Fast and Practical Approximate String Matching

Ricardo A. Baeza-Yates and Chris H. Perleberg

Depto. de Ciencias de la Computación
Universidad de Chile
Casilla 2777, Santiago, Chile [*]

Abstract

We present new algorithms for approximate string matching based in simple, but efficient, ideas. First, we present an algorithm for string matching with mismatches based in arithmetical operations that runs in linear worst case time for most practical cases. This is a new approach to string searching. Second, we present an algorithm for string matching with errors based on partitioning the pattern that requires linear expected time for typical inputs.

1 Introduction

Approximate string matching is one of the main problems in combinatorial pattern matching. Recently, several new approaches emphasizing the expected search time and practicality have appeared [3, 4, 27, 32, 31, 17], in contrast to older results, most of them are only of theoretical interest. Here, we continue this trend, by presenting two new simple and efficient algorithms for approximate string matching.

First, we present an algorithm for string matching with k mismatches. This problem consists of finding all instances of a pattern string $P = p_1p_2p_3....p_m$ in a text string $T = t_1t_2t_3....t_n$ such that there are at most k mismatches (characters that are not the same) for each instance of P in T. When $k = 0$ (no mismatches) we have the simple string matching problem, solvable in $O(n)$ time.

Various algorithms have been developed to solve the problem of string matching with k mismatches. Running times have ranged from $O(mn)$ for the brute force algorithm to $O(kn)$ [19, 12] or $O(n \log m)$ [4, 15]. In this paper we present a simple algorithm (one page of C code) that runs in $O(n)$ time, worst case, if all the characters p_i in P are distinct (i.e. none of the characters in P are identical) and in $O(n + R)$ worst case time if there are identical characters in P, where R is the total number of ordered pairs of positions at which P and T match. Note that $0 \leq R \leq f_{max}n$, where f_{max} is the frequency of the most commonly occurring character of P. The space used is $O(2m + |\Sigma|)$ where Σ is the underlying alphabet (typically ASCII symbols). The only algorithm which is similar to ours is a $O(n + R \log\log m)$ result for insertions and deletions, reported in [21]. Related ideas can be found in [11, 1].

String matching with errors consists of finding all substrings of the text that have at most k errors with the pattern, where the errors are counted as the minimal number of insertions, deletions, or substitutions of characters needed to convert one string to another. The best worst case running time is $O(kn)$ [20, 13, 29]. Recently, practical algorithms have been proposed [27, 32, 31], the latter

[*]This work was partially supported by Grant 1950622 from Fondecyt. E-mail: rbaeza@dcc.uchile.cl.

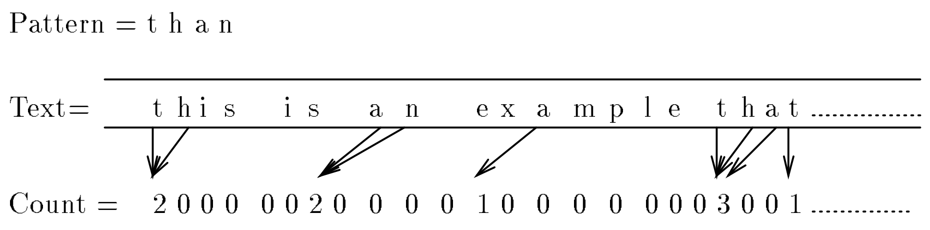
Pattern = t h a n
Text= t h i s i s a n e x a m p l e t h a t ....
Count = 2 0 0 0 0 0 2 0 0 0 0 1 0 0 0 0 0 0 0 3 0 0 1 ....

Figure 1: Searching example for the counting approach.

of them called "agrep" which is several times faster than "grep" (the well known Unix file searching utility). We present a new algorithm based on partitioning the pattern, which achieves a $O(n)$ expected search time for $k \leq O(m/\log m)$ using $O(m^2)$ extra space. This result generalizes to any algorithm based in partitioning the pattern, as in [32], and is simpler than Chang and Lawler's algorithm [9].

For a complete set of references on these problems we refer the reader to [14]. A preliminary version of this paper was presented in [7].

2 String Matching with Mismatches

First let us describe the string matching with k mismatches algorithm using the best (and practical) case of only distinct characters in P. In this case, each character in P has a unique offset from the beginning of P, and every time a character of P is encountered in T, the position of the beginning of this instance of P is uniquely determined. An array of the same size as the alphabet of characters can be used to indicate the offset of each character from the beginning of P. If a character is not in P, a special flag value can replace the offset.

Suppose there is a counter c_i for every t_i. The counters are initialized to zero, and as each character of T is read and examined, its offset (if it is a character in P) is used to find the counter at the beginning of this instance of P, and this counter is incremented. If there is an instance of P with $k = 0$, an exact match, the counter at the position of this instance of P will be incremented m times. The number of mismatches is equal to m minus the value of the counter, and if it is less than or equal to k, a match is reported. Only m counters are necessary at any one time, so the counters can be implemented with an array of size m that is traversed in a circular manner. This idea is shown in Figure 1.

The case of non-distinct characters in P can be handled by converting the array of offsets into an array of linked lists of offsets. If a character p_i occurs two times in P, it has two distinct offsets from the beginning of P, and these two offsets can be obtained by indexing into the array using the character p_i and traversing the linked list which contains the two offsets in two nodes. The two offsets can then be used to increment two distinct counters.

The C code for the algorithm is given in Figure 2. Several optimizations are included that deviate from the simple description of the algorithm given above. The array `count` that contains the m counters has 256 entries so it can be accessed in a circular manner using the fast AND (&) operator and not the slow MOD (%) operator. Counters are maintained at the position of the last character (not at the position of first character) of each possible instance of P. Each counter is initialized with the value m, and the counters are decremented with each occurrence of a match, so the final value of each counter is the number of mismatches. To handle the special case of the first $m-1$ characters in T, the first $m-1$ counters are initialized to high values so they can never reach a valid value of mismatches. Finally, it is worth noting that if all the characters of the pattern are distinct, the innermost "for" loop can be removed from the subroutine `search`.

Now we discuss the running time. Consider first the simple case of only distinct characters in P. In this case, for each character in T, the array of offsets is accessed, and a counter is possibly updated. This case has a running time of $O(n + R)$, where R is the number of ordered pairs of positions where characters of P and T match. Note that $0 \leq R \leq n$ for this case, having $O(n)$ worst case. Space used is $O(m + |\Sigma|)$ which includes the array of offsets which has an entry for every possible character of the alphabet Σ, and the array of m counters. Running time for preprocessing is $O(2m + |\Sigma|)$ as each entry of the array of offsets (size $|\Sigma|$) is initialized (this can be avoided, see [16]), m entries for the m characters of P are written into the array, and the m counters are initialized.

The space used for the case of non-distinct characters in P is $O(2m + |\Sigma|)$ (we use up to $m - 1$ more linked list nodes than in the simple case), and the running time for preprocessing is of the same order as described above. Worst case running time is $O(n + R)$. For this case, we have $0 \leq R \leq f_{max} n$ where f_{max} is the frequency of the most commonly occurring character of P. Every time this character appears in T, a linked list of size f_{max} is traversed, and f_{max} counters are updated. In the absolute worst situation, P consists of m occurrences of a single distinct character, and T consists of n instances of this character, giving a running time of $O(mn)$. However, this is neither a common nor useful situation. In the average case, for an alphabet in which each character is equally likely to occur, the probability of two symbols being equal is $1/|\Sigma|$. Then, if a pattern has m/d different symbols, each one appears on average d times. With probability $1/|\Sigma|$ for each different character, the algorithm performs d operations. Given that there are m/d different characters in the pattern, the expected value of R is $d/|\Sigma| \times m/d = m/|\Sigma|$. Therefore, the expected running time is

$$O\left(\left(1 + \frac{m}{|\Sigma|}\right) n\right)$$

Note that the average case running time is independent of the number of distinct characters in P. In practice with ASCII characters, $m < |\Sigma|$, which gives a linear expected time. It is worth mentioning that in DNA applications, where often $m >> |\Sigma|$, the algorithm approaches its worst case running time of $O(mn)$.

Because the algorithm is independent of k, it can be easily adapted to find the "best match" (smallest error). Many times this is desirable when a bound on k is not known *a priori*.

3 String Matching with Errors

In this section we describe a new algorithm which is based in two classical algorithms and the partition approach mentioned in [32]. We show that this very simple algorithm has linear expected running time for most values of k. In [22] is presented an algorithm that achieves expected linear time when one of the strings is random.

The partition approach is based on the following fact: an occurrence with at most k errors of a pattern of length m implies that at least one substring of length r in the pattern matches a substring of the text occurrence exactly, where

$$r = \left\lfloor \frac{m}{k+1} \right\rfloor .$$

There are many ways to use this idea. Perhaps the simplest one, used in [32], is to search for the first $k + 1$ consecutive blocks of size r of the pattern P. If any of the blocks is an exact match, we try to extend the match, checking if there are at most k errors. This idea was used in conjunction

```
#define SIZE    256         /* size of alpha index and count array */
#define MOD256  0xff        /* for the mod operation */
typedef struct idxnode {    /* structure for index of alphabet */
    int offset;             /* distance of char from start of pattern */
    struct idxnode *next;   /* pointer to next idxnode if it exists */
} anode;
anode alpha[SIZE];  /* offset for each alphabetic character */
int   count[SIZE];  /* count of the characters that don't match */

int search(t,n,m,k,alpha,count)     /* string searching with mismatches */
char *t;        /* text */
int   n,m;      /* number of characters in text and pattern */
int   k;        /* the Hamming Distance */
anode alpha[];  /* index of alphabet */
int   count[];  /* circular buffer of count of mismatched characters */
{
    int i, off1;    anode *aptr;

    for (i=0; i<n; i++) {
        if ((off1=(aptr=&alpha[*t++])->offset) >= 0) {
            count[(i+off1)&MOD256]--;
            for (aptr=aptr->next; aptr!=NULL; aptr=aptr->next)
                count[(i+aptr->offset)&MOD256]--;
        }
        if (count[i&MOD256] <= k)
            printf("Match in position %d with %d mismatches\n"
                                                ,i-m+1,count[i&MOD256]);
        count[i&MOD256] = m;
    }
}
int preprocess(p,m,alpha,count)          /* preprocessing routine */
char *p;        /* pointer to pattern */
int   m;        /* number of characters in pattern */
anode alpha[];  /* alphabetical index giving offsets */
int   count[];  /* circular buffer for counts of mismatches */
{
    int    i,j;     anode *aptr;

    for (i=0; i<SIZE; i++) {
        alpha[i].offset = -1;  alpha[i].next   = NULL;  count[i] = m;
    }
    for (i=0,j=128; i<m; i++,p++) {
        count[i] = SIZE;
        if (alpha[*p].offset == -1) alpha[*p].offset = m-i-1;
        else {
            aptr = alpha[*p].next;           alpha[*p].next = &alpha[j++];
            alpha[*p].next->offset = m-i-1;  alpha[*p].next->next = aptr;
        }
    }
    count[m-1] = m;
}
```

Figure 2: C code for string matching with mismatches

with the extension of the shift-or algorithm [3] to string matching with errors. Here we combine the idea with traditional multiple string searching algorithms.

The simplest algorithm is to build an Aho-Corasick machine [2] (the extension of the Knuth-Morris-Pratt algorithm [18] to search for multiple patterns) for the $k+1$ blocks of length r (less blocks if some of them are equal). For every match found, we extend the match, checking if there are at most k errors, by using the standard dynamic programming algorithm to check the edit distance between two strings. We can decompose the running time in the search phase and the checking phase. The search phase requires linear worst case time for finite alphabets. The checking phase will depend on the number of potential matches found. Assuming that both the pattern and the text are random, the expected number of occurrences of any of $k+1$ patterns of length r in a text of length n is

$$\frac{(k+1)n}{|\Sigma|^r}$$

Because we may have the block in more than one position of the pattern (the case of repeated blocks), instead of storing all the possible positions and checking each case, it is much easier to check the whole pattern in the neighborhood of the block occurrence. Thus, we search for the pattern between positions $i-(m-r)-k$ and $i+m+k$ if the block was found in the i-th position of the text. Then, in the worst case, for every match checked we need $m(2(m+k)-r)$ operations by using dynamic programming. Although in average less operations are needed, this worst case bound is sufficient to prove expected time linearity. Thus, to achieve expected linear time, we should have

$$\frac{m(2(m+k)-r)(k+1)n}{|\Sigma|^r} \leq cn$$

for some fixed constant c. To solve for r in this transcendental equation, we use the following weaker inequality (replacing k by $m-1$):

$$\frac{(4m-3)m^2 n}{|\Sigma|^r} \leq cn$$

Solving for r, and then using $k \leq m/r - 1$, we have

$$k \leq \frac{m}{\log_{|\Sigma|}(m^2(4m-3)/c)} - 1$$

which shows that the maximum value for k is $O(m/\log m)$. Note that this bound is pessimistic, as we are using the worst case number of comparisons when checking. Also note that c can be used as parameter to tune the maximum value of k. This bound will be less restrictive if we use better algorithms in the checking phase. That is, this expected linearity result is valid for other algorithms, by example for [3]. Just as an example, for $c = 1.5$, $m = 10$ and $|\Sigma| = 32$, we have $k \leq 4$. Note that our technique works for reasonable large alphabets ($|\Sigma| \geq 32$). This bound can be relaxed if we use $c = f(k)$ such that $f(k) = o(k)$; for example $c = \log(k)$ or $c = \sqrt{k}$. For these cases the expected time is better than the best worst case known of $O(kn)$.

Figure 3 gives experimental results for the following algorithms in addition to ours: Ukkonen's [28], Chang's [8], Wu and Manber's [32] and Wu, Manber and Myers's [33]. Our algorithm uses a multiple string searching algorithm based in the Boyer-Moore-Sunday [26] string searching algorithm and Ukkonen's [28] dynamic programming algorithm for the checking phase. We used a random text of size 1Mb with an alphabet of size 32. The values are the average of ten searches with random patterns of size $m = 9$ and $m = 31$ and all possible values of k. We can observe that

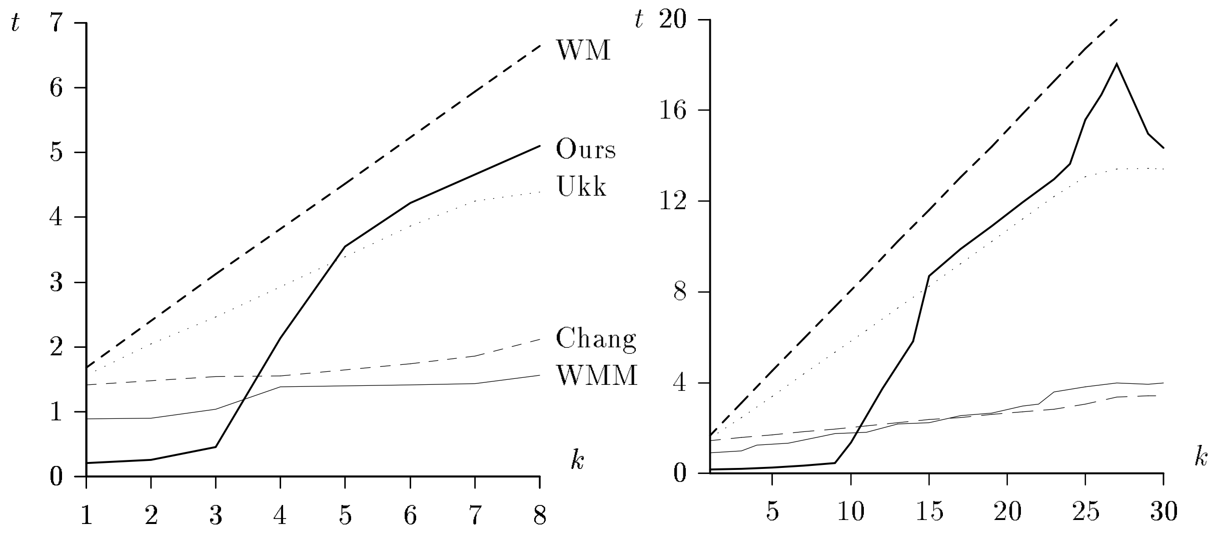

Figure 3: Times in seconds for $m = 9$ and $k = 0..9$ (left) and $m = 31$ and $k = 0..30$ (right).

our algorithm is the fastest for small k (up to 30% of the pattern length). For larger k, Chang's and Wu, Manber and Myers's algorithms are the fastest.

We can improve the searching phase by using multiple string searching algorithms based on the Boyer-Moore algorithm [10, 25, 6] or the shift-or algorithm [3, 31]. This improvement is significant when the number of blocks found is small (or in other words, when the alphabet is large).

To improve the check phase, we need to decrease the number of potential matches. Two possible solutions are:

- Select the blocks to be searched (the possible number of distinct sets of blocks depends on $m - (k + 1)r$) according to character frequency in the text. Another possibility is to use the frequency of small sequences of characters [30].
- Search more than $k + 1$ blocks, say $2(k + 1)$ blocks. This forces to have at least two exact matches in a text window of size $m + k$, which decreases the number of potential matches. On the other hand, it may increase the search time phase, but the optimal number of blocks will depend on the input.

Also, the same algorithm can be used for string matching with mismatches, or other error measures that obey the partition scheme used. The partition idea may be also useful to solve this problem allowing preprocessing (that is, when some kind of index of the text is available).

4 Final Remarks

Our first algorithm, based on counting, achieves $O(n)$ worst case time independent of k and without restrictions on m (which [3, 32] have), when all or almost all the symbols in the pattern are different. This seems to show a difference between the bit-parallelism of the shift-or algorithm [3] which is bounded by the word size, and counting, which seems to be bounded by symbol frequency.

This algorithm, in the vein of [3, 32, 31], shows the potential of logical and arithmetical operations over text comparisons. This indicates that there might be new ways of improving algorithms for similar problems by changing the computation model. In fact, this idea has been used for simple and fast algorithms to find the longest common substring between a text and a pattern[23], and variations of string matching with errors.

The second algorithm achieves linear expected time in most cases, and the bound on k is similar to the algorithms of Chang *et al.* [9]. One way to improve it would be to have a good algorithm to search patterns with at most 1 or 2 errors. This could be used to search for blocks of length $2r$ or $3r$, decreasing the number of potential matches. This idea has been used in [5]. Our algorithm shows that reducing a text searching problem to simpler problems to find potential answers may lead to simpler and fast algorithms, if the number of potential matches to check is small. This approach has also been used for two dimensional text searching [6]. This problem reduction should satisfy:

- The reduction itself and solving the new problem is faster than the original problem, and if possible, with running time independent of the number of potential matches.
- The number of potential matches is small, where "small" may mean on average or related to the real number of matches.

Finally, as in many text searching problem, there is no algorithm that is the best for all searching problems. That will depend on the application or the input. This suggests the use of hybrid algorithms as in "agrep" or adaptive algorithms as in [24] for string searching.

Acknowledgements

We thank Gonzalo Navarro for the experimental results and the refeeres for their suggestions which improved the paper.

References

[1] K. Abrahamson. Generalized string matching. *SIAM J on Computing*, 16:1039–1051, 1987.

[2] A.V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *C.ACM*, 18(6):333–340, June 1975.

[3] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35:74–82, Oct 1992.

[4] R. Baeza-Yates and G.H. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994.

[5] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Combinatorial Pattern Matching (CPM'96)*, Irvine, CA, Jun 1996.

[6] R. Baeza-Yates and M. Régnier. Fast two dimensional pattern matching. *Information Processing Letters*, 45:51–57, 1993.

[7] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate pattern matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching*, Lecture Notes in Computer Science 644, pages 185–192, Tucson, AZ, April/May 1992. Springer Verlag.

[8] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. of CPM'92*, pages 172–181. Springer-Verlag, 1992. LNCS 644.

[9] W. Chang and E. Lawler. Approximated string matching in sublinear expected time. In *Proc. 31st FOCS*, pages 116–124, St. Louis, MO, Oct 1990. IEEE.

[10] B. Commentz-Walter. A string matching algorithm fast on the average. In *ICALP*, volume 6 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 1979.

[11] M. Fischer and M. Paterson. String matching and other products. In R. Karp, editor, *Complexity of Computation (SIAM-AMS Proceedings 7)*, volume 7, pages 113–125. American Mathematical Society, Providence, RI, 1974.

[12] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17:52–54, 1986.

[13] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.

[14] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C.* Addison-Wesley, Wokingham, UK, 1991. (second edition).

[15] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Inf. Proc. Letters*, 33(3):113–120, November 1989.

[16] T. Hagerup. On saving space in parallel computation. *Inf. Proc. Letters*, 29:327–329, 1988.

[17] A. Hume and D.M. Sunday. Fast string searching. *Software - Practice and Experience*, 21(11):1221–1248, Nov 1991.

[18] D.E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J on Computing*, 6:323–350, 1977.

[19] G. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[20] G. Landau and U. Vishkin. Fast string matching with k differences. *JCSS*, 37:63–78, 1988.

[21] U. Manber and S. Wu. An algorithm for approximate string matching with non uniform costs. Technical Report TR-89-19, Department of Computer Science, University of Arizona, Tucson, Arizona, Sept 1989.

[22] E.W. Myers. An $O(ND)$ difference algorithm and its variants. *Algorithmica*, 1(2):251–266, 1986.

[23] C. Perleberg. Three longest substring algorithms. *J. of the Brazilian Computer Society*, 1(3):12–22, Apr 1995.

[24] P.D. Smith. Experiments with a very fast substring search algorithm. *Software - Practice and Experience*, 21(10):1065–1074, Oct 1991.

[25] M.A. Sridhar. Efficient algorithms for multiple pattern matching. Technical Report Computer Sciences 661, University of Wisconsin-Madison, 1986.

[26] D.M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, Aug 1990.

[27] J. Tarhio and E. Ukkonen. Boyer-moore approach to approximate string matching. In J.R. Gilbert and R.G. Karlsson, editors, *2nd Scandinavian Workshop in Algorithmic Theory, SWAT'90*, Lecture Notes in Computer Science 447, pages 348–359, Bergen, Norway, July 1990. Springer-Verlag.

[28] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[29] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10:353–364, 1993.

[30] S. Wu. Personal communication. 1992.

[31] S. Wu and U. Manber. Agrep - a fast approximate pattern-matching tool. In *Proceedings of USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, Jan 1992.

[32] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35:83–91, Oct 1992.

[33] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.