Fast Text Searching With Errors

Sun Wu and Udi Manber

TR 91-11

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

FAST TEXT SEARCHING WITH ERRORS

Sun Wu and Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721

June 1991

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 present a new algorithm for approximate text searching which is very fast and very flexible. We believe that the new algorithm will find its way to many searching applications and will enable searching with errors to be just as common as searching exactly.

1. Introduction

The string-matching problem is a very common problem. We are searching for a pattern $P = p_1 p_2 ... p_m$ inside a large text file $T = t_1 t_2 \cdots t_n$. The pattern 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. We want to find all occurrences of P in T; namely, we are searching for the set of starting positions $F = \{ i \mid 1 \le i \le n-m+1$ such that $t_i t_{i+1} \cdots t_{i+m-1} = P \}$. The two most famous algorithms for this problem are the Knuth Morris Pratt algorithm [KMP77] and the Boyer-Moore algorithm [BM77]. There are many extensions to this problem; for example, we may be looking for a set of patterns,

---

[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.

a regular expression, a pattern with ‘‘wild cards,’’ etc. String-matching tools are included in every reasonable text editor and they serve many different applications.

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. The most common measure of closeness is known as the edit distance (also the Levenshtein measure [Le66]). 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*, deletions of single characters in *P*, or substitutions of characters. IN some cases we may want to define closeness differently. For example, a policeman may be searching for a license plate ABC123 with the knowledge that the letters are correct, but there may be an error with the numbers. In this case, a string is of distance 1 to ABC123 if only one error occurs and it is within the digits area. Maybe there are always 3 digits in a license plate, in which case only substitutions are allowed. 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 ([CL90], [GG88], [GP90], [HD80], [LV88], [LV89], [My86], [TU90], and [Uk85a] is a partial list). In this paper we present a new algorithm which is very fast in practice, reasonably simple to implement, and supports a large number of variations of the approximate string-matching problem. The algorithm is based on a numeric scheme for exact string matching developed by Baeza-Yates and Gonnet [BG89] (See also [BG91]). The algorithm can handle several variations of measures and most of the common types of queries, including arbitrary regular expressions. In our experiments, the algorithm was at least twice as fast as other algorithms we tested (which are not as flexible), and for many cases an order of magnitude faster. For example, finding all occurrences of *Homogenos* allowing two errors in a one megabyte bibliographic text takes about 0.4 seconds on a SUN SparcStation II, which is about twice as fast as running the program *egrep* (which will not find anything because of the misspelling). We actually used this example and found a misspelling in our text.

The paper is organized as follows. We first describe the algorithm for the pure string-matching problem (i.e., the pattern is a simple string). In Section 3, we present many variations and extensions of the basic algorithm, culminating with matching arbitrary regular expressions with errors. Experimental results are given in Section 4. In Section 5 we describe a tool called *agrep* for approximate string matching based on the algorithm. Agrep is available through anonymous ftp from cs.arizona.edu.

2. The Algorithm

We first describe the case of exact string matching. The algorithm for this case is identical with that of Baeza-Yates and Gonnet [BG89]. We then show how to extend the algorithm to search with errors. We then describe how to speed up the search with errors by using an exact search most of the time.

2.1. Exact Matching

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 that end 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 (i.e., $p_1 p_2 \cdots p_i = t_{j-i+1} t_{j-i+2} \cdots t_j$). When we read t_{j+1} we need to determine whether t_{j+1} can extend any of the partial matches so far. For each i such that $R_j[i]=1$ we need to check whether t_{j+1} is equal to p_{i+1}. If $R_j[i]=0$ then there is no match up to i and there cannot be a match up to $i+1$. If $t_{j+1}=p_1$ then $R_{j+1}[1]=1$. If $R_{j+1}[m]=1$ then we have a complete match, starting at $j-m+2$, and we output it. The transition from R_j to R_{j+1} can be summarized as follows:

Initially, $R_0[i]=0$ for all i, $1 \le i \le m$; $R_0[0]=1$ (to avoid having a special case for i=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 at $j-m+2$;

This transition, which we have to compute once for every text character, seems quite complicated. Other fast string-matching algorithms avoid the need to maintain the whole array by storing only the *best match* so far and some more information that depends on the pattern. The main observation about this transition, due to Baeza-Yates and Gonnet [BG89], is that it can be computed very fast in practice as follows. Let the alphabet be $\Sigma = s_1, s_2, ..., s_{|\Sigma|}$. 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.) In other words, S_i denotes the indices in the pattern that contain s_i. 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. 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. (Baeza-Yates and Gonnet [BG89] used 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, so we opted for the easier definition.) An example is given in Figure 1a, where the pattern is *aabac* and the text is *aabaacaabacab*. The masks for *a b* and *c* are given in Figure 1b.

The discussion above assumes, of course, that the pattern's size is no more than the word size, which is often the case. If the pattern's size is twice the word size, then 4 arithmetic

	a	a	b	a	a	c	a	a	b	a	c	a	b
a	1	1	0	1	1	0	1	1	0	1	0	1	0
a	0	1	0	0	1	0	0	1	0	0	0	0	0
b	0	0	1	0	0	0	0	0	1	0	0	0	0
a	0	0	0	1	0	0	0	0	0	1	0	0	0
c	0	0	0	0	0	0	0	0	0	0	1	0	0

a.

a	b	c
1	0	0
1	0	0
0	1	0
1	0	0
0	0	1

b.

Figure 1: An example of exact matching and the corresponding masks.

operations will suffice. Patterns of more than 64 characters are quite rare in practice, although there are applications for which they can appear. We discuss this issue further in section 3, but for now we'll assume that the pattern's size is no more than the word size. This algorithm is clearly very easy to implement. Its running time is totally predictable because it depends only on the size of the text (assuming again that the pattern fits into a word) and not on the actual text or the alphabet.

2.2. Matching With Errors

We now show how to adapt the previous algorithm to allow errors. (Baeza-Yates and Gonnet [BG89] showed how to handle only mismatches by essentially counting k of them with a $\log_2 k$ size counter, but they did not handle insertions or deletions.) We start with a very simple case: only one insertion is allowed into the pattern at any position. In other words, we want to find all intervals of size at most $m+1$ in the text that contain the pattern as a subsequence. We define the R and S arrays as before, but now we have two possibilities for each prefix match. We can have an exact match or a match with one insertion. Therefore, we introduce another array, denoted by R^1_j, which indicates all possible matches up to t_j with at most one insertion. More precisely, $R^1_j[i]=1$ if the first i characters of the pattern match i of the last $i+1$ characters up to j in the text. If we can maintain both R and R^1 then we can find all matches with at most one insertion: $R_j[m]=1$ indicates that there is an exact match and $R^1_j[m]=1$ indicates that there is a match with at most one insertion (sometimes both will equal to 1 at the same time).

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 insertion of the first i characters of P up to t_{j+1}:

I1. There is an exact match of the first i characters up to t_j. In this case, inserting t_{j+1} at the end of the exact match creates a match with one insertion.

I2. There is a match of the first $i-1$ characters up to t_j with one insertion *and* $t_{j+1}=p_i$. In this case, the insertion is somewhere inside the pattern and not at the end.

Case I1 can be handled by just copying the value of R to R^1 and case I2 can be handled with a right shift of R^1 and an AND operation with S_i such that $s_i=t_{j+1}$. So, to compute R_j^1 we need one additional shift (the shift of R is done already), one AND operation and one OR operation. An example (with the same pattern and text as the example for the exact matching, is given in figure 2.

Consider now allowing one deletion from the pattern (and no insertions). We will define R, R^1 (which now indicates one deletion), and S as before. There are again two cases for a match with at most one deletion of the first i characters of P up to t_{j+1}:

D1. There is an exact match of the first $i-1$ characters up to t_{j+1} (which is indicated by the new value of the R array $R_{j+1}[i-1]$). This case corresponds to deleting p_i and matching the first $i-1$ characters.

D2. There is a match of the first $i-1$ characters up to t_j with one deletion *and* $t_{j+1}=p_i$. In this case, the deletion is somewhere inside the pattern and not at the end.

Case D2 is handled as before (it is exactly the same), and case D1 is handled by a right shift of the new value of R_{j+1}.

Finally let's consider a substitution. That is, we allow replacing one character of P with one character of T. (We can achieve substitution with one deletion and one insertion, but in many cases we want substitution to count as only one error.) We again have two cases:

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* $t_{j+1}=p_i$. In this case, the substitution is somewhere inside the pattern and not at the end.

	a	a	b	a	a	c	a	a	b	a	c	a	b
a	1	1	0	1	1	0	1	1	0	1	0	1	0
a	0	1	0	0	1	0	0	1	0	0	0	0	0
b	0	0	1	0	0	0	0	0	1	0	0	0	0
a	0	0	0	1	0	0	0	0	0	1	0	0	0
c	0	0	0	0	0	0	0	0	0	0	1	0	0

R

	a	a	b	a	a	c	a	a	b	a	c	a	b
a	1	1	1	1	1	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	1	1	1	1	0	0	0
b	0	0	1	1	0	0	0	0	1	1	0	0	0
a	0	0	0	1	1	0	0	0	0	1	1	0	0
c	0	0	0	0	0	0	0	0	0	0	1	1	0

R^1

Figure 2: An example for approximate matching with one insertion.

Case S2 is again the same. Case S1 corresponds to looking at $R_j[i-1]$ as opposed to looking at $R_{j+1}[i-1]$ in case D1. Still very few operations cover one substitution as well.

We are now ready to consider the general case of up to k errors, where an error can be either an insertion, a deletion, or a substitution (the Levenshtein or the edit-distance measure). Overall, instead of one additional R^1 array, we will maintain k additional arrays $R^1, R^2, ..., R^k$, such that array R^d stores all possible matches with up to d errors. We need to determine the transition from array R_j^d to R_{j+1}^d. There are 4 possibilities for obtaining a match of the first i characters with $\leq d$ errors up to t_{j+1}:

1. There is a match of the first $i-1$ characters with $\leq d$ errors up to t_j and $t_{j+1} = p_i$. This case corresponds to matching t_{j+1}.
2. There is a match of the first $i-1$ characters with $\leq d-1$ errors up to t_j. This case corresponds to substituting t_{j+1}.
3. There is a match of the first $i-1$ characters with $\leq d-1$ errors up to t_{j+1}. This case corresponds to deleting p_i.
4. There is a match of the first i characters with $\leq d-1$ errors up to t_j. This case corresponds to inserting t_{j+1}.

Let's denote R as R^0, and assume that $t_{j+1} = s_c$. Overall, we have the following expression for R_{j+1}^d:

R_0^d = 11..100...000 d ones.

$$R_{j+1}^d = Rshift[R_j^d] \text{ AND } S_c \text{ OR } Rshift[R_j^{d-1}] \text{ OR } Rshift[R_{j+1}^{d-1}] \text{ OR } R_j^{d-1}$$

$$= Rshift[R_j^d] \text{ AND } S_c \text{ OR } Rshift[R_j^{d-1} \text{ OR } R_{j+1}^{d-1}] \text{ OR } R_j^{d-1}. \quad (2.1)$$

Overall, we have a total of two shifts, one AND, and three ORs for each R^d. There are $k+1$ arrays, so the total amount of work is $O((k+1)n)$. An important feature of this algorithm is that it can be relatively easily extended to several more complicated patterns. This is the topic of Section 3.

2.3. An Improvement to the Main Algorithm

If the number of errors is small compared to the size of the pattern, then we can improve the running time sometimes by what we call *the partition approach*. Suppose again that the pattern P is of size m and that at most k errors are allowed. Let $r = \lfloor \frac{m}{k+1} \rfloor$, and let $P_1, P_2, ..., P_{k+1}$ be the first $k+1$ blocks of P each of size r. In other words, $P_1 = p_1 p_2 \cdots p_r$, ..., $P_j = p_{(j-1)r+1} \cdots p_{jr}$. 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 scheme in 2.2) 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 all k of the R^d 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 number of such matches depend on many factors including the size of the alphabet, the actual text, and the values of r and m. For example, if $r = 1$, then we will need to check any time there is a character match, which is probably too often. On the other hand, if $r = 3$, $m = 12$ (which implies $k = 3$), the alphabet size is 26, and the text is uniformly random (i.e., each character appears with the same probability), the expected number of matches of any of the P_j's is about 0.02% of the time. In this case, it is obviously advantageous to search for exact matches and use the approximate scheme only at the rare occasions where a match occurs. The running time in this case is essentially the same as the running time of a search without errors. Experiments using this partition scheme for different alphabet sizes are given in Section 4.

The main advantage of this scheme is that the algorithm for exact matching presented in Section 2.1 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 (see Section 2.1). 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.

3. Extensions

An important feature of our algorithm is its flexibility. In addition to asking about a single string, the algorithm supports range of characters (e.g., ''0-9''), complements (e.g., everything except blank), arbitrary sets of characters (e.g., {a,e,i,o,u}), unlimited ''wild cards,'' and combinations of the above. Searching for several strings at the same time is also possible, although the size of the pattern becomes the sum of the sizes of the different strings (and might thus require more than one word to represent). The algorithm can be extended to support any regular expression. We discuss regular expressions briefly in section 3.8.

3.1. Sets of Characters

Replacing one character with a set of allowable characters is very easy to achieve with this algorithm (as was shown by Baeza-Yates and Gonnet [BG89]). Suppose that the pattern we want to find is P_1 followed by one digit followed by P_2 and that we allow up to k errors. We denote this pattern by $P_1[0–9]P_2$. The only thing we need to do to accept a set of characters is to include the position of [0–9] in the S arrays for all digits. That is, in the preprocessing stage, when we decide for each character the positions that this character matches in the pattern, we include all the characters in the set within that position. The rest of the algorithm is identical with the regular

algorithm. A complement of a character is a special case of a set of characters and it can obviously be handled in the same way.

3.2. Wild Cards

A single wild card is a symbol that matches all characters. As such, it is a special case of a set of characters and can be handled as we discussed in the previous section. Sometimes, however, we want to indicate that we allow an unbounded number of characters to appear in the middle of the pattern (or even do it several times in the middle of the pattern). This case requires modifying the algorithm slightly. Let the pattern be $P = p_1 p_2 \cdots p_m$, and assume that the positions of '#' (which indicates unlimited wild cards in agrep) are after the characters $p_{i_1}, p_{i_2}, ..., p_{i_s}$. (There is no reason to have two #'s in a row.) Let $S^{\#}$ be a bit array that has 1 in exactly the positions $i_1, i_2, ..., i_s$. The effect of putting a '#' following p_i can be defined as follows. If we are scanning t_j and we find a match with up to d errors that ends at p_i, then later when we scan t_r, for any $r > j$, we can start matching t_r to p_{i+1} no matter how many characters we skipped. In other words, if at some point there is a match up to p_i then this match is always valid later on (because all the characters later on can be considered as part of the '#').

We can adjust the algorithm for this case as follows. At each step, we apply the regular algorithm to compute all the R arrays. That is, we compute $R_j^0, R_j^1, ..., R_j^d$ using (2.1). Then, for each i, $1 \le i \le d$, we set $R_j^i = R_j^i$ *OR* $[R_{j-1}^i$ *AND* $S^{\#}]$. This step corresponds to the action ''if at any point, there is a 1 entry in R^i *AND* $S^{\#}$, then this entry should remain 1 from now on.''

3.3. Unknown Number of Errors

In some cases, we do not know the number of errors a-priori. We would like to find all occurrences of the pattern with the *minimal* number of errors possible. The algorithm can be extended to this case as follows. We first try to find the pattern with no errors. If we are unsuccessful, we try with one error, then with three errors, then with 7 errors, and so on, essentially doubling the number of errors (and adding one) at each attempt. If the number of errors turns out to be k, then the running time will be $O(1 \cdot n + 2 \cdot n + 4 \cdot n + \cdots + 2^b \cdot n)$, where 2^b is the first power of 2 greater than k. In the worst case, we perform 4 times as many operations as we would have had we known k (in most cases, the factor is actually 2 or 3). This is not desirable, but not prohibitive. There are other methods to find the minimum number of errors.

3.4. A Combination of Patterns With and Without Errors

Sometimes we do not want to allow parts of the pattern to have errors. For example, we may look for license plate ABC123, and we know that the letters are correct but the numbers may have one error in them. We denote this pattern by <ABC>123. We can modify the algorithm to shield parts of the pattern from having any errors in them. Let's assume that I is the set of indices in the pattern where no error is allowed, and let M be a masking array (of size m) that has a 0 in the

indices of I and a 1 otherwise. We would like to modify (2.1) such that insertions, deletions, and substitutions can only occur outside of I. This is done by masking these cases with M. The expression in (2.1) is changed to

$$R_{j+1}^{d} = \left(Rshift[R_j^d] \text{ AND } S_c \right) \text{ OR } \left(Rshift[R_j^{d-1} \text{ OR } R_{j+1}^{d-1}] \text{ OR } R_j^{d-1} \right) \text{ AND } M. \qquad (2.2)$$

3.5. Non-Uniform Costs

The edit distance measure, defined in section 1, assumes that insertions, deletions, and substitutions all have the same cost. But in some cases, we want to allow fewer deletions, say, than substitutions. The algorithm can be extended, albeit in a limited way, to the case where each operation has a different cost. We illustrate this extension with an example. Suppose that substitutions add 1 to the distance, but insertions and deletions add 3 each. Insertions and deletions are handled in cases 4 and 3 (see section 2). Insertions contribute the OR of R_j^{d-1} and deletions contribute the OR of $Rshift[R_{j+1}^{d-1}]$ (2.1). We would like them to cost 3 times as much. In other words, a deletion or insertion that leads to a match with d errors should come from a match with $d-3$ errors. This can be achieved by simply replacing the $d-1$ in both expressions with $d-3$. This modification is very simple and it does not add to the running time; however, it works only for small integer costs.

3.6. A Set of Patterns

If we have several patterns and we want to find all occurrences of any of them, then we can either search them one at a time or together. The advantage of searching for all of them together is that it can be done in one scan (and in one command). Suppose that we are looking for $P_1, P_2, ..., P_r$. We concatenate all the patterns and put them in one array (using as many words as needed), and apply the algorithm on that array with the following modifications. Let M be a bit array the size of the combined pattern, and let bit i be 1 if and only if i corresponds to the first character of any of the patterns. For each $s \in \Sigma$, we build two bit arrays. The first, S_s is identical with the one described in section 2. It is used to determine if a match occurs. The second array S'_s = S_s *AND* M. It indicates whether s is the first character of any pattern. If so, then we must start the match at that pattern: we do not want to depend on the end of the previous pattern. Thus, after we compute R_j, we OR it with S'_s (where $s = t_j$). We compute the rest of the R arrays as before, except that in each step we OR them to a special mask that sets the first d bits in R^d of each separate pattern to 1; this allows d initial errors in each pattern. (This is not the most efficient way to solve this problem, but it's reasonably simple.) This case is a special case of patterns as regular expressions, which we will discuss shortly.

3.7. Long Patterns

Suppose that the pattern occupies several words and that it is a simple string. The algorithm proceeds in the same fashion by computing the R^d arrays for all words. However, unless the number of errors is large, the first part of the pattern will not match the text quite often. If there is no match with k errors starting after position r of the pattern, then there is no need to maintain the R arrays corresponding to positions larger than r (their values will be 0). Thus, most of the time there will be no need to maintain the R^d arrays for the right side of the pattern. We only need to be alerted when the last bit of the last R^d array that we maintain gets the value of 1. In that case, we start maintaining the R_d arrays for the next part of the pattern. This improvement works only for simple strings and not for sets of strings or regular expressions.

3.8. Regular Expressions

The algorithm can be extended to allow any regular expression as a pattern. We describe the method here only briefly. Algorithms for matching regular expressions with errors, based on the dynamic-programming approach, appear in [MM89]. First, we illustrate the algorithm with a simple example. We do not try to optimize the algorithm at this stage; we try to make it as simple as possible (a more detailed discussion and more efficient algorithms will be presented elsewhere). Let the pattern be $P = ab(cd|e)^* fg$ (i.e., starting with ab and ending with fg with any number of either cd or e in between). This regular expression is translated to the non-deterministic finite automata shown in Fig. 3 (for more on such translations see [HU79]). We now assign a bit array to represent the automata. We number the states including the null states that do not correspond to any character (see Fig. 3). This ‘‘linearizes’’ the automata. Each state corresponds to one entry in the array. Thus, for P we have an array of size 11. Notice that all the non-ε moves go to the next state and thus can be handled by essentially a shift and an AND operation. We need to find a way

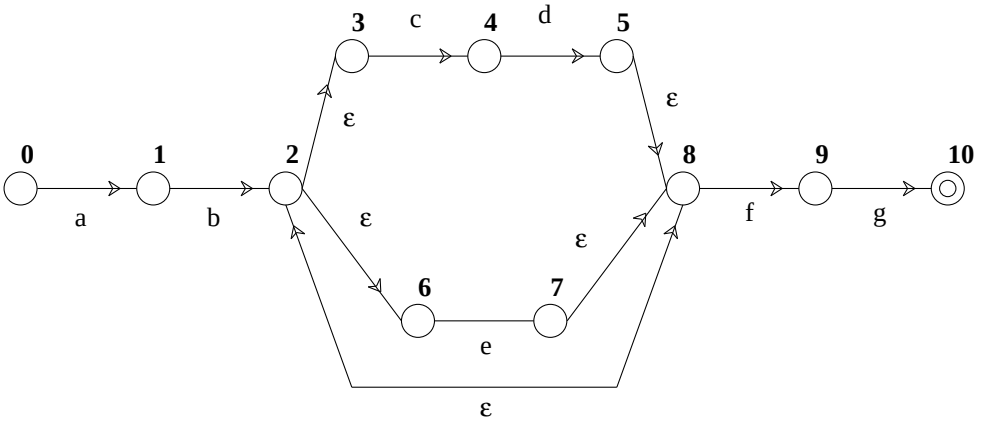

Figure 3: The non-deterministic automata corresponding to $ab(cd|e)^*fg$.

to deal with arbitrary jumps required by the ε moves (e.g., from state 2 to state 8) and with ‘‘non-jumps’’ that happen to be in consecutive states (e.g., from state 5 to state 6). The non-jumps can be handled easily with a mask. The arbitrary jumps are harder to handle. The meaning of an ε move from state *i* to state *j* is that if, at any point, we match up to state *i* then the same match holds also up to state *j*. In other words, if there is a 1 corresponding to state *i* in the array, then the ε move from *i* to *j* implies that there should be a 1 corresponding to state *j*. The main observation is that a given bit array and set of ε moves completely determine the value of the bit array after the ε moves are taken. Thus, the set of ε moves defines a function that maps a bit array to another. We need to be able to implement this function efficiently.

Let *f* denote the function that maps one bit array to another by applying all the ε moves. We divide the bit array into bytes, i.e., groups of 8 bits each. Consider the first 8 bits of the bit array. The values of these bits determine which 1’s should be set when we apply the ε moves on states 1 to 8. Since there are only 256 ($=2^8$) possible values for 8 bits, we can preprocess all possibilities and construct a table of size 256 which will hold, for each possible byte, the whole bit array with 1’s only in places corresponding to the ε moves. (We need the whole array and not just the first 8 bits, because there might be forward jumps.) We can do that for each byte. Given now a current value of *R*, we first apply the regular algorithm, taking care of regular non ε moves, then we divide the array into bytes, find the corresponding arrays in the tables (we have one table per byte), and OR all of them to *R*. This implements all the jumps, and if the pattern is occupies no more than 32 bits (as is often the case), only 4 more steps are required. (If the text is large, it is worthwhile to preprocess 16 bits for a table of size 65536 and half as many steps.)

The algorithm sketched above is not the most efficient algorithm, but it is reasonably simple. More efficient algorithms will be described elsewhere.

3.9. Very Large Alphabets

Sometimes the alphabet is very large. For example, the pattern can be a segment of a program which we want to find inside a large program. Instead of counting each text character as a character in the pattern, we may wish to count each line as a character (this is done, for example, in the program *diff* which is used to compare files). The problem we have with large alphabets is that the preprocessing stage, where all the S_s arrays are constructed, will take too long and require too much space (the set of all possible program lines is, for all practical purposes, infinite). However, we can use hashing to map the alphabet to a reasonable size. We first decide the hashing table size, which should be large enough to avoid many collisions but not too large so that we can afford the space (e.g., 1024). The alphabet becomes the set of integers from 1 to the table size. The algorithm is applied to the hash values. At the end, all matches should be checked again, however, to remove matchings that result from collisions. We do not support, at this time, large alphabets in *agrep*.

4. Experimental Results

We have implemented the algorithm and many of its extensions and tested them against previous algorithms. All tests were run on a SUN SparcStation II running UNIX. A description of *agrep* — a tool for approximate string matching based on this algorithm, which we used for the experiments — is given in the next section. In almost all the cases, our algorithm beat the other algorithms, sometimes by a wide margin.

The numbers given here should be taken with caution. Any such results depend on the architecture, the operating system, and the compilers used. We probably put more efforts into optimizing our algorithm than we did for other algorithms (although we put significant effort into that too), and we did not test all possible scenarios. However, we tried not only to be fair but also to be conservative. We used the final agrep program for all our tests even though it contains many options that slow it down and are not available in the other programs (e.g., the fact that agrep is record oriented — see next section — slows it down considerably). For the simple cases that are listed in the tables below, we sometimes obtained 20-30% faster running times with versions of the program that has only the tested options. We believe that the main strength of this algorithm is that it is more flexible, general, and convenient than all previous algorithms. The fact that for most of the common applications of it, agrep is also significantly faster than other algorithms is nice, but speed is mostly a secondary issue here; other algorithms are reasonably fast already.

First, we tested searching without errors vs. the grep family of programs available in UNIX and against an implementation of the Boyer-Moore algorithm. The text was a bibliography file of size one Megabytes. We used 5 patterns of varying sizes (4-10) and averaged the results. Agrep consistently beats the grep family, but it is not as fast as the Boyer-Moore algorithm. (The Boyer-Moore algorithm cannot be used, as far as we know, for most of the extensions in the previous section; even finding line numbers for all the matches is not trivial and slows the algorithm down considerably.)

We then tested the approximate string-matching algorithm against two other algorithms, one by Ukkonen [Uk85a] (which we implemented) and one by Galil and Park [GP90] (labeled MN2; the program was provided to us by W. I. Chang) which is based on another technique by Ukkonen [Uk85b]. The last algorithm was found by Chang and Lawler [CL90] to be the fastest among the algorithms they tested. We used random text (of size 1,000,000) and pattern (of size 20), and two different alphabet sizes. In this case, since we use the idea of partitioning the pattern, the size of the alphabet makes a big difference. A large alphabet leads to very few accidental exact matches,

BM	agrep	egrep	grep	fgrep	wc
0.21	0.35	0.79	1.22	1.61	1.19

Table 1: Exact matching of simple strings.

thus the running time is essentially the same as the one for exact matching. A small alphabet leads to many matches and the algorithm's performance degrades. The case of binary alphabet serves as the worst case for this purpose. Results are shown in Table 2. The final test was for more complicated patterns, including some of the extensions discussed in the previous section. (Anything within the <> brackets must match exactly; a '#' stands for a wild card of arbitrary length; A ';' serves as the Boolean AND operation, namely all patterns must appear within the same line; a '|' is the regular expression union operation; and a '*' is the Kleene closure.) The results are given in Table 3 (the file was the same bibliographic file used in Table 1). 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 high for small errors).

	agrep		MN2		Uk85a	
	$\Sigma = 2$	$\Sigma = 30$	$\Sigma = 2$	$\Sigma = 30$	$\Sigma = 2$	$\Sigma = 30$
k = 0	0.35	0.35	1.21	0.98	2.36	0.90
k = 1	0.52	0.38	3.03	2.42	5.01	2.06
k = 2	1.78	0.38	4.94	3.87	7.93	3.19
k = 3	2.56	0.39	6.68	5.33	11.80	4.38
k = 4	3.83	0.41	8.72	6.89	13.40	5.55
k = 5	4.42	0.42	10.41	8.28	15.45	6.77
k = 6	5.13	0.73	11.83	9.75	17.07	7.99

Table 2: Approximate string matching of simple strings.

pattern	k = 0	k = 1	k = 2	k = 3
Homogenious	0.35	0.39	0.41	0.47
<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 3: Approximate matching of complicated patterns.

5. A Description of agrep

agrep is used similarly to egrep and it supports most of the features of egrep (although it is not 100% compatible) and many additional features. One notable difference between agrep and the grep family (besides the additional features) is that agrep is *record oriented* (rather than line oriented). A record is defined by the user (with the default being a line). Agrep outputs all records that match the query (see also the -d option described below). Agrep is available by anonymous ftp from cs.arizona.edu (IP number 192.12.69.5).

agrep *pattern* file-name — finds all occurrences of *pattern* (that is, output all records containing *pattern*) in the text file file-name without errors. The pattern can be a string or an arbitrary regular expression. We describe below the unusual features of agrep that are not found in similar programs. A complete description is given in the manual pages distributed with agrep.

-*k* finds all occurrences with at most *k* errors (insertions, deletions, or substitutions), where *k* is a positive integer.

-D*c* each deletion counts as *c* errors; *c* must be a non-negative integer; the default value of *c* is 1

-I*c* each insertion counts as *c* errors; *c* must be a non-negative integer; the default value of *c* is 1

-S*c* each substitution counts as *c* errors; *c* must be a non-negative integer; the default value of *c* is 1

-d '**delim**'

delim is a user-defined symbol (or string) for record delimiter (the default is the new-line symbol). This enables searching paragraphs (in which case **delim** = 2 new lines in a row) or mail messages (**delim** = '^From '). This feature makes agrep a record-oriented program rather than just a line-oriented program. We believe that it will be very useful.

Examples

agrep -3 -D2

finds all occurrences with at most 3 errors where a deletion counts as 2 errors and each insertion or substitution counts as one error.

agrep -4 -I5

finds all occurrences with at most 4 errors but no insertions allowed (because their cost is prohibited).

agrep -d '^From ' 'breakdown; (inter|arpa|bit)net'

outputs all mail messages (the pattern '^From ' separates mail messages in a mail file) that contain breakdown and one of either internet, arpanet, or bitnet.

agrep -d '$$' -1 '<word1> <word2>'

finds all paragraphs that contain word1 followed by word2 with one error in place of the blank between the words (the <> indicate that no error is allowed inside; see section 3.4). In particular, if word1 is the last word in a line and word2 is the first word in the next line, then

the space will be substituted by a newline symbol and it will match. Thus, this is a way to overcome separation by a newline. Note that -d '' (or another delim that spans more than one line) is necessary, because otherwise agrep searches only one line at a time.

6. Conclusions

Searching text in the presence of errors is commonly done 'by hand' — one tries all possibilities. This is frustrating, slow, and with no guarantee of success. The new algorithm presented in this paper for searching with errors can alleviate this problem and make searching in general more robust. It also makes searching more convenient by not having to spell everything precisely. The algorithm is very fast and general and it should find numerous applications.

There is one important area of searching with errors that we did not address — searching an indexed file. Throughout the paper we assumed that the files are not indexed (preprocessed) in any way, thus a sequential scan through them is necessary. We believe that the problem of finding good indexing schemes that allow approximate search is the main open problem is this area. Unfortunately, we do not know of any satisfactory solution at this point. However, with the speed of current computers, scanning large files (up to tens of megabytes) can be done reasonably fast. One can argue that the size of our data increases as our speed of processing it increases. This is certainly true for some applications, but not for all. Many applications have an upper bound on size and sequential search for those applications will be realistic.

Acknowledgements:

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

References

[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,''

FOCS 90, pp. 116–124.

[GG88]

Galil Z., and R. Giancarlo, ‘‘Data structures and algorithms for approximate string matching,’’ *Journal of Complexity*, **4** (1988), pp. 33–72.

[GP90]

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

[GB91]

Gonnet, G. H. and R. A. Baeza-Yates, *Handbook of Algorithms and Data Structures*, Second Edition, Addison-Wesley, Reading, MA, 1991.

[HD80]

Hall, P. A. V., and G. R. Dowling, ‘‘Approximate string matching,’’ *Computing Surveys*, (December 1980), pp. 381–402.

[HU79]

Hopcroft, J.E., and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass (1979).

[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.

[LV88]

Landau G. M., and U. Vishkin, ‘‘Fast string matching with k differences,’’ *Journal of Computer and System Sciences*, **37** (1988), pp. 63–78.

[LV89]

Landau G. M., and U. Vishkin, ‘‘Fast parallel and serial approximate string matching,’’ *Journal of Algorithms*, **10** (1989).

[Le66]

Levenshtein, V. I., ‘‘Binary codes capable of correcting deletions, insertions, and reversals,’’ *Sov. Phys. Dokl.*, (February 1966), pp. 707–710.

[My86]

Myers, E. W., ‘‘An O(ND) difference algorithm and its variations,’’ *Algorithmica*, **1** (1986), pp. 251–266.

[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)

[Uk85a]

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

[Uk85b]

Ukkonen E., ‘‘Algorithms for approximate string matching,’’ *Information and Control*, **64**, (1985), pp. 100–118.