Approximate Pattern Matching with Samples*

Tadao TAKAOKA

Department of Computer Science
Ibaraki University, Hitachi, Ibaraki 316, JAPAN
E-mail: takaoka@cis.ibaraki.ac.jp

Abstract. We simplify in this paper the algorithm by Chang and Lawler for the approximate string matching problem, by adopting the concept of sampling. We have a more general analysis of expected time with the simplified algorithm for the one-dimensional case under a non-uniform probability distribution, and we show that our method can easily be generalized to the two-dimensional approximate pattern matching problem with sublinear expected time.

1 Introduction

Since the inaugural papers on string matching algorithms were published by Knuth, Morris and Pratt[11] and Boyer and Moore [5], the problem diversified into various directions. Let us call string matching one-dimensional pattern matching. One is two-dimensional pattern matching and the other is approximate pattern matching where up to k differences are allowed for a match. Yet another theme is two-dimensional approximate pattern matching. There are numerous papers in these new research areas. We cite just a few of them to compare our results with the previous works. In the one-dimensional approximate pattern matching problems, there are two variations. One is the matching problem with k mismatches, where k mismatches are allowed for a match. The other is that with k differences, where a difference is a mismatch, a superfluous character, or a missing character. For the former problem, Landau and Vishkin [12] gave an algorithm with $O(kn)$ time and $O(m \log m)$ preprocessing time for a pattern, where m and n are the lengths of pattern and text. For the k difference problem, Ukkonen [14], Landau and Vishkin [13] and Galil and Park [8] gave algorithms with $O(kn)$ time for matching and $O(m^2)$ time for preprocessing. Ukkonen's algorithm [15] runs in $O(n)$ time at the cost of exponential preprocessing time. The approximate string matching problem with minimum differences is discussed in Erickson and Sellers [7]. These are algorithms with the worst case analysis. For the expected time, Chang and Lawler gave a very efficient algorithm with sublinear $O((kn/m) \log m / \log r)$ expected time for matching and $O(m)$ preprocessing time where r is the size of alphabet. Here we assume the base of logarithm is 2 and we modified the time complexity of their notation of $O((kn/m) \log_r m)$.

* Partially supported by Grant-in-aid No. 06249201 by Monbusho Scientific Research Program and a research grant from Hitachi Engineering Co., Ltd.

It is this algorithm that we try to improve in this paper; a much simpler algorithm and a more general analysis of expected time under a wider probabilistic assumption, and generalization to two-dimensional cases.

Turning to the two-dimensional pattern matching, the first linear time algorithms were given by Baker [3] and Bird [4] with $O(n^2)$ time for matching and $O(m^2)$ time for preprocessing. Zhu and Takaoka [18] gave a randomized algorithm with the same time complexity. Precisely speaking, the time complexities of these algorithms depend on the alphabet size. Recently Galil and Park [9] gave an algorithm with time complexity independent of the alphabet size. In the area of two-dimensional approximate pattern matching, Amir and Landau [1] gave algorithms with $O(m^2)$ preprocessing time and (kn^2) matching time for the k mismatch problem and $O(k^2n^2)$ matching time for the k difference problem where superfluous characters and missing characters are only given in rows or columns, that is, the effects do not propagate to the next rows or columns. This definition does not seem as natural as the k difference problem in the one-dimensional case and the k mismatch problem seems adequately useful in graphics. In the present paper, we give an algorithm with sublinear expected time for the k mismatch problem, which is derived by applying the sampling method to the two-dimensional case.

The central idea in this paper is sampling. That is, we take small samples from the text sparsely and with the aid of information obtained by preprocessing the pattern we discard quickly many candidate positions of the text for an approximate match. Extracting sample information from the pattern was done by Karp and Rabin [10] and Vishkin [16], whereby we can have a somewhat compressed pattern for matching. In the present paper we extract samples from the text for approximate matching. Samples of $O(\log m)$ length are drawn from text sparsely at $O(m/k)$ periods. Samples of the same length are drawn densely from pattern, that is, at every position of pattern. The length is chosen so that the probability of sample match is $O(m^{-3})$, that is, many positions of text for an approximate match can be discarded quickly. The period is determined so that there must be at least one sample match for a position to be a candidate for an approximate match. To handle samples easily, they are converted into numerical values. The converted samples of pattern are organized in a data structure, called position table. The table is accessed by the numerical values. Each entry of the table contains the list of positions of the samples that have the numerical value corresponding to the table entry. This table is viewed as an approximate alternative of the position (or suffix) tree of the pattern used in [6], which gives the positions of the substrings of the pattern. Our matching strategy is similar to that of [6]. As we scan the text, we convert text samples into numerical values and access the table with these numerical values. If these values have positions in the table, that is, text samples match pattern samples, we take up the corresponding text positions as candidates and perform exhaustive check. Since the position table is simpler than the position tree, our algorithm is easier to implement and expected to run faster than the position tree version. A more important aspect of our algorithm is that it is easy to extend our algorithm to

the two-dimensional approximate pattern matching. The expected time of our algorithm for the one-dimensional case is basically equal to that of the position tree version, which is sublinear. The expected time of our algorithm for the two-dimensional approximate matching is sublinear $O((kn^2/m^2)\log m)$, which is a new result and expected to have wide applications in graphics, such as template matching.

In [6], the authors implicitly assume in their analysis a uniform distribution over the character occurrences. Suppose that we have an overwhelming probability of the occurrence of a character, then we have almost always a match and we can not discard candidate positions easily. In our paper, we amend this point and incorporate into our analysis the probability that a character of pattern matches one of text under a non-uniform distribution. The expected time in the previous paragraph is given, assuming this probability is a constant. The base of logarithm is assumed to be two unless otherwise stated.

2 Probabilistic Analysis of Matching and Sampling

We give the pattern in array $pat[0\,..\,m-1]$ and the text in array $text[0\,..\,n-1]$. We say we have a match at j, if $pat[i] = text[j+i]$ for $i = 0, \cdots, m-1$, which is denoted by $pat[0\,..\,m-1] = text[j\,..\,j+m-1]$.

We assume that the characters in pat and $text$ are drawn independently from a fixed distribution with probability $p(c) > 0$ for character c. Let the alphabet be Σ. Then we have $\sum_{c\in\Sigma} p(c) = 1$. The probability q that a character in pat matches one in $text$ is given by $q = \sum_{c\in\Sigma} p^2(c)$. The value of q is minimum $1/r$ when all $p(c)$ are equal to $1/r$ where $|\Sigma| = r$. Therefore we have $1 < 1/q \leq r$.

Now let pat be put at position j. Then the probability that $pat[i] = text[j+i]$ is given by q and this event is independent from the event that $pat[i'] = text[j+i']$ for $i' \neq i$. We extend this fact to the following lemma.

Lemma 1. *Let the event that $pat[i\,..\,i+\ell-1] = text[j\,..\,j+\ell-1]$ be denoted by $Eve(i,j)$. We assume that ℓ is a fixed length of the substrings. Then we have that $prob[Eve(i,j)] = q^\ell$.*

The lemma is illustrated in the following figure.

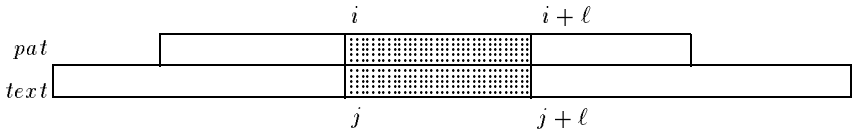

Fig. 1. Events $Eve(i,j)$

Now we introduce the idea of sampling into pat and $text$. Samples are substrings of length ℓ of pat and $text$. We extract samples densely from pat and sparsely from $text$. That is, we define samples from pat, denoted by $p\_sam(i)$, by

$$p\_sam(i) = pat[i\,..\,i+\ell-1], \text{ for } i = 0, 1, \cdots, m-\ell\ ,$$

and samples from $text$, $t\_sam(j)$, by

$$t\_sam(j) = text[j\,..\,j+\ell-1]\ , \text{for } j = 0, h, 2h, \cdots, (L-1)h\ ,$$

where $L = \lceil n/h \rceil$ and h is the sampling period. For our approximate pattern matching purpose, we determine the values of ℓ and h as follows:

$$\ell = \lceil 3 \log m / \log(1/q) \rceil,\ h = \lceil m/(k+1) \rceil\ ,$$

where k is the number of differences allowed for approximate matching. As we require that there be no overlaps between samples in Lemma 2, we need the condition that $k \leq O(m \log(1/q)/\log m)$. The value of ℓ is determined so that the probability that samples of pat and $text$ match is very small and thus many candidate positions for approximate matches can be discarded quickly. The value of h is determined to detect unallowable differences efficiently as we see in the following lemma.

Lemma 2. *If we have an approximate match with up to k differences when pat is positioned at j, there is at least one sample of pat that matches one of those of $text$.*

A similar idea of partitioning the pattern into $k+1$ sections is seen in [2] and [17], in which the idea of sampling is not used. The effect of the length ℓ of samples is seen from the following lemma.

Lemma 3. *When pat is positioned at j, the probability that d samples of pat matches those of $text$ is given by $\binom{k+1}{d} q^{d\ell} \leq \binom{k+1}{d} m^{-3d}$.*

A statement on probabilistic analysis in [6] is reproduced in the following.

Lemma 4. *Let pat be positioned at j. Then several samples of $text$ may match the corresponding samples in pat. Similar events may occur if pat is positioned at other positions j'. Based on these events, we do some work on the $text$ starting at j. Then we have*

$$\begin{aligned} & E[E[\textit{work at start position } j \textit{ given any conditioning}]] \\ = {} & E[\textit{work at start position } j]\ , \end{aligned}$$

where $E[X]$ is the expected value of random variable X.

3 Conversion of Sample Strings into Numbers

To handle the samples of pat and $text$ easily, we convert them into numbers. Let an arbitrary array $a[0\,..\,\ell-1]$ of integer type $0\,..\,r-1$ be converted into an integer value $num(a)$ by

$$num(a) = a[0]r^{\ell-1} + a[1]r^{\ell-2} + \cdots + a[\ell-1]\ .$$

The function num is a bijection between r^ℓ possible strings of length ℓ over alphabet $0\,..\,r-1$ and the integers $0, \cdots, r^\ell - 1$. The conversion of the samples of pat and $text$ into numbers can be done by converting each character c into integer $num(c)$ which is a bijection between the alphabet and the integers $0, \cdots, r-1$ and the following algorithms.

Algorithm 1.

Compute $num(p\_sam(0)) = num(pat[0])r^{\ell-1} + \cdots + num(pat[\ell-1])$
by Horner's method;
Compute $M = r^{\ell}$;
for $i := 1$ to $m - \ell$ **do**
$num(p\_sam(i)) := num(p\_sam(i-1))r - num(pat[i-1])M + num(pat[i+\ell-1])$;

Algorithm 2.

for $j := 0$ **to** $(L-1)h$ **step** h **do** compute $num(t\_sam(j))$ by Horner's method.

The computing time of Algorithm 1 is $O(m)$ and that of Algorithm 2 is $O((n/(m\log(1/q)))k\log m)$.

4 Construction of Position Table

The position table of *pat* is similar to the suffix tree used in [6]. It gives the positions of the samples in *pat* like the suffix tree gives the positions of substrings of *pat*. The b-th component of the position table is t, $t[b]$, is the list of positions i of $p\_sam(i)$ such that $b = num(p\_sam(i))$. Technically $t[b]$ is the pointer to the first element of the list. We can construct the position table by the following algorithm. Let $M = r^{\ell} = O(m^{3\log r/\log(1/q)})$.

Algorithm 3.

for $b := 0$ **to** $M-1$ **do** $t[b]$:=nil;
for $i = 0$ **to** $m - \ell$ **do begin**
$b := num(p\_sam(i))$; Append i to $t[b]$
end.

The computing time of Algorithm 3 is $O(M)$. To avoid the $O(M)$ time and space, we use a hash table of $O(m)$ size. That is, we compute the value of $f(b)$ of b with some hash function f such as the function by the linear congruential method and access the table with $f(b)$. Then the positions are distributed over the table with the expected size of list $t(b)$ being $O(mp(b))$ where $p(b)$ is the probability of the occurrence of $p\_sam(i)$ such that $b = p\_sam(i)$.

5 The Main Matching Algorithm

The idea of approximate matching in this paper is to find possible positions on text at which *pat* may have an approximate match by using $b = num(t\_sam(j))$ and the positions in $t[b]$. More specifically, if i is in $t[b]$, the position $j-i$ on $text$ is a candidate for an approximate match and so we perform an exhaustive check starting at $j-i$ on $text$. By an exhaustive check we mean we use an algorithm for the k-difference problem for $pat[0\,..\,m-1]$ and $text[j-i\,..\,j-i+m-1]$. We come to the end of *pat* with no more than k-differences, reporting "success," or find

$k+1$ differences on the way and discard the position $j-i$. Using the algorithm for the k-difference problem in [8], we can do exhaustive check in $O(m^2)$ time with $O(m \log r)$ time for preprocessing the pattern.

Now we describe the data structure array $pos[0\,..\,L-1]$ for maintaining the candidate positions. Each component of pos is a list of positions. If $t\_sam(j)$ matches $p\_sam(i)$, we append the value of $j-i$ to $pos[j/h]$. All components of pos are initialized to nil.

Algorithm 4. Main matching algorithm.

for $j := 0$ **to** $(L-1)h$ **step** h **do** $pos[j/h] :=$ nil;
for $j := 0$ **to** $(L-1)h$ **step** h **do begin**
  $b := num(t\_sam(j))$;
  for $i \in t[b]$ **do** append $j-i$ **to** $pos[j/h]$
end;
for $x := 0$ **to** $L-1$ **do begin**
  for $u \in pos[x]$ **do** exhaustive check at u;
  if "success" is reported **then output**(u)
end;
if "success" is never reported **then output**("no match").

The expected number of "append" operations executed in the fourth line is given by

$$\sum_{b=0}^{r^\ell-1} mp^2(b) \le m \cdot r^\ell \cdot (1/r^{2\ell}) = m/M, \text{ since } \sum_{b=0}^{r^\ell-1} p(b) = 1$$

and the left hand side is maximum when all $p(b)$ are equal. Therefore the total time for "append" operations is bounded by $O(kn/M)$.

The expected time $T(j)$ for the exhaustive check at j over text is bounded by Lemma 3 as follows:

$$T(j) = \sum_{d=1}^{k+1} \binom{k+1}{d} m^{-3d} \cdot dm^2 \le O(km^{-1}) \ .$$

From Lemma 4, we can have an algebraic sum of $T(j)$ for the total expected time for exhaustive checks, which is $O(kn/m)$. Now we summarize the complexities as follows:

Algorithm 1:	$O(m)$
Algorithm 2:	$O((n/m)k \log m / \log(1/q))$
Algorithm 3:	$O(m)$
Algorithm 4:	$O(kn/m)$.

Since the complexity of Algorithm 2 is dominant, assuming $\log r \le O(\log m)$, we have the following summary.

Preprocessing the pattern:	$O(m \log r)$
Text analysis:	$O((kn/m) \log m / \log(1/q))$

Keeping the same analysis, we can further relax the above condition of $\log r \leq O(\log m)$ to almost no limitation if we increase the factor 3 in the definition of ℓ in Section 2.

To make the worst case time $O(kn)$, we modify Algorithm 4 slightly. We maintain the last position where an exhaustive check is done in the variable *last*, whose value is initially -1. We do exhaustive check at the candidate position u if $u > last$ for $pat[0..m-1]$ and $text[u..u+2m-1]$ using the algorithm for the k-difference problem in [8] and update *last* to $u+m$. To ensure correctness of this algorithm, all the candidate positions must be sorted in increasing order. This can be done by making the list $cand = pos[0]||pos[1]|| \cdots ||pos[L-1]$, where "||" means concatenation of lists, and sort the list *cand*. This sorting can be done efficiently in the following way. We first divide the list into sections of length $k+1$ and possibly a remaining section and sort the list within the sections. Next merge consecutive even numbered sections and odd numbered sections. That is, merge section 0 and section 1, merge section 2 and section 3 and so on. Lastly merge consecutive odd numbered sections and even numbered sections. That is, merge section 1 and section 2, merge section 3 and section 4, and so on. This sorting method is correct since elements in the list do not move more than $k+1$ positions. This task of sorting will be done in an array rather than in a linked list. The candidate position u is supposed to be taken from this sorted list (or array) in ascending order. The time for sorting is $O((n/m)k \log k)$.

Now exhaustive checks are done at most $[n/m]$ times, each spending $O(km)$ time. Hence the worst case time is $O(kn)$. The average case analysis is basically the same as the above analysis for Algorithm 4.

6 Two Dimensional Matching

In this section we describe briefly how to apply the technique of sampling to the two dimensional approximate pattern matching. Things become two-dimensional in this section. The pattern and text are given by $pat[0..m-1, 0..m-1]$ and $text[0..n-1, 0..n-1]$. We say we position *pat* at (i, j), if we align $pat[0,0]$ with $text[i, j]$, $pat[0, 1]$ with $text[i, j+1]$, $\cdots$, $pat[m-1, m-1]$ with $text[i+m-1, j+m-1]$. If we have no more than k character mismatches with this positioning, we say we have an approximate match. That is, we consider the k-mismatch problem in this section.

Samples are (ℓ, ℓ) square portions of *pat* and *text*. Samples from *pat* are drawn densely, that is, consecutively horizontally and vertically. Samples from *text* are drawn sparsely, that is, at periods of length h horizontally and vertically. The size ℓ and period h are determined by

$$\ell = \left\lceil \sqrt{4 \log m / \log(1/q)} \right\rceil, \quad h = \left\lceil m/\sqrt{k+1} \right\rceil .$$

Similarly to the one dimensional case, we can say that the positioning of *pat* at any (i, j) has at least one sample match if it has an approximate match.

The function num is defined for array $a[0\,..\,\ell-1, 0\,..\,\ell-1]$ of integer type $0\,..\,r-1$ as follows:

$$A[i] = a[i,0]r^{\ell-1} + a[i,1]r^{\ell-2} + \cdots + a[i,\ell-1]\ ,$$
$$num(a) = A[0]R^{\ell-1} + A[1]R^{\ell-2} + \cdots + A[\ell-1], \text{ where } R = r^{\ell}\ .$$

Then the function num is a bijection between r^{ℓ^2} possible array values and integers $0, \cdots, r^{\ell^2}-1$. The function values of num of $p\_pat(i',j')$ and $t\_sam(i,j)$ are defined using the above defined num and conversion of characters to integers $0, \cdots, r-1$. It can be shown that the computing time for $num(p\_sam(i',j'))$ for $i', j' = 0, \cdots, m-\ell$ is $O(m^2)$ and the time for $num(t\_sam(i,j))$ for $i, j = 0, , h, \cdots, (L-1)h$ is $O((n^2/m^2)k \log m/\log(1/q))$, where $L = \lceil n/h \rceil$.

The position table t is one-dimensional. The component $t[b]$ is the list of positions (i',j') such that $b = num(p\_sam(i',j'))$. The size of t is $O((m^4)^{\log r/\log(1/q)})$. Since the number of sample positions (i',j') is not greater than m^2, the construction of t can be optimized using a similar technique to that in Section 4.

To hold the candidate positions for approximate matching, we use two-dimensional array $pos[0\,..\,L-1, 0\,..\,L-1]$. We append $(i-i', j-j')$ to $pos[i/h, j/h]$ if $b = num(t\_sam(i,j))$ and (i',j') is in $t[b]$, for $i, j = 0, h, \cdots, (L-1)h$. Note that array elements $pos[i,j]$ are lists of pairs of integers. The main matching algorithm goes scanning the array pos as in Algorithm 4. The exhaustive check is literally exhaustive check; it compares pat and $text$ character by character until it comes to the end of pat and report "success", or come up with $k+1$ mismatches and report "failure." The probability that the position (i,j) has at least one sample match is given by m^{-4}. From this we have that the expected time for the exhaustive checks is given by $O(n^2/m^2)$. To summarize the computing times, we have that

Time for preprocessing the pattern:	$O(m^2)$
Expected time for matching:	$O((kn^2/m^2)\log m/\log(1/q))$.

The condition $\log r \leq O(k \log m)$ can be relaxed as in the one-dimensional case.

To keep the worst case time to be $O(kn^2)$ for theoretical reasons although the above algorithm is practical and efficient, we modify it in a similar way to that in Section 6. Specifically we modify our exhaustive check for candidate position (u,v) to the algorithm in [1] for the k-mismatch problem, with $pat[0\,..\,m-1, 0\,..\,m-1]$ and $text[base(u)\,..\,base(u)+2m-1\ , base(v)\,..\,base(v)+2m-1]$, where $base(u) = (u \text{ div } m)m$ and 'div" is division with truncation. If we do exhaustive check for (u,v), we can exclude the portion of m^2 area, $text[base(u)\,..\,base(u)+m-1, base(v)\,..\,base(v)+m-1]$ from the text for later exhaustive checks. This exclusion is accomplished by incorporating a two-dimensional Boolean array $done$ whose components are initialized to "no". We do exhaustive check for (u,v) if $done[base(u), base(v)]$ ="no" and let $done[base(u), base(v)]$ ="yes". We need some care at portions of $text$ near the left and bottom ends if $2m$ does not divide n. Note that exhaustive checks are done at most $\lceil n^2/m^2 \rceil$ times, each spending $O(km^2)$ time. The expected time for main matching remains of the same order and the worst case time is kept to be $O(kn^2)$.

7 Concluding Remarks

An interesting aspect of our algorithm is that the dominant part of computing time is that of text sampling, which is not subject to probabilistic fluctuation. That is, the expected time is almost worst case time. The probability that the time of Algorithm 4 exceeds the time for text sampling is very small and will be evaluated by using its standard deviation in the future research.

References

1. Amir, A., Landau, G. M.: Fast parallel and serial multidimensional approximate array matching. Theor. Comput. Sci. **81** (1991) 97–115
2. Baeza-Yates, R. A., Perleberg, C. H.: Fast and practical approximate string matching. Proc. 3rd Ann. Symp. on Combinatorial Pattern Matching (1992) 185–192
3. Baker, T. P.: A technique for extending rapid exact-match string matching to arrays of more than one dimension. SIAM J. Comput. **7** (1978) 533–541
4. Bird, R. S.: Two dimensional pattern matching. Info. Proc. Lett. **6** (1977) 168–170
5. Boyer, R. S., Moore, J. S.: A fast string searching algorithm. CACM **20** (1977), 767–772
6. Chang, W. I., Lawler,E. L.: Approximate string matching in sublinear expected time. FOCS **31** (1990) 116–124
7. Erickson, B. W., Sellers, P. H.: Recognition of patterns in genetic sequences, in Sankoff and Kruskal (eds.), Time warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley, 55–91 (1983)
8. Galil, Z., Park, K.: An improved algorithm for approximate string matching. ICALP'89, LNCS **372** (1989) 394–404
9. Galil, Z., Park, K.: Truly alphabet-independent two-dimensional pattern matching. FOCS **33** (1992) 247–256
10. Karp, R. M., Rabin, M. O.: Efficient randomized pattern-matching algorithms. IBM J. of Res. and Dev. **31** (1987) 249–260
11. Knuth, D. E., Morris, J. H., Pratt, V. R.: Fast pattern matching in strings. SIAM J. Comput. **6** (1977) 323–350
12. Landau, G. M., Vishkin, U.: Efficient string matching with k mismatches. Theor. Comput. Sci. (1985) 239–249
13. Landau, G. M., Vishkin, U. Fast string matching with k differences. JCSS **37** (1988) 63–78
14. Ukkonen, E.: Algorithms for approximate string matching. Information and Control **64** (1985) 100–118
15. Ukkonen, E.: Finding approximate patterns in strings. J. Algorithms **6** (1985) 132–137
16. Vishkin, U.: Deterministic sampling—A new technique for fast pattern matching. STOC **22** (1990) 170–180
17. Wu, S., Manber, U.: Fast text searching with errors. Tech. Rep. TR-91-11, Dept. of Comp. Sci., Univ. of Arizona (1991)
18. Zhu, R. F., Takaoka, T.: A technique for two-dimensional pattern matching. CACM **32** (1989) 1110–1120

This article was processed using the LaTeX macro package with LLNCS style