Multiple Approximate String Matching by Counting

Gonzalo Navarro[1,2]

[1] Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.
[2] This work has been supported in part by Fondecyt Grants 1950622 and 1960881.

Abstract. We present a very simple and efficient algorithm for online multiple approximate string matching. It uses a previously known counting-based filter [9] that searches for a single pattern by quickly discarding uninteresting parts of the text. Our multi-pattern algorithm is based on the simulation of many parallel filters using bits of the computer word. Our average complexity to search r patterns of length m is $O(rn \log m / \log n)$, being n is the text size. We can search patterns of different length, each one with a different number of errors. We show experimentally that our algorithm is competitive with the fastest known algorithms, being the fastest for a wide range of intermediate error ratios. We give the first average-case analysis of the filtering efficiency of the counting method, applicable also to [9].

1 Introduction

A number of important problems related to string processing lead to algorithms for approximate string matching: text searching, pattern recognition, computational biology, audio processing, etc.

The *edit distance* between two strings a and b, $ed(a,b)$, is defined as the minimum number of *edit operations* that must be carried out to make them equal. The allowed operations are insertion, deletion and replacement of characters in a or b. The problem of *approximate string matching* is defined as follows: given a *text* of length n, and a *pattern* of length m, both being sequences over an alphabet Σ of size σ, find all segments (or "occurrences") of *text* whose edit distance to *pattern* is at most k, where $0 < k < m$. In typical text searching, n is large, m is small (say, less than 30), $\alpha = k/m$ is small (say, less than 1/3), and σ is not very small (at least 25). It is common to report only the endpoints of occurrences in the text.

In the online version of the problem, the pattern can be preprocessed but the text cannot. We are interested in online algorithms in this work. The classical solution is $O(mn)$ time and involves dynamic programming [13].

The different approaches to solve the problem efficiently can be divided in three main areas:

- Those that use cleverly the geometric properties of the dynamic programming matrix, e.g. [7, 11, 17, 5]. These algorithms normally achieve $O(kn)$ time complexity in the worst or the average case. An exception is [5], which achieves $O(kn/\sqrt{\sigma})$ on average.

- Those that filter the text, quickly leaving out most of the text and verifying only the areas that seem interesting, e.g. [16, 15, 14, 6, 4]. They achieve sublinear expected time in many cases (e.g. $O(kn \log_\sigma m/m)$) for small α. However, they tend to be practical only for m not too small and low error ratios. Some exceptions are [4, 9], which are $O(n)$ for small α even when m is small.

- Those that parallelize the computation of a classical algorithm in the bits of computer words [21, 19, 22, 2, 1]. We call w the number of bits in the computer word, which is assumed to be $\Theta(\log n)$. These algorithms normally obtain a factor of $O(1/\log n)$ over their classical counterparts. An exception is [2, 1], which is $O(n)$ for small patterns regardless of α, and $O(\sqrt{mk/\log n}\ n)$ for larger patterns and moderate α.

On the other hand, multi-pattern approximate search has only recently been considered. The only previous approaches we are aware of are [12] and [3]. In [12], hashing is used to handle thousands of patterns in parallel, although with only one error. In [3], extensions of [2] and [4] are presented that improve [12] for a moderate number of patterns, and can handle any number of errors.

In this work we extend a single-pattern approximate search algorithm to the case of multiple patterns. The algorithm that we extend is a filter based on counting matching positions [9]. The single-pattern filter is linear on average, and as any filtration algorithm, is useful up to a certain α value. The strongest point of that filter is its extreme simplicity. Despite that simplicity, it is among the fastest ones in its area of usefulness. Because of its simplicity, it can be parallelized in bits of a computer word.

Our multi-pattern algorithm is also competitive with previous work, being the fastest for intermediate error ratios. For r patterns and moderate α, it is $O(rn \log m/\log n)$ time. We present the new algorithm, give the first average-case analysis of its filtration efficiency (applicable also to [9]), and present experimental results for the single- and multi-pattern algorithms.

This paper is organized as follows. In section 2 we explain our minor variation of the single-pattern algorithm [9]. In section 3 we present our multi-pattern algorithm. In section 4 we analyze both algorithms. In section 5 we show experiments about the statistics of the problem and compare both algorithms against previous ones. Finally, in section 6 we give our conclusions.

2 A Simple Counting Filter

In this section we describe a minor variation of [9] (also very close to [8]), which is the basic single-pattern counting filter that our multi-pattern algorithm uses. Our approach is somewhat simpler because we use a fixed-size instead of variable-size text window (a possibility already noted in [18]).

We begin by proving a very simple lemma, which is a special case ($q = 1$) of Lemma 7 of [10].

Lemma: If there are $i \leq j$ such that $ed(text[i..j], pattern) \leq k$, then $text[j - m + 1..j]$ includes at least $m - k$ characters of *pattern*.

Proof: Suppose the opposite. If $j - i < m$, then we observe that there are less than $m - k$ characters of *pattern* in $text[i..j]$. Hence, more than k characters must be deleted from *pattern* to match the text. If $j - i \geq m$, we observe that there are more than k characters in $text[i..j]$ that are not in *pattern*, and hence we must insert more than k characters in *pattern* to match the text. A contradiction in both cases.

Note that in case of repeated characters in the pattern, they must be counted as different occurrences. For example, if we search `aaaa` with one error in the text, the last four letters of each occurrence must include at least three `a`'s.

The filter is based on the lemma. It passes over the text examining an m-letters long window. It keeps track of how many characters of *pattern* are present in the current text window (accounting for multiplicities too). If, at a given text position j, $m-k$ or more characters of *pattern* are in the window $text[j-m+1..j]$, the window area is verified with a classical algorithm (e.g. [17]). Verification is of course necessary, since the characters of the text could be at different positions in the pattern.

To avoid re-verification due to overlapping areas, we keep track of the last position verified and the state of the verification algorithm. If a new verification requirement starts before the last verified position, we start the verification from the last verified position, avoiding to re-verify the preceding area.

Observe that it is not necessary to verify the longer area $text[j - m - k + 1..j]$ (what would be the obvious area, since the occurrence can be of length up to $m + k$). This is because the lemma holds also for the window at *any* position inside an occurrence, so that the counter will reach $m-k$ also m characters past the *beginning* of the occurrence. A longer occurrence will keep triggering verifications while the window is inside the occurrence. This fact, together with our mechanism to avoid re-verifications by keeping the current state of verification, ensures that the occurrence will be caught.

We implement the filtering algorithm as follows: we build a table A where, for each character $c \in \Sigma$, the number of times that c appears in *pattern* is kept. We also keep a counter of matching characters. To advance the window, we must include the new character $text[j+1]$ and exclude the last character, $text[j-m+1]$.

To include the new character, we subtract one at the proper entry of A. If the entry was greater than zero before the operation, it is because the character is in *pattern*, so we increment the counter. To exclude the old character, we add one at the proper entry of A. If the entry is greater than zero after the operation, it is because the character was in *pattern*, so we decrement the counter. When the counter reaches $m-k$ we verify the preceding area.

Throughout the algorithm, each entry of A indicates how many occurrences of that character can still be taken as belonging to *pattern*. When it is negative, it means that that number of characters must exit the window before we take new characters. For example, if we run the pattern `aloha` over the text `aaaaaaaa`, it will hold $A[\mathtt{a}] = -3$, and the value of the counter will be 2.

Figure 1 shows the pseudocode of the algorithm. We use C notation. As it can be seen, the algorithm is not only linear (excluding verifications), but the number of operations per character is very small.

```
CountFilter (text,n,pat,m,k)
{                          /* preprocessing */
   for (c ∈ Σ) A[c] = 0;
   for (i = 0; i < m; i++) A[pat[i]]++;
   count = -(m - k);
                           /* searching */
   for (j = 0; j < m; j++) /* fill the initial window */
      if (A[text[j++]]-- > 0) count++;
   while (j ≤ n)           /* move the window */
    { if (count ≥ 0) { verify text[j - m..j - 1] with dynamic programming }
      if (++A[text[j - m]] > 0) count--;
      if (A[text[j++]]-- > 0) count++;
    }
}
```

Fig. 1. The code of the single-pattern algorithm.

3 Our Multi-pattern Search Algorithm

To search r patterns in the same text, we use bit-parallelism to keep all the counters in a single machine word. We must do that for the A table and for *count*.

The values of the entries of A lie in the range $-m..m$, so we need exactly $1 + \lceil \log_2(m+1) \rceil$ bits to store them. This is also enough for *count*, since it is in the range $-(m-k)..k$. Hence, if we call w the number of bits in the computer word, we can pack

$$\left\lfloor \frac{w}{1 + \lceil \log_2(m+1) \rceil} \right\rfloor$$

patterns of length m in a single search. If the patterns have different lengths, this limit holds for the longest one. If we have more patterns, we must divide

the set in subsets of at most this size and search each subset separately. For example, in a 32-bit architecture we can handle in a single word up to 8 patterns of length 7, or 6 of length 15, or 5 of length 31, or 4 of length 127, etc. We focus our attention on a single subset now.

The algorithm simulates the simple one as follows. We have a table MA that packs all the A tables. Each entry of MA is divided in bit areas of the appropriate length. In the area of the machine word corresponding to each pattern, we store its normal $A[]$ value minus 1, and set to 1 the most significant bit of the area. If we have to add or subtract 1, we can easily do it in parallel without causing overflow from an area to the next. Moreover, the corresponding $A[]$ value is not positive if and only if the most significant bit of the area is zero.

We have a parallel counter $Mcount$, where the areas are aligned with MA. It is initialized by setting to 1 the most significant bit of each area and then subtracting $m - k$ at each one. We can add or subtract one in parallel without causing overflow. Moreover, the window must be verified for a pattern whenever the most significant bit of its area reaches 1. The condition can be checked in parallel, although each verification is sequential. Note that this allows to have different k values for each pattern. It is also possible to have different m values, but the performance of the algorithm may be degraded if they are very different, because we have to use the longest text window for all the patterns.

Observe that the counters that we want to increment or decrement correspond exactly to the MA areas that have a 1 in their most significant bit. This allows an obvious bit mask-shift-add mechanism to perform this operation also in parallel.

Figure 2 shows the pseudocode of the parallel algorithm. As it can be seen, the algorithm is now more complex but the number of operations per character is still very low.

4 Analysis

We analyze the space requirements and time complexity of the algorithms for single and multiple patterns.

The space requirement of all the algorithms is $O(\sigma)$. The preprocessing cost is $O(\sigma+m)$ for the simple algorithm and $O(\sigma+rm)$ for the multi-pattern algorithm.

If the number of verifications is negligible, each pass of the algorithms is $O(n)$. That means that the simple algorithm is $O(n)$. In the case of multiple patterns, only $O(w/\log m)$ patterns can be packed in a single search, so the cost to search r patterns is $O(rn \log m/\log n)$ (taking $w = \Theta(\log n)$ as usual in the RAM model).

In the worst case all the text positions are verified, and the algorithms take the same as dynamic programming, i.e. $O(mn)$ the simple one and $O(rmn)$ the multi-pattern one. This is because we avoid re-verifying a text position, even in the case of overlapping verification requirements.

The difficult part of the analysis is the maximum error ratio α that the filtra-

```
CountFilter (text, n, pat_{1..r}, m_{1..r}, k_{1..r})

{       /* preprocessing */
   m = max{m_s, s ∈ 1..r};
   ℓ = ⌈log_2 m⌉;
   for (c ∈ Σ) MA[c] = (01^ℓ)^r;
   for (s = 0; s < r; s++)
      for (i = 0; i < m_s; i++) MA[pat_s[i]] += 10^{s(ℓ+1)};
   high = (10^ℓ)^r;
   ones = (0^ℓ 1)^r;
   Mcount = (10^ℓ − (m − k)) × ones;

        /* searching */
   j = 0;
   while (j < m)      /* fill the initial window */
    { c = text[j++];
      Mcount += (MA[c] >> ℓ) & ones;
      MA[c] −= ones;
    }
   while (j ≤ n)      /* move the window */
    { if (Mcount & high)! = 0 then      /* verify the area */
         verify text[j − m..j − 1] with dynamic programming
           (for each pattern whose high Mcount bit is 1)
      c = text[j − m];
      MA[c] += ones;
      Mcount −= (MA[c] >> ℓ) & ones;
      c = text[j];
      Mcount += (MA[c] >> ℓ) & ones;
      MA[c] −= ones;
    }
}
```

Fig. 2. The code of our multiple-pattern algorithm. The exponentiation of bits means repetition, e.g. $0^3 1 = 0001$.

tion scheme can tolerate while keeping the number of verifications low. If the probability of verifying is $O(1/m^2)$ the algorithm keeps linear on average. If it exceeds $1/m$, it becomes completely ineffective. This is because the verifications cost $O(m^2)$, and hence this is the point where the algorithm becomes $O(mn)$, the same as plain dynamic programming. We call that point the "limit of usability", and say that the algorithm is "useful" before that limit.

We derive in the Appendix a pessimistic bound for the limit of linearity and usability, namely $\alpha < e^{-m/\sigma}$ (Eq. (2)). The analysis shows that, as m grows, we can tolerate smaller error ratios. This is experimentally verified in the next section.

5 Experiments

We first show experiments about the maximum allowable error ratio for the filter (i.e. up to where it is better than plain dynamic programming). Later, we compare both algorithms against others.

5.1 Maximum Error Ratio

We experimentally find out which is the limit of usability of the algorithm for different types of texts, and use least squares to find a formula which is very accurate for the range of values we are interested in practice, i.e. $m \leq 100$ and $20 \leq \sigma \leq 60$. That type of formula was selected among a number of classes we tried, since it gave us the best results. It is close in spirit to Eq. (2) (recall that that equation is pessimistic).

The experiments were carried out as follows. For every σ in the set $\{20, 30...60\}$ and every m in $\{4..100\}$, we generated a random text of 1 Mb, and repeated 100 times the experiment of generating a random pattern and verifying which was the maximum error (k) up to where the number of verifications triggered was less than $1/m$ times the size of the text.

Separately for each value of σ, we used least squares for the model $\alpha_{\max} = ab^m$, which gave us the best results. Later, once a different value of a and b was obtained for each σ, we used the models $a = c\sigma^d$ and $b = 1 - f\sigma^g$. The result is the formula

$$\alpha_{\max} = 0.11\ \sigma^{0.43}(1 - 0.032/\sigma^{0.37})^m$$

for which we obtained an average squared error near 0.0004 (its square root being 0.02).

We also performed the test on English lower-case text, selecting the patterns randomly from the same text at the beginning of non-stopwords, to mimic classical information retrieval queries.

The experimental results are shown in Figure 3. The smooth curves are those obtained with least squares. The theoretical pessimistic approximations found

in the Appendix are totally below the experimental curves, but have the same shape of those of least squares. Therefore, they are less exact for very small or very large m. In the first case this is because the analysis works with probabilities of the form $O(1/m)$, which allows larger errors for small m. In the second case it is because the pessimistic part of the model refers to letters that appear many times in the text window of length m, which is more noticeable for large m (when it is more probable to repeat letters).

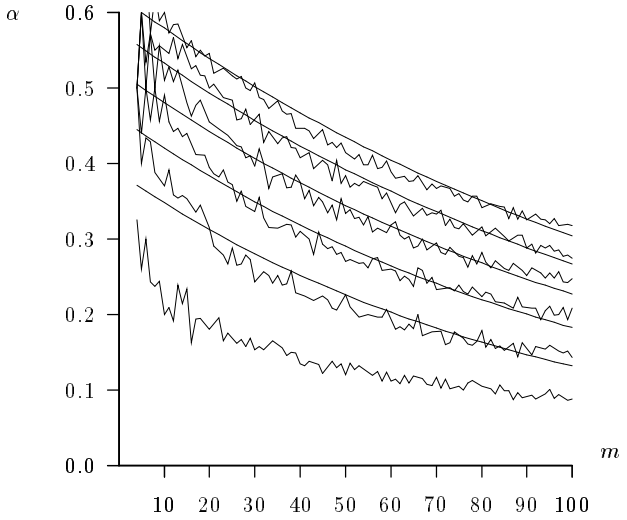

Fig. 3. Experimental maximum level of usefulness of our algorithm. The lowest line is for English lowercase text. The rest of non-smooth lines are for $\sigma = 20, 30...60$ (from lower to higher). The smooth lines are our approximation.

5.2 Comparison with Other Algorithms

In this section we experimentally compare the algorithms against the fastest previous algorithms we are aware of. We leave aside a number of algorithms that were not competitive in our experiments, at least for the range of parameters we used.

We tested random patterns against 1 Mb of random text on a Sun SparcStation 4 running Solaris 2.3, with 32 Mb of RAM. We use $w = 32$ and $\sigma = 30$ (typical case in text searching). We also tested on 1 Mb of lower-case English literary text, where the patterns were randomly selected from the same text, at the beginning of non-stopwords. Since the algorithms work on main memory, we measure user times. Each data point was obtained by averaging over 10 trials.

Single Patterns We compare the counting filter against: Ukkonen [17], Chang-Lampe [5], Sutinen-Tarhio [14], Baeza-Yates/Perleberg [4], Wu-Manber [21], Wu-Manber-Myers [22], Agrep [20], and Baeza-Yates/Navarro [2].

In all cases the code is from the authors, except Ukkonen (code is ours and in fact it is used as our verification engine), Baeza-Yates/Perleberg (code is ours) and Wu-Manber (code is from Alden Wright [19]). The codes have been used with the parameters suggested by the authors to achieve optimal behavior, and we verified that that was the case. The code for the counting filter is that of Figure 1.

Figures 4 and 5 (upper part) show the results (for random and English text, respectively). Although we show results only for $m = 20$, similar results were obtained for $m = 10$ and 30.

It can be seen that in the area of usefulness the algorithm is very fast, being competitive with the best known algorithms. It is even the fastest in a small area of moderate error ratio. We observe that the point where the algorithm is the fastest is close to its maximum level of usability (which can be noted because it worsens immediately after that point, or by looking at the values in Figure 3). The reason for this is that the filter, though simple and fast, inspects all text characters and therefore is not faster than the best sublinear filters. However, it is more tolerant to errors and therefore it is the best one between the point where the other filters stop working up to where it also stops working because of the error level.

Multiple Patterns There are few previous algorithms for multiple approximate matching. We compare ours against all them: Muth-Manber [12] (which is defined only for $k = 1$) and Baeza-Yates/Navarro [3] (which proposes 3 algorithms: "exact partitioning" (EP), and "superimposed automata" with two variants, namely "pattern partitioning" (PP) and "automaton partitioning" (AP)).

Figures 4 and 5 (lower part) show some comparisons (for random and English text, respectively). Although we show only the case $m = 20$, similar results were obtained for $m = 10$ and $m = 30$.

We first compare all the algorithms for $k = 1$ (so that Muth-Manber can be included), to show that our algorithm is better than Muth-Manber for r up to 10. However, Baeza-Yates/Navarro (EP) is the fastest algorithm in this area.

We then compare all the algorithms for more errors (hence excluding Muth-Manber), for fixed k ($k = 7$ on random text and $k = 4$ on English text, i.e. moderate error level) and for fixed $r = 15$. As it can be seen, our algorithm is the best one for a moderate number of errors (i.e. in the last plot, from the point where Baeza-Yates/Navarro (EP) is eliminated for its many verifications to where the same happens to our algorithm). In the multi-pattern case, the area where our algorithm is the fastest one is wider than in the single pattern case.

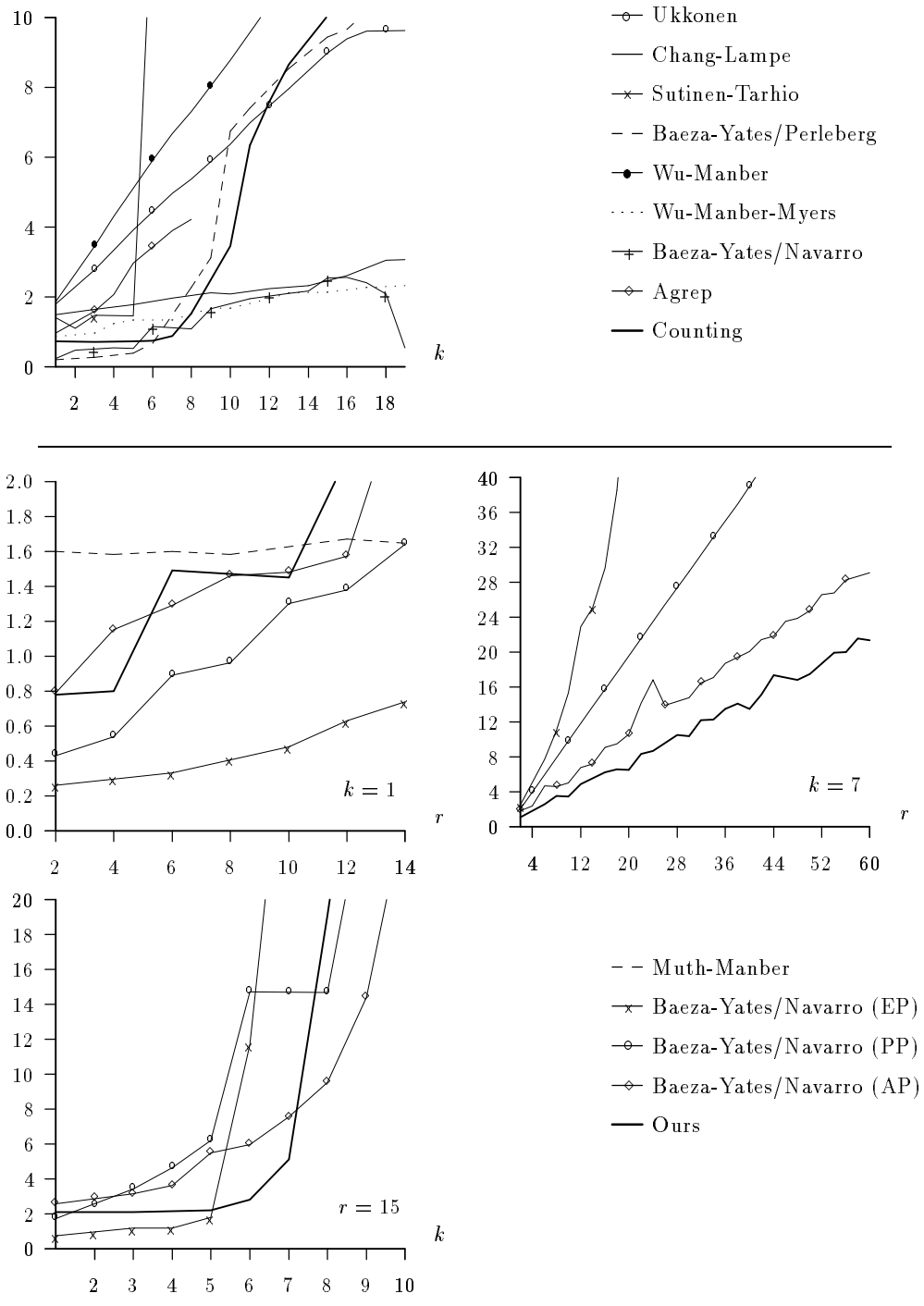

Fig. 4. Times in seconds for different variations of the simple (upper part) and multiple (lower part) search problem. The plots are for $m = 20$ and random text with $\sigma = 30$.

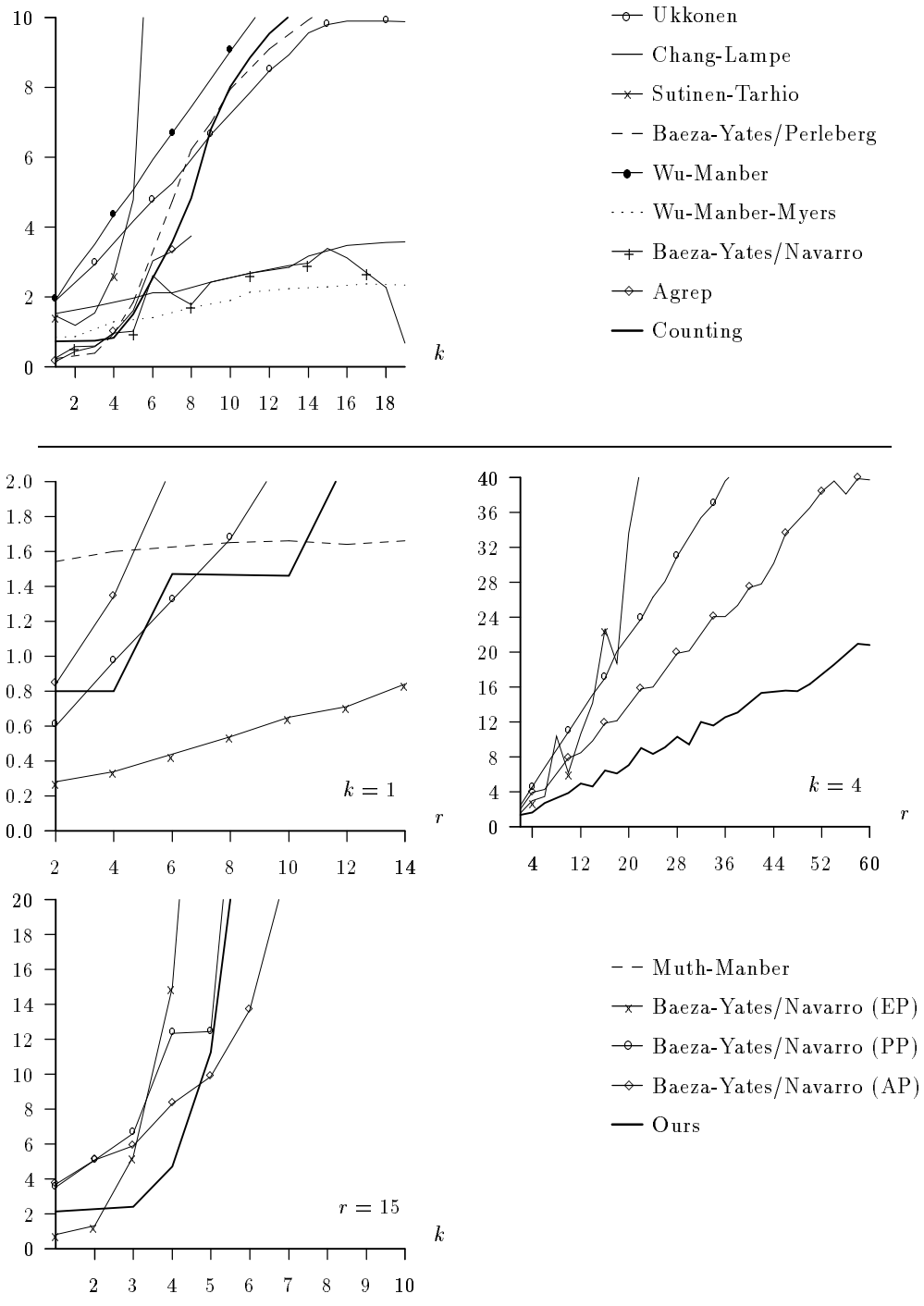

Fig. 5. Times in seconds for different variations of the simple (upper part) and multiple (lower part) search problem. The plots are for $m = 20$ and English text.

6 Conclusions

We presented a new algorithm for multiple approximate string matching. Our algorithm is the fastest one for intermediate error ratios and can search patterns with different lengths and number of errors. Up to a given error ratio, it is linear on average and very fast in practice. It performs a few operations per inspected character. Its tolerance to errors is more than enough for most text searching applications. The algorithm is based on a bit-parallel simulation of a previous filter based on counting matching positions [9].

We analyze and experimentally show which is the maximum error ratio up to where the algorithms filtrate efficiently, giving the first average-case analysis for the counting filter. We experimentally compare both algorithms against the fastest we are aware of. In their area of usefulness, they are competitive with the best ones. Moreover, the algorithms are the fastest for a range of intermediate error values (this range is wider for our multi-pattern version). This is due to the fact that, although the algorithms are not faster than the best filters, their tolerance to errors is higher.

Acknowledgments

We thank Ricardo Baeza-Yates for his support and numerous useful comments to improve this work. Thanks also to Erkki Sutinen, who gave us useful references to related works, and Jorma Tarhio for an early version of [9]. Finally, we thank an anonymous referee for its comments and references.

References

1. R. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In *Proc. WSP'96*, pages 47–63, 1996. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/wsp96.2.ps.gz`.

2. R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, pages 1–23, 1996. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/cpm96.ps.gz`.

3. R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS'97*, pages 174–184, Halifax, Nova Scotia, Canada, 1997. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/wads97.ps.gz`.

4. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, pages 185–192, 1992. LNCS 644.

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

6. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, pages 259–273, 1994. LNCS 807.

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

8. R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *IPL*, 33(3):113–120, Nov. 1989.

9. P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.

10. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, pages 240–248, 1991. LNCS 520.

11. G. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer Systems Science*, 37:63–78, 1988.

12. R. Muth and U. Manber. Approximate multiple string search. In *Proc. CPM'96*, pages 75–86, 1996.

13. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

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

15. T. Takaoka. Approximate pattern matching with samples. In *Proc. ISAAC'94*, pages 234–242, 1994. LNCS 834.

16. J. Tarhio and E. Ukkonen. Boyer-Moore approach to approximate string matching. In *Proc. SWAT'90*, pages 348–359, 1990. LNCS 447.

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

18. E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 1:191–211, 1992.

19. A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, Apr. 1994.

20. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX*, pages 153–162, 1992.

21. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, Oct. 1992.

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

Appendix: Probability of Verifying

We find an upper bound for the probability of triggering a verification, and use it to derive a safe limit for α to make verification costs negligible. We consider constant α and varying m (the results are therefore a limit on α). We then extend the results to the other cases.

The upper bound is obtained by using a pessimistic model which is simpler than reality. We assume that every time a letter in the text window matches the pattern, it is counted regardless of how many times it appeared in the window. Therefore, if we search `aloha` with 1 error in the text window `aaaaa` the verification will be triggered because there are 5 letters in the pattern (where in fact our counter will not trigger a verification because it counts only 2 `a`'s).

Consider a given letter in the text window. The probability of that letter being counted is that of appearing in the pattern. This is the same as being equal to some letter of the pattern. The probability of *not* being equal to a given letter is $(1-1/\sigma)$. The probability of being in the pattern is therefore $p = 1-(1-1/\sigma)^m$.

In our simplified model, each pattern letter is counted independently of the rest. Therefore the number X of letters in the text window that matched the pattern is the sum of m (window length) random variables that take the value 1 with probability p, and zero otherwise. This has a Binomial distribution $B(m,p)$.

Our question is therefore when the probability $P(X \geq m-k)$ is $O(1/m^2)$ (so that the algorithm is linear) or when it is $O(1/m)$ (so that it is useful). In the proof we use $O(1/m^2)$, since as we see shortly the result is the same for any polynomial in $1/m$.

We first analyze the case where the mean of the distribution is below $m-k$, i.e. $mp < m-k$. This is the same as the condition $\alpha < 1-p$.

We begin by showing that, if X has a binomial distribution $B(m,p)$ and $j > mp$, then $P(X \geq j) = O(P(X=j))$, i.e. the first term of the summation of probabilities dominates the rest once we passed the mean of the distribution. If we call $p_r = P(X=r)$, we have

$$P(X \geq j) \quad = \quad \sum_{r=j}^{m} p_r \quad = \quad \sum_{r=j}^{m} \binom{m}{r} p^r (1-p)^{m-r}$$

and we observe that

$$\frac{p_{r+1}}{p_r} \quad = \quad \frac{(m-r)p}{(r+1)(1-p)} \quad \leq \quad \frac{mp}{(r+1)} \quad \leq \quad C \quad < \quad 1$$

where the inequalities come from the fact that $r \geq mp$. $C = mp/(mp+1)$ is a constant which acts as a fixed upper bound for all p_{r+1}/p_r which is independent of r and smaller than 1. Therefore, the terms of the summation decrease at least by a multiplicative constant, what makes their sum a constant proportion

of the first summand, i.e. Dp_j, where the constant D is bounded above by $D = 1/(1-C)$.

Therefore, it suffices to prove that $P(X = m-k) = O(1/m^2)$. By using the Stirling approximation to the factorial we have

$$P(X = m-k) \quad = \quad \binom{m}{k} p^{m-k}(1-p)^k \quad = \quad \frac{m^m p^{m-k}(1-p)^k}{k^k (m-k)^{m-k}}\, O(\sqrt{m})$$

which can be rewritten as

$$\left(\frac{p^{1-\alpha}(1-p)^\alpha}{\alpha^\alpha (1-\alpha)^{1-\alpha}}\right)^m \; O(\sqrt{m})$$

It is clear that the above formula is $O(1/m)$ or $O(1/m^2)$ whenever the base of the exponential is < 1. This is

$$p^{1-\alpha}(1-p)^\alpha < \alpha^\alpha (1-\alpha)^{1-\alpha} \tag{1}$$

To determine when the above condition is valid, we define the function

$$f(x) = x^\alpha (1-x)^{1-\alpha}$$

which reaches its maximum at $x = \alpha$. This shows that Eq. (1) holds everywhere, and therefore the probability of matching is $O(1/m^2)$ in this case, i.e. whenever $\alpha < 1-p$.

On the other hand, if the median of the distribution is beyond $m-k$, then just the term of the summation corresponding to the median $r = mp$ is (using Stirling again)

$$\binom{m}{mp} p^{mp}(1-p)^{m(1-p)} \quad = \quad \left(\frac{p^p(1-p)^{1-p}}{p^p(1-p)^{1-p}}\right)^m \; \Omega(m^{-1/2}) \quad = \quad \Omega(m^{-1/2})$$

which is not $O(1/m)$.

Therefore, we arrive to the conclusion that the filter is linear and useful whenever

$$\alpha < 1-p = \left(1-\frac{1}{\sigma}\right)^m = e^{-m/\sigma}\;\; (1+O(1/\sigma)) \tag{2}$$

and is not useful otherwise.

We have considered the case of constant $\alpha = k/m$. Obviously, the filter is linear for $k = o(m)$ and is not useful for $k = m - o(m)$. The unexplored area is $k = mp - o(m)$. It is easy to see that the filter is not useful in this case, by considering $P(X = mp + \epsilon)$ with $\epsilon = o(m)$, and using Stirling. The resulting condition is $1 - \epsilon^2/(m^2 p(1-p)) = O(m^{-1/2})$, which does not hold for any $\epsilon = o(m)$.

This article was processed using the LaTeX 2_ε macro package with CUP_CS class