Very Fast and Simple Approximate String Matching

Gonzalo Navarro Ricardo Baeza-Yates

Department of Computer Science
University of Chile
Blanco Encalada 2120 - Santiago - Chile
{gnavarro,rbaeza}@dcc.uchile.cl

Abstract

We improve the fastest known algorithm for approximate string matching. This algorithm can only be used for low error levels. By using a new algorithm to verify potential matches and a new optimization technique for biased texts (such as English), the algorithm becomes the fastest one for medium error levels too. This includes most of the interesting cases in this area.

Key words: Approximate Matching, Text Searching, Information Retrieval.

1 Introduction

Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc.

The problem can be formally stated as follows: given a (long) text of length n, a (short) pattern of length m, and a maximal number of errors allowed k, find all text positions that match the pattern with up to k errors. The allowed errors are character insertions, deletions and substitutions. Text and pattern are sequences of characters from an alphabet of size σ. We call $\alpha = k/m$ the *error ratio* or *error level*.

In this work we focus on on-line algorithms, where the classical solution, involving dynamic programming, is $O(mn)$ time [8]. In the last years many

⋆ This work has been supported in part by FONDECYT grant 1950622.

algorithms have been presented that improve the classical algorithm, for instance [12,5,4,11,3,14,15,6,2].

For low error levels, the fastest known algorithm is [3]. It is a filtration algorithm, i.e. it quickly discards large text areas that cannot contain a match and later verifies suspicious areas using a classical algorithm. As all filtration algorithms, it ceases to work well for moderate error levels. In this paper we use a new technique to verify potential matches developed in [7]. We also improve the filter on biased texts such as natural language. The result is the fastest algorithm for approximate string matching for all but high error levels.

2 The Original Algorithm

The original idea of this algorithm was first presented in [14]:

Lemma: If a pattern is partitioned in $k+1$ pieces, then at least one of the pieces can be found with no errors in any approximate occurrence of the pattern.

This property is easily verified by considering that k errors cannot alter all the $k+1$ pieces of the pattern, and therefore at least one of the pieces must appear unaltered.

This reduces the problem of approximate string searching to a problem of multipattern exact search plus verification of potential matches. That is, we split the pattern in $k+1$ pieces and search all them in parallel with a multipattern exact search algorithm. Each time we find a piece in the text, we verify a neighborhood to determine if the complete pattern appears.

In the original proposal [14], a variant of the Shift-Or algorithm was used for multipattern search. To search r patterns of length m', this algorithm is $O(rm'n/w)$. Since in this case $r = k+1$ and $m' = \lfloor m/(k+1) \rfloor$, the search cost is $O(mn/w)$, which is the same cost for exact searching using Shift-Or. Later, in [3], the use of a multipattern extension of an algorithm of the Boyer-Moore (BM) family was proposed.

In [1,2] we tested an extension of the BM-Sunday algorithm [10]: we split the pattern in pieces of length $\lfloor m/(k+1) \rfloor$ and $\lceil m/(k+1) \rceil$ and form a trie with the pieces. We also build a pessimistic d table with all the pieces (the longer pieces are pruned to build this table). This table stores, for each character, the smallest shift allowed among all the pieces. Now, at each text position we enter in the trie using the text characters from that position on. If we end up in a leaf, we found a piece, otherwise we did not. In any case, we use the d

table to shift to the next text position.

Suppose we find at text position i the end of a match for the subpattern ending at position j in the pattern. Then, the potential match must be searched in the area between positions $i-j+1-k$ and $i-j+1+m+k$ of the text, an $(m+2k)$-wide area. This checking must be done with an algorithm resistant to high error levels, such as dynamic programming.

This algorithm is the fastest in practice when the total number of verifications triggered is low enough, in which case the search cost is close to $O(kn/m) = O(\alpha n)$. We find out now when the total amount of work due to verifications is not higher.

An exact pattern of length ℓ appears in random text with probability $1/\sigma^\ell$. In our case, this is $1/\sigma^{\lfloor m/(k+1)\rfloor} \approx 1/\sigma^{1/\alpha}$. Since the cost to verify a potential match using dynamic programming is $O(m^2)$, and since there are $k+1 \approx m\alpha$ pieces to search, the total cost for verifications is $m^3\alpha/\sigma^{1/\alpha}$. This cost must be $O(\alpha)$ so that it does not affect the total cost of the algorithm. This happens for $\alpha < 1/(3\log_\sigma m)$. On English text we found empirically the limit $\alpha < 1/5$.

3 A New Verification Technique

In [7] we presented a different verification technique. We show now how to use it in this algorithm.

The idea is to try to quickly determine that the match of the small piece is not in fact part of a complete match. To explain the use of this technique, a stronger version of the Lemma must be used, which was proved in [7].

Stronger Lemma: If $segm = Text[a..b]$ matches pat with k errors, and $pat = p_1...p_j$ (a concatenation of subpatterns), then $segm$ includes a segment that matches at least one of the p_i's, with $\lfloor a_i k/A \rfloor$ errors, where $A = \sum_{i=1}^{j} a_i$.

We explain now the technique. First assume that $k+1$ is a power of 2. We use $A = 2$, $j = 2$, $a_1 = a_2 = 1$. That is, we split the pattern in two halves (halving also the number of errors). The stronger lemma states that at least one half must match with $\lceil k/2 \rceil$ errors. We recursively continue with this splitting until we reach the pieces that are to be searched directly (with no errors). Each time a leaf reports an occurrence, its parent node checks the area looking for its pattern (whose size is close to twice the size of the leaf pattern). Only if the parent node finds the longer pattern, it reports the occurrence to its parent, and so on. The occurrences reported by the root of the tree are the answers.

This construction is correct because the partitioning lemma applies to each

level of the tree, i.e. any occurrence reported by the root node *must* include an occurrence reported by one of the two halves, so we search both halves. The argument applies then recursively to each half.

Figure 1 illustrates this concept. If we search the pattern `"aaabbbcccddd"` with three errors in the text `"xxxbbbxxxxxx"`, and split the pattern in four pieces, the piece `"bbb"` will be found in the text. In the original approach, we would verify the complete pattern in the text area, while with the new approach we verify only if its parent `"aaabbb"` appears with one error and we immediately determine that there cannot be a complete match.

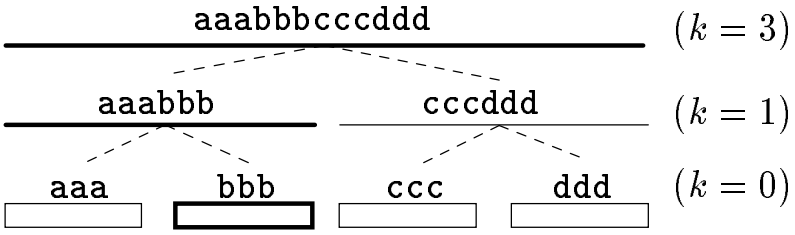

Fig. 1. The hierarchical verification method. The boxes (leaves) are the elements which are really searched, and the root represents the whole pattern. At least one pattern at each level must match in any occurrence of the complete pattern. If the bold box is found, all the bold lines may be verified.

If $k+1$ is not a power of two we try to build the tree as well balanced as possible (to avoid verifying a very long parent because of a very short child). For instance, if $k = 4$ ($k+1 = 5$), we partition the tree in, say, a left child with three pieces and a right child with two pieces. We then search the left subtree with $\lfloor 3k/5 \rfloor$ errors and the right one with $\lfloor 2k/5 \rfloor$ errors. Continuing with this policy we arrive to the leaves, which are searched with $\lfloor 4/5 \rfloor = 0$ errors each as expected.

Although when there are few matches (i.e. low error level) the old and new methods behave similarly, there is an important difference for medium error levels: the new algorithm is more tolerant to errors. Figure 2 illustrates the improvement obtained (the experimental setup is described in Section 5). As it can be seen, on random text the new method works well up to $\alpha = 1/2$, while the previous works well up to $\alpha = 1/3$. On the other hand, after that point the verifications cost much more than in the original method. This is because of the hierarchy of verifications which is carried out for most text positions when the error level is high. On the other hand, it is hard to improve the barrier of $\alpha < 1/2$ with this method, since at this point we are searching for single characters and performing a verification each time some of the characters is found in the text (which is too probable).

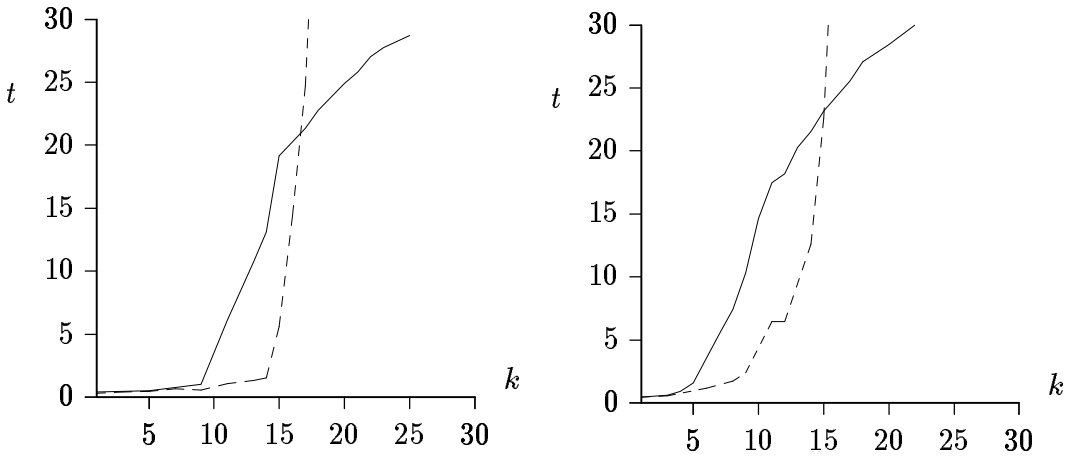

Fig. 2. The old (dashed) versus the new (full) verification technique. We use $m = 60$ and show random (left) and English text (right).

4 Optimizing the Partition

When splitting the pattern, we are free to determine the $k + 1$ pieces as we like. This can be used to minimize the expected number of verifications when the letters of the alphabet do not have the same probability of occurrence (e.g. in English text).

For example, imagine that $Pr('\texttt{e}') = 0.3$ and $Pr('\texttt{z}') = 0.003$. Then, if we search for `"eeez"` it is better to partition it as `"eee"` and `"z"` (with probabilities 0.0027 and 0.003 respectively) rather than `"ee"` and `"ez"` (with probabilities 0.09 and 0.0009 respectively). More generally, given that the probability of a sequence is the product of the individual letter probabilities, we want a partition that minimizes the sum of the probabilities of the pieces (which is directly related to the number of verifications to perform).

A dynamic programming algorithm to optimize the partition of $pat[0..m-1]$ follows. Let $R[i,j] = \prod_{r=i}^{j-1} Pr(pat[r])$ for every $0 \leq i \leq j \leq m$. It is easy to see that R can be computed in $O(m^2)$ time since $R[i,j+1] = R[i,j] * Pr(patt[j])$. Using R we build two matrices, namely

$P[i,k]$ = sum of the probabilities of the pieces in the best partition for $pat[i..m-1]$ with k errors.

$C[i,k]$ = where the next piece must start in order to obtain $P[i,k]$.

This takes $O(m^2)$ space. The following algorithm computes the optimal partition in $O(m^2 k)$ time.

```
for (i = 0; i < m; i++)
    P[i,0] = R[i,m];
```

```
    C[i,0] = m;
for (r = 1; r ≤ k; r++)
    for (i = 0; i < m − r; i++)
        P[i,r] = min_{j ∈ i+1..m−r}(R[i,j] + P[j,r − 1]);
        C[i,r] = j that minimizes the expression above;
```

The final probability of verification is $P[0,k]$ (note that we can use it to estimate the real cost of the algorithm in runtime, before running it on the text). The pieces start at $\ell_0 = 0$, $\ell_1 = C[\ell_0, k]$, $\ell_2 = C[\ell_1, k-1]$, ..., $\ell_k = C[\ell_{k-1}, 1]$.

As we presented the optimization, the obtained speedup is very modest and even counterproductive in some cases. This is because we consider only the probability of verifying. The search times of the extended Sunday algorithm degrades as the length of the shortest piece is reduced, as it happens in an uneven partition. We consider in fact a cost model which is closer to the real search cost. We optimize

$$\frac{1}{\text{minimum length}} + P(\text{verifying}) \times m^2$$

Figure 3 shows experimental results comparing the normal versus the optimized partitioning algorithms. The experimental setup is described in Section 5. However we repeated this experiment 100 times instead of 50 because of its very high variance.. This experiment is only run on English text since it has no effect on random text. Both cases uses the original verification method, not the hierarchical one. As it can be seen, the achieved improvements are especially noticeable in the intermediate range of errors.

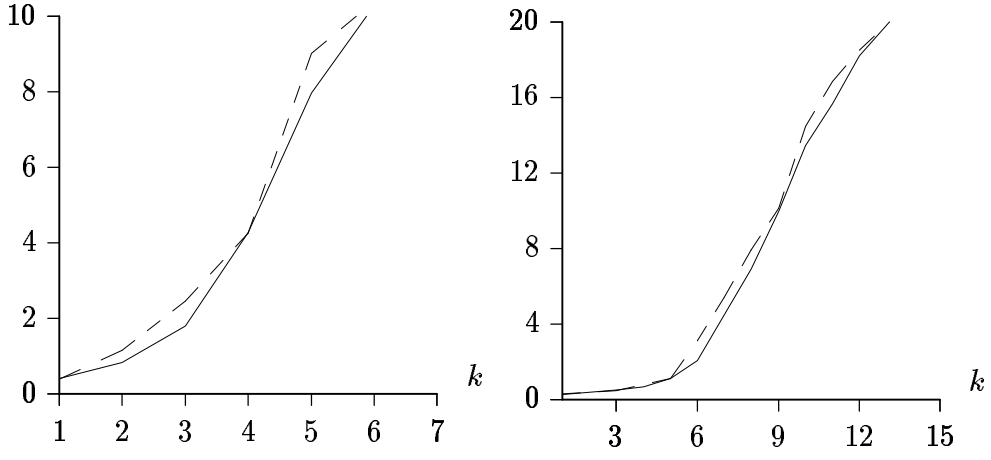

Fig. 3. The optimized (solid line) and the normal splitting (dashed), for $m = 10$ and 30 on English text.

5 Experimental Comparison

In this section we experimentally compare the old and new algorithms against the fastest algorithms we are aware of. Since we compare only the fastest algorithms, we leave aside [8,5,14,12,4,11,15] and many others, which are not competitive in the range of parameters we study here.

All the experimental results were obtained on a Sun UltraSparc-1 running Solaris 2.5.1, with 32 Mb of RAM, which is a 32-bit machine. We tested random text with $\sigma = 32$, and lower-case English text. The patterns were randomly selected from the text (at word beginnings in the English case). Except when otherwise stated, each data point was obtained by averaging the Unix's user time over 50 trials on 10 megabytes of text. We present all the times in seconds. The algorithms included in this comparison are: BYP [3] (i.e. the original version of this algorithm), BYN [2] and Myers [6] (the other fastest algorithms, based on bit-parallelism), and Ours (our modification to BYP with hierarchical verification). The code is from the authors in all cases.

On English text we add two extra algorithms: Agrep [13] (the fastest known approximate search software), and a version of our algorithm that includes the splitting optimization. On English text the code "Ours" corresponds to our algorithm with hierarchical verification and splitting optimization, while "Ours/NO" shows hierarchical verification and no splitting optimization.

The algorithms included in this comparison are: Agrep [13] (the fastest known approximate search software, only for English text), BYP [3] (i.e. the original version of this algorithm), BYN [2] and Myers [6] (the other fastest algorithms, based on bit-parallelism), Ours (our modification to BYP with both improvements) and Ours/NO (the same, including the hierarchical verification but not the splitting optimization). The splitting optimization is shown only for English text, of course. The code is from the authors in all cases.

As seen in Figure 4, for $\sigma = 32$ the new algorithm is more efficient than any other for $\alpha < 1/2$, while for English text it is the fastest for $\alpha < 1/3$. Notice that although Agrep is normally faster than BYP (i.e. the original version of this technique), we are faster than Agrep with the hierarchical verification, and the splitting optimization improves a little over this.

6 Conclusions and Future Work

We modified the fastest known algorithm for approximate string matching with the use of an improved technique to verify potential matches and a new

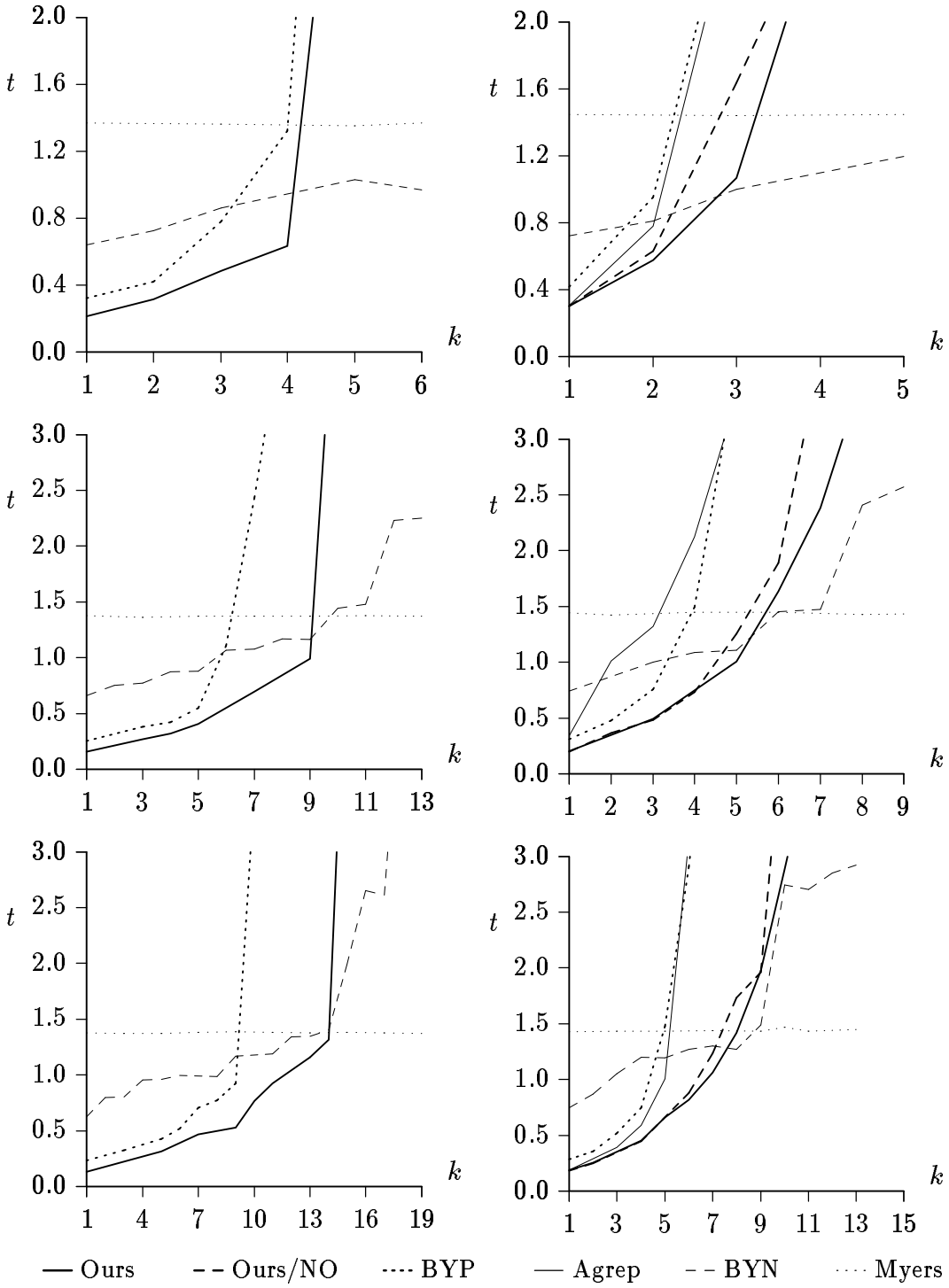

Fig. 4. Experimental results for random (left) and English text (right). From top to bottom m =10, 20 and 30.

optimization technique for biased texts. As a result, the area of applicability of the original algorithm is enlarged to spread almost all the interesting cases in approximate string searching, where it is still the fastest algorithm.

We are working on better cost functions for the splitting optimization technique. We also plan to study the online effect of splitting the pattern in more than $k+1$ pieces (so that more than one piece has to match), as suggested in [9] for offline searching.

References

[1] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 1–23, 1996.

[2] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 1998. To appear.

[3] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. *Information Processing Letters*, 59:21–27, 1996.

[4] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, 1992.

[5] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

[6] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, New Jersey, July 1998. To appear.

[7] G. Navarro and R. Baeza-Yates. Analysis for algorithm engineering: improving an algorithm for approximate string matching. Submitted, 1998.

[8] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[9] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.

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

[11] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, LNCS 979. Springer-Verlag, 1995.

[12] Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[13] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference*, pages 153–162, 1992.

[14] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.

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