 1994 Technical Conference,* San Francisco (January 1994), pp. 1–10.

[MW94] U. Manber and S. Wu, ‘‘GLIMPSE: A Tool to Search Through Entire File Systems,’’ *Usenix Winter 1994 Technical Conference,* San Francisco (January 1994), pp. 23–32.

[2] We expect to make *mar* available by anonymous ftp in a few weeks.

[WM92a] Wu S., and U. Manber, ‘‘Agrep — A Fast Approximate Pattern-Matching Tool,’’ *Usenix Winter 1992 Technical Conference,* San Francisco (January 1992), pp. 153–162.

[WM92b] Wu S., and U. Manber, ‘‘Fast Text Searching Allowing Errors,’’ *Communications of the ACM* **35** (October 1992), pp. 83–91.