AGREP — A FAST APPROXIMATE PATTERN-MATCHING TOOL

(Preliminary version)

Sun Wu and Udi Manber[1]

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

ABSTRACT

Searching for a pattern in a text file is a very common operation in many applications ranging from text editors and databases to applications in molecular biology. In many instances the pattern does not appear in the text exactly. Errors in the text or in the query can result from misspelling or from experimental errors (e.g., when the text is a DNA sequence). The use of such approximate pattern matching has been limited until now to specific applications. Most text editors and searching programs do not support searching with errors because of the complexity involved in implementing it. In this paper we describe a new tool, called *agrep*, for approximate pattern matching. Agrep is based on a new efficient and flexible algorithm for approximate string matching. Agrep is also competitive with other tools for exact string matching; it include many options that make searching more powerful and convenient.

1. Introduction

The most common string-searching problem is to find all occurrences of a string $P = p_1 p_2 ... p_m$ inside a large text file $T = t_1 t_2 \cdots t_n$. We assume that the string and the text are sequences of *characters* from a finite character set Σ. The characters may be English characters in a text file, DNA base pairs, lines of source code, angles between edges in polygons, machines or machine parts in a production schedule, music notes and tempo in a musical score, etc. The two most famous algorithms for this problem are the Knuth-Morris-Pratt algorithm [KMP77] and the Boyer-Moore algorithm [BM77] (see also [Ba89] and [HS91]). There are many extensions to this problem; for example, we may be looking for a set of patterns, a regular expression, a pattern with ‘‘wild cards,’’ etc. String searching in Unix is most often done with the *grep* family.

In some instances, however, the pattern and/or the text are not exact. We may not remember the exact spelling of a name we are searching, the name may be misspelled in the text, the text may correspond to a sequence of numbers with a certain property and we do not have an exact pattern, the text may be a sequence of DNA molecules and we are looking for approximate patterns, etc. The approximate string-matching problem is to find all substrings in T that are *close* to P under some measure of closeness. We will concentrate here on the edit-distance measure (also known as the *Levenshtein measure*). A string P is said to be of distance k to a string Q if we can transform P to be equal to Q with a sequence of k insertions of single characters in (arbitrary places in) P,

[1] Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, and by an NSF grant CCR-9002351.

deletions of single characters in *P*, or substitutions of characters. Sometimes one wants to vary the cost of the different edit operations, say deletions cost 3, insertions 2, and substitutions 1.

Many different approximate string-matching algorithms have been suggested (too many to list here), but none is widely used, mainly because of their complexity and/or lack of generality. We present here a new tool, called *agrep* (for *approximate* grep), which has a very similar user interface to the grep family (although it is not 100% compatible), and which supports several important extensions to grep. Version 1.0 of agrep is available by anonymous ftp from cs.arizona.edu (IP 192.12.69.5) as agrep/agrep.tar.Z. It has been developed on a SUN SparcStation and has been successfully ported to DECstation 5000, NeXT, Sequent, HP 9000, and Silicon Graphics workstations. We expect version 2.0 to be available (at the same place) by the end of 1991; most of the discussion here refers to version 2. The three most significant features of agrep that are not supported by the grep family are 1) searching for approximate patterns, 2) searching for records rather than just lines, and 3) searching for multiple patterns with AND (or OR) logic queries. (All 3 features are available in version 1.0.) Other features include searching for regular expressions (with or without errors), efficient multi-pattern search, unlimited wild cards, limiting the errors to only insertions or only substitutions or any combination, allowing each deletion, for example, to be counted as, say, 2 substitutions or 3 insertions, restricting parts of the query to be exact and parts to be approximate, and many more. Examples of the use of agrep are given in the next section.

Agrep not only supports a large number of options, but it is also very efficient. In our experiments, agrep was competitive with the best exact string-matching tools that we could find (Hume's gre [Hu91] and GNU e?grep [Ha89]), and in many cases one to two orders of magnitude faster than other approximate string-matching algorithms. For example, finding all occurrences of *Homogenos* allowing two errors in a 1MB bibliographic text takes about 0.2 seconds on a SUN SparcStation II. (We actually used this example and found a misspelling in the bib file.) This is almost as fast as exact string matching.

This paper is organized as follows. We start by giving examples of the use of agrep that illustrate how flexible and general it is. We then briefly describe the main ideas behind the algorithms and their extensions. (More details are given in the technical report and man pages which are available by ftp.) We then give some experimental results, and we close with conclusions.

2. Using Agrep

We have been using agrep for about 6 months now and find it an indispensable tool. We present here only a sample of the uses that we found. As we said in the introduction, the three most significant features of agrep that are not supported by the grep family are

1. **the ability to search for approximate patterns**
   for example, `agrep -2 Homogenos bib` will find Homogeneous as well as any other word that can be obtained from Homogenos with at most 2 substitutions, insertions, or deletions. It is possible to assign different costs to insertions, deletions, or substitutions. For example, `agrep -1 -I2 -D2 555-3217 phone` will find all numbers that differ from 555-3217 in at most one digit. The -I (-D) option sets the cost of insertions (deletions); in this case, setting it to 2 prevents insertions and deletions.

2. **agrep is record oriented rather than just line oriented**
   a record is by default a line, but it can be user defined; for example, `agrep -d '^From ' 'pizza' mbox` outputs all mail messages that contain the keyword "pizza". Another example: `agrep -d '$$' pattern foo` will output all paragraphs (separated by an empty line) that contain pattern.

3. **multiple patterns with AND (or OR) logic queries**
   For example, `agrep -d '^From ' 'burger,pizza' mbox` outputs all mail messages containing at least one of the two keywords (',' stands for OR); `agrep -d '^From ' 'good;pizza' mbox`

outputs all mail messages containing both keywords (‘;’ stands for AND).

Putting these options together one can ask queries like

```
        agrep -d ’$$’ -1 ’<CACM>;TheAuthor;Curriculum;<198[5-9]>’ bib-file
```

which outputs all paragraphs referencing articles in CACM between 1985 and 1989 by TheAuthor dealing with curriculum. One error is allowed in any of the sub-patterns, but it cannot be in either CACM or the year (the <> brackets forbid errors in the pattern between them).

These features and several more enable users to compose complex queries rather easily. We give several examples of the daily use of agrep from our experience. For a complete list of options, see the manual pages distributed with agrep.

2.1. Finding words in a dictionary

The most common tool available in UNIX for finding the correct spelling of a word is the program *look*, which outputs all words in the dictionary with a given prefix. We have many times looked for spelling of words for which we did not know a prefix. We use the following alias for findword:

```
   alias findword agrep -i -!:2 !:1 /usr/dict/web2
```

(web2 is a large collection of words, about 2.5MB long; one can use /usr/dict/words instead.) For example, one of the authors can never remember the correct spelling of bureaucracy (and he is irritated enough with it not wanting to remember). `findword breacracy 2` searches for all occurrences of breacracy with at most two errors. (web2 contains one more match - squireocracy).

One can also use the -w option which matches the pattern to a complete word (rather than possibly a subword). In the example above, the extra match (squireocracy) will not be a match, because with the -w option its beginning (squi) will count as 4 extra errors.

2.2. Searching a Mail File

We found that one of the most frequent uses of agrep is to search inside mail files for mail messages using the record option. We use the following alias

```
   alias agmail agrep -!:2 -d ’ˆFrom ’ !:1
```

Notice that it is possible with this alias to use complicated queries; for example,
`agmail ’<pizza>;<great>;Manbar’ 1 mail/food,` or
`agmail ’\.gov;October;surprise’ 0 mail/*,` which searches all mail messages from .gov (a . without the \ matches every character) that include the two keywords.

2.3. Extracting Procedures

It is usually possible to easily extract a procedure from a large program by defining a procedure as a record and using agrep. For example, `agrep -t -d ’ˆ}’ ’ˆroutine1’ prog1/*.c > routine1.c` will work assuming routines in C always end with } at the beginning of a line (and that ’ˆroutine1’ uniquely identifies that routine). One should be careful when dealing with other people’s programs (because the conventions may not be followed). Other programming languages have other ways to identify the end (or beginning of a procedure). The -t option puts the record delimiter at the end of the record rather than at the beginning (which is more appropriate for mail messages, for example).

2.4. Finding Interesting Words

At some point we needed to find all words in the dictionary with 4-7 characters. This can be done with one agrep command `agrep -3 -w -D4 ’....’ /usr/dict/words.` (The -D4 prevents deletions, and the . in the pattern stands for any character.)

We end this section with a cute example, which although is not important, shows how flexible agrep can be. The following query finds all words in the dictionary that contain 5 of the first 10 letters of the alphabet in order: `agrep -5 'a#b#c#d#e#f#g#h#i#j' /usr/dict/words` (the # symbol stands for a wild card of any size - the same as .*). Try it. The answer starts with the word *academia* and ends with *sacrilegious*; it must mean something..

3. The Algorithms

Agrep utilizes several different algorithms to optimize the performance for the different cases. For simple exact queries we use a variant of the Boyer-Moore algorithm. For simple patterns with errors, we use a partition scheme, described at the end of section 3.2, hand in hand with the Boyer-Moore scheme. For more complicated patterns, e.g., patterns with unlimited wild cards, patterns with uneven costs of the different edit operations, multi-patterns, arbitrary regular expressions, we use new algorithms altogether. In this section, we briefly outline the basis for two of the interesting new algorithms that we use, the algorithm for arbitrary patterns with errors and the algorithm for multi patterns. For some more details on the algorithms see [WM91, WM92].

3.1. Arbitrary Patterns With Errors

We describe only the main idea behind the simplest case of the algorithm, finding all occurrences of a given string in a given text. The algorithm is based on the 'shift-or' algorithm of Baeza-Yates and Gonnet [BG89]. Let R be a bit array of size m (the size of the pattern). We denote by R_j the value of the array R after the j character of the text has been processed. The array R_j contains information about all matches of prefixes of P with a suffix of the text that ends at j. More precisely, $R_j[i] = 1$ if the first i characters of the pattern match exactly the last i characters up to j in the text. These are all the partial matches that may lead to full matches later on. When we read t_{j+1} we need to determine whether t_{j+1} can extend any of the partial matches so far. The transition from R_j to R_{j+1} can be summarized as follows:

Initially, $R_0[i] = 0$ for all i, $1 \le i \le m$; $R_0[0] = 1$.

$$R_{j+1}[i] = \begin{cases} 1 & \text{if } R_j[i-1] = 1 \text{ and } p_i = t_{j+1} \\ 0 & \text{otherwise} \end{cases}$$

If $R_{j+1}[m] = 1$ then we output a match that ends at position $j+1$;

This transition, which we have to compute once for every text character, seems quite complicated. However, suppose that $m \le 32$ (which is usually the case in practice), and that R is represented as a bit vector using one 32-bit word. For each character s_i in the alphabet we construct a bit array S_i of size m such that $S_i[r] = 1$ if $p_r = s_i$. (It is sufficient to construct the S arrays only for the characters that appear in the pattern.) It is easy to verify now that the transition from R_j to R_{j+1} amounts to no more than a *right shift* of R_j and an AND operation with S_i, where $s_i = t_{j+1}$. So, each transition can be executed with only two simple arithmetic operations, a shift and an AND.[2]

Suppose now that we want to allow one substitution error. We introduce one more array, denoted by R_j^1, which indicates all possible matches up to t_j with at most one substitution. The transition for the R array is the same as before. We need only to specify the transition for R^1. There are two cases for a match with at most one substitution of the first i characters of P up to t_{j+1}:

[2] We assume that the right shift fills the first position with a 1. If only 0-filled shifts are available (as is the case with C), then we can add one more OR operation with a mask that has one bit. Alternatively, we can use 0 to indicate a match and an OR operation instead of an AND; that way, 0-filled shifts are sufficient. This is counterintuitive to explain (and it is not adaptable to some of the extensions), so we opted for the easier definition.

S1. There is an exact match of the first $i-1$ characters up to t_j This case corresponds to substituting t_{j+1} with p_i (whether or not they are equal — the equality will be indicated in R) and matching the first $i-1$ characters.

S2. There is a match of the first $i-1$ characters up to t_j with one substitution *and* $t_{j+1} = p_i$.

It turns out that both cases can be handled with two arithmetic operations on R^1. If we allow insertions, deletions, and substitutions, then we will need 4 operations on R^1. If we want to allow more than one error, then we maintain more than one additional R^1 array. Overall, the number of operations is proportional to the number of errors. But we can do even better than that.

Suppose again that the pattern P is of size m and that at most k errors are allowed. Let $r = \lfloor \frac{m}{k+1} \rfloor$; divide P into $k+1$ blocks each of size r and call them $P_1, P_2, ..., P_{k+1}$. If P matches the text with at most k errors, then at least one of the P_j's must match the text exactly. We can search for all P_j's at the same time (we discuss how to do that in the next paragraph) and, if one of them matches, then we check the whole pattern directly (using the previous scheme) but only within a neighborhood of size m from the position of the match. Since we are looking for an exact match, there is no need to maintain the additional vectors. This scheme will run fast if the number of exact matches to any one of the P_j's is not too high.

The main advantage of this scheme is that the algorithm for exact matching can be adapted in an elegant way to support it. We illustrate the idea with an example. Suppose that the pattern is ABCDEFGHIJKL ($m = 12$) and $k = 3$. We divide the pattern into $k+1 = 4$ blocks: ABC, DEF, GHI, and JKL. We need to find whether any of them appears in the text. We create one combined pattern by interleaving the 4 blocks: ADGJBEHKCFIL. We then build the mask vector R as usual for this interleaved pattern. The only difference is that, instead of shifting by one in each step, we shift by four! There is a match if any of the last four bits is 1. (When we shift we need to fill the first four positions with 1's, or better yet, use shift-OR.) Thus, the match for all blocks can be done exactly the same way as regular matches and it takes essentially the same running time.

The algorithm described so far is efficient for simple string matching. But more important, it is also adaptable to many extensions of the basic problem. For example, suppose that we want to search for ABC followed by a digit, which is defined in agrep by ABC[0-9]. The only thing we need to do is in the preprocessing, for each digit, allow a match at position 4. Everything else remains exactly the same. Other extensions include arbitrary wild cards, a combination of patterns with and without errors, different costs for insertions, deletions, and/or substitutions, and probably the most important, arbitrary regular expressions. We have no room to describe the implementation of these extensions (see [WM91]). The main technique is to use additional masking and preprocessing. It is sometimes relatively easy (as is the case with wild cards) and it sometimes requires clever ideas (as is the case with arbitrary regular expressions with errors). Next, we describe a very fast algorithm for multiple patterns which also leads to a fast approximate-matching algorithm for simple patterns.

3.2. An Algorithm for Multi Patterns

Suppose that the pattern consists of a set of k simple patterns $P_1, P_2, ..., P_k$, such that each P_i is a string of m characters from a fixed alphabet Σ. The text is again a large string T of characters from Σ. (We assume that all subpatterns have the same size for simplicity of description only; agrep makes no such assumption.) The *multi-pattern string matching problem* is to find all substrings in the text that match at least one of the patterns in the set.

The first efficient algorithm for solving this problem is by Aho and Corasick [AC75], which solves the problem in linear time. (This algorithm is the basis of *fgrep*.) Commentz-Walter [CW79] presented an algorithm which combines the Boyer-Moore technique with the Aho-Corasick algorithm. The Commentz-Walter Algorithm is substantially faster than the Aho-Corasick algorithm when the number of patterns is not too big. The pattern matching tool *gre* [Hu91] (which covers almost all functions of egrep/grep/fgrep) developed by Andrew Hume has incorporated the Commentz-Walter algorithm for the multi-pattern string matching problem.

Our algorithm uses a hashing technique combined with a different Boyer-Moore technique. Instead of building a shift table based on single characters, we build the shift table based on a block of characters. (The idea of using a block of characters was first proposed by Knuth-Morris-Pratt in section 8 of [KMP77].) Like other Boyer-Moore style algorithms, our algorithm preprocesses the patterns to build some data structures such that the search process can be speeded up. Let c denote the size of the alphabet, $M = k \cdot m$, and $b = \lceil \log_c M \rceil$. We assume that $b \le m/2$. In the preprocessing, a shift table *SHIFT* and a hashing table *HASH* are built. We look at the text b characters at a time. The values in the SHIFT table determine how far we can shift forward during the search process. The shift table *SHIFT* is an array of size c^b. Each entry of *SHIFT* corresponds to a distinct substring of length b. Let $X = x_1 x_2 \cdots x_b$ be a string corresponding to the i'th entry of *SHIFT*. There are two cases: either X appears somewhere in one of the P_j's or not. For the latter case, we store $m - b + 1$ in $SHIFT[i]$. For the former case, we find the rightmost occurrence of X in any of the P_j's that contain it; suppose it is in P_y and that X ends at position q of P_y. Then we store $m - q$ in $SHIFT[i]$.

If the shift value is > 0, then we can safely shift. Otherwise, it is possible that the current substring we are looking at in the text matches some pattern in the pattern list. 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. In the preprocessing we build a hash table *HASH* such that a pattern with hash value j is put in a linked-list pointed to by $HASH[j]$. So, in the search process, whenever we are going to compare current aligned substring to the patterns, we first compute the hash value of the substring and compare the substring only to those patterns that have the same hash value. The algorithm for searching the text is sketched in Figure 1.

The multi-pattern matching algorithm described above can be used to solve the approximate string-matching problem. Let $P = p_1, p_2, ..., p_M$ be a pattern string, and let $T = a_1, a_2, ... a_N$ be a text string. We partition P into $k + 1$ fragments $P_1, P_2, ..., P_{k+1}$, each of size $m = M/(k+1)$. Let $T_{ij} = a_i, ..., a_j$ be a substring of T. By a pigeonhole principle, if T_{ij} differs from P by no more than k errors, then one of the fragment must match a substring of T_{ij} exactly.

The approximate string matching algorithm is conducted in two phases. In the first phase we partition the pattern into $k + 1$ fragments and use the multi-pattern string matching algorithm to find all those places that contain one of the fragments. If there is a match of a fragment at position i of the text, we mark the positions $i - M - k$ to $i + M + k - m$ as a 'candidate' area. After the first phase is done we apply the approximate matching algorithm described in section 3.1 to find the actual matches in those marked area. (If the pattern size is > 32, we use

```
Algorithm Multi-Patterns
        Let $p$ be the current position of the text ;
        while ($p < N$)  /* $N$ is the end position of the text */
        {
            $blk\_idx = map(a_{p-b+1} a_{p-b+2} \cdots a_p)$ /* map transforms a string of size $b$ into an integer */
            $shift\_value = SHIFT[blk\_idx]$;
            if ($shift\_value > 0$) $p = p + shift\_value$;
            else
                compute the hash value of $a_{p-m+1} \cdots a_p$;
                compare $a_{p-m+1} \cdots a_p$ to every pattern that has the same hash value;
                if there is a match then reports $a_{p-m+1} \cdots a_p$;
                $p = p + 1$;
        }
```

Figure 1: A sketch of the algorithm for multi-pattern searching.

Ukkonen's $O(Nk)$ expected-time algorithm [Uk85].)

Our algorithm is very fast when the pattern is long and the number of errors is not high (assuming that $k < M/\log_b M$). Unlike the approximate Boyer-Moore string matching algorithm in [TU90], whose performance degrades greatly when the size of the alphabet set is small, our algorithm is not sensitive to the alphabet size. For example, for DNA patterns of size 500, allowing 25 errors, our algorithm is about two orders of magnitude faster than Ukkonen's $O(Nk)$ expected-time algorithm [Uk85] and algorithm MN2 [GP90] (which are the two fastest algorithms among the algorithms compared in [CL90]). Experimental results are given in the next section. The algorithm is very fast for practical applications for searching both English text and DNA data.

4. Experimental Results

We present four brief experiments. The numbers given here should be taken with caution. Any such results depend on the architecture, the operating system, the compilers used, not to mention the patterns and test files. These tests are by no means comprehensive. They are given here to show that agrep is fast enough to be used for large files. Differences of 20-30% in the running times are not significant. Thus, all Boyer-Moore type programs are about the same for simple patterns. Agrep seems better for multi patterns. For approximate matching, agrep is one to two orders of magnitude faster than other programs that we tested. We believe that the main strength of agrep is that it is more flexible, general, and convenient than all previous programs.

All tests were run on a SUN SparcStation II running UNIX. Two files were used, both of size 1MB, one a sub-dictionary and one a collection of bibliographic data. The numbers are in seconds and are the averages of several experiments. They were measured by the time facility in UNIX and only user times were taken (which adds considerably to their impreciseness).

Table 1 compares agrep against other programs for *exact* string matching. The first three programs use Boyer-Moore type algorithms. The original egrep does not. We used 50 words of varying sizes (3-12) as patterns and averaged the results.

text size	agrep	gre	e?grep	egrep
1Mb	0.09	0.11	0.11	0.79
200Kb	0.028	0.024	0.038	0.218

Table 1: Exact matching of simple strings.

Table 2 shows results of searching for multi patterns. In the first line the pattern consisted of 50 words (the same words that were used in Table 1, but all in once) searched inside a dictionary; in the second line the pattern consists of 20 separate titles (each two words long), searched in a bibliographic file.

pattern	agrep	gre	e?grep	fgrep
50 words	1.15	2.57	6.11	8.13
20 titles	0.18	0.71	1.53	5.64

Table 2: Exact matching of multi patterns.

Table 3 shows typical running times for approximate matching. Two patterns were used — 'matching' and 'string matching' — and we tested each one with 1, 2, and 3 errors (denoted by Er). Other programs that we tested did not come close.

pattern	Er = 1	Er = 2	Er = 3
‘string matching’	0.26	0.55	0.76
‘matching’	0.22	0.66	1.14

Table 3: Approximate matching of simple strings.

Table 4 shows typical running times for more complicated patterns, including regular expressions. Agrep does not yet use any Boyer-Moore type filtering for these patterns. As a result, the running times are slower than they are for simpler patterns. The best algorithm we know for approximate matching to arbitrary regular expressions is by Myers and Miller [MM89]. Its running times for the cases we tested were more than an order of magnitude slower than our algorithm, but this is not a fair test, because Myers and Miller’s algorithm can handle arbitrary costs (which we cannot handle) and its running time is independent of the number of errors (which makes it competitive with or better than ours if the number of errors is very large).

pattern	Er = 0	Er = 1	Er = 2	Er = 3
<Hom>ogenious	0.53	1.10	1.42	1.74
JACM; 1981; Graph	0.53	1.10	1.43	1.75
Prob#tic; Algo#m	0.55	1.10	1.42	1.76
<[CJ]ACM>; Prob#tic; trees	0.54	1.11	1.43	1.75
(<[23]>–[23]*|<B>).*<Tr>ees	0.66	1.53	2.19	2.83

Table 4: Approximate matching of complicated patterns.

5. Conclusions

Searching text in the presence of errors is commonly done ‘by hand’ — one tries many possibilities. This is frustrating, slow, and with no guarantee of success. Agrep can alleviate this problem and make searching in general more robust. It also makes searching more convenient by not having to spell everything precisely. Agrep is very fast and general and it should find numerous applications. It has already been used in the Collaboratory system [HPS90], in a new tool (under development) for locating files in a UNIX system [FMW92], and in a new algorithm for finding information in a distributed environment [FM92]. In the last two applications, agrep is modified in a novel way to search inside specially compressed files *without* having to decompress them first.

Acknowledgements:

We thank Ricardo Baeza-Yates, Gene Myers, and Chunghwa Rao for many helpful conversations about approximate string matching and for comments that improved the manuscript. We thank Ric Anderson, Cliff Hathaway, Andrew Hume, David Sanderson, and Shu-Ing Tsuei for their help and comments that improved the implementation of agrep. We also thank William Chang and Andrew Hume for kindly providing programs for some of the experiments.

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.

[Ba89]

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

[BG89]

Baeza-Yates R. A., and G. H. Gonnet, ‘‘A new approach to text searching,’’ *Proceedings of the 12th Annual ACM-SIGIR conference on Information Retrieval,* Cambridge, MA (June 1989), pp. 168−175.

[BM77]

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

[CL90]

Chang W. I., and E. L. Lawler, ‘‘Approximate string matching in sublinear expected time,’’ *the 31th Annual Symp. on Foundations of Computer Science,* (October 1990), pp. 116−124.

[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.

[FM92]

Finkel R. A., and U. Manber, ‘‘The design and implementation of a server for retrieving distributed data,’’ in preparation.

[FMW92]

Finkel R. A., U. Manber, and S. Wu, ‘‘Findfile — a tool for locating files in a large file system,’’ in preparation.

[GP90]

Galil Z., and K. Park, ‘‘An improved algorithm for approximate string matching,’’ *SIAM J. on Computing* **19** (December 1990), pp. 989−999.

[Ha89]

Haertel, M., ‘‘GNU e?grep,’’ Usenet archive `comp.source.unix,` Volume 17 (February 1989).

[HPS90]

Hudson, S. E., L. L. Peterson, and B. R. Schatz, ‘‘Systems Technology for Building a National Collaboratory,’’ University of Arizona Technical Report #TR 90-24 (July 1990).

[HS91]

Hume A., and D. Sunday, ‘‘Fast string searching,’’ *Software — Practice and Experience* **21** (November 1991), pp. 1221−1248.

[Hu91]

Hume A., personal communication (1991).

[KMP77]

Knuth D. E., J. H. Morris, and V. R. Pratt, ‘‘Fast pattern matching in strings,’’ *SIAM Journal on Computing* **6** (June 1977), pp. 323–350.

[MM89]
Myers, E. W., and W. Miller, ‘‘Approximate matching of regular expressions,’’ *Bull. of Mathematical Biology* **51** (1989), pp. 5–37.

[TU90]
Tarhio J. and E. Ukkonen, ‘‘Approximate Boyer-Moore string matching,’’ Technical Report #A-1990-3, Dept. of Computer Science, University of Helsinki (March 1990).

[Uk85]
Ukkonen E., ‘‘Finding approximate patterns in strings,’’ *Journal of Algorithms* **6** (1985), pp. 132–137.

[WM91]
Wu S. and U. Manber, ‘‘Fast Text Searching With Errors,’’ Technical Report TR-91-11, Department of Computer Science, University of Arizona (June 1991).

[WM92]
Wu S. and U. Manber, ‘‘Filtering search approach for some string matching problems,’’ in preparation.

Biographical Sketches

Sun Wu is a Ph.D. candidate in computer science at the University of Arizona. His research interests include design of algorithms, in particular, string matching and graph algorithms.

Udi Manber is a professor of computer science at the University of Arizona, where he has been since 1987. He received his Ph.D. degree in computer science from the University of Washington in 1982. His research interests include design of algorithms and computer networks. He is the author of ‘‘Introduction to Algorithms - A Creative Approach’’ (Addison-Wesley, 1989). He received a Presidential Young Investigator Award in 1985, the best paper award of the seventh International Conference on Distributed Computing Systems, 1987, and a Distinguished Teaching Award of the Faculty of Science at the University of Arizona, 1990.