Theoretical Computer Science 92 (1992) 165–179
Elsevier

Fast average-case pattern matching by multiplexing sparse tables

Russell W. Quong
School of Electrical Engineering, Purdue University at West Lafayette, USA

Abstract

Quong, R.W., Fast average-case pattern matching by multiplexing sparse tables, Theoretical Computer Science 92 (1992) 165–179.

Pattern matching consists of finding occurrences of a *pattern* in some data. One general approach is to sample the data collecting evidence about possible matches. By sampling appropriately, we force matches to be sparse and can encode a table of size m as a series of smaller tables with total size $\Theta((\ln m)^2/\ln\ln m)$. This method yields practical algorithms with fast average-case running times for a wide variety of pattern matching and pattern recognition problems.

We apply our technique of multiplexing sparse tables to the k-mismatches string searching problem which asks for all occurrences of a pattern string $P=p_0, p_1, \ldots, p_{m-1}$, in a text string $T=t_0, t_1, \ldots, t_{n-1}$, with $\leqslant k$ mismatches (substitutions), where P, T and k are given. Assuming a uniform character distribution over an alphabet of size A, for $k \leqslant m/(2\log_A m)$, our algorithm has an *average-case* running time of $\Theta(kn(\log m)^2/(m\log\log m))$ and uses $\Theta(m(\log m)^2/(\log\log m))$ space.

1. Introduction

The term "pattern matching" encompasses a wide variety of problems, including two broad classes: (1) *Pattern recognition*: given a *dictionary* D of m known patterns, and data X, find the entries in D that match X; (2) *Pattern Searching*: given a fixed pattern P of size m, and *text* data T of size n, find all occurrences of P in T. Pattern recognition problems commonly arises in spelling correctors, speech recognition and machine vision. Pattern searching arises in problems such as string search, keyword search and DNA sequence analysis.

We present a general, practical algorithmic method applicable to many of these problems. In this paper, our focus is on fast *average-case* pattern searching. In particular, we illustrate our method in an algorithm for the k-mismatches string searching problem.

0304-3975/92/$05.00 © 1992—Elsevier Science Publishers B.V. All rights reserved

1.1. Background

In searching a text string of n characters for all occurrences of a fixed pattern of m characters, the linear time algorithms of Aho and Corasick [2] and Knuth, Morris, and Pratt [12] use a finite automaton with a failure function. Boyer and Moore [5] give a fast, practical algorithm with a sublinear average-case running time of roughly $\Theta(n/m)$ for small m. For large m, Knuth et al. [12] give a variant of the Boyer–Moore algorithm with an average case running time of $O(n(\log_A m)/m)$, where A = the alphabet size. Yao [17] shows that this bound was optimal. For two-dimensional matching, Zhu and Takaoka [18] have an $O(n^2(\log m_2)/m_2)$ average case algorithm, where the pattern is an $m_1 \times m_2$ array and text is an $n \times n$ array. Baeza-Yates and Régneir [4] improve this result to $\Theta(n^2/m)$ time.

If the pattern string and text might have wildcard characters, Fischer and Paterson [8] give an $O(n \log m \log\log m)$ based on multiplication. Depending on the character distribution in the pattern, Abrahamson [1] dynamically chooses the better of two algorithms for a $\Theta(n\sqrt{m}\,\text{polylog}(m))$ worst case solution for "generalized string matching" where each position of the pattern can match an arbitrary set of characters.

The k-mismatches problem allows for $\leqslant k$ mismatches but not for insertions or deletions between the pattern and text. That is a match occurs when the pattern and a text substring have a Hamming distance $\leqslant k$. Abrahamson's approach works on this problem also. Atallah [3] has mentioned a fast $\Theta(n \log m)$ probabilistic estimator which works well if there is a statistically significant match between the pattern and text.

In the k-differences problem, we allow $\leqslant k$ differences (mismatches, insertions or deletions) to occur between the pattern and text. By finding least common ancestors in a suffix tree [16, 14] of the pattern and text, Landau and Vishkin [13] solve the k-differences problem in $\Theta(kn)$ time in the worst (and average) case. Several variations have since been devised [9, 15]. Chang and Lawler [6] use a suffix tree of the pattern achieving a sublinear average-case running time of $\Theta(kn(\log m)/m)$ when $k \leqslant (m/(c_1 + \log m)) - c_2$. This last algorithm gives strictly better asymptotic results than our method (and for a harder problem). However, our method is of interest because (i) it uses a different approach, giving an upper bound on how well every alignment matches the text, and (ii) it is applicable to other types of pattern matching.

1.2. Overview

Our approach consists of (i) sampling the (text) data forcing matches to be sparse and (ii) multiplexing a table of partial matches of size m into a series of smaller *subtables* of total size $\Theta((\ln m)^2/\ln\ln m)$. We can update and check the subtables quickly, ruling out almost all mismatches and identifying potential matches, which then must be explicitly checked. This approach gives practical algorithms with good average-case running times. In the worst case, our technique provides no information, however, this probability is small. Our method applies naturally to a wide variety

of problems including multi-dimensional matching, generalized string matching, searching for multiple keywords, and matching with mismatches.

In the next section, we state our definitions and assumptions about the problem. Next, we give a naive algorithm which forms the basis for our approach. We present our method in Section 4. We then give an example of exact string matching using our approach in Section 5. In Section 6 we show how to multiplex a large table as a series of smaller tables. In Section 7 we analyze the probability of incorrect potential matches. In Section 8 we briefly describe modifications for different pattern searching problems and outline a few of the possible variations to our method. In Section 9 we consider implementation details and provide some preliminary data comparing our algorithm with previous algorithms. Finally, we calculate the sample size in Appendix A.

2. Definitions and assumptions

Let the *pattern* $P = p_0 p_1 \ldots p_{m-1}$ and the *text*, $T = t_0 t_1 \ldots t_{n-1}$ be strings from a finite alphabet Σ of size $A = |\Sigma|$. We assume all t_i and p_j can be sampled in constant time. Characters in T and P are uniformly and independently distributed so that averaged over all p_i and t_j, $\Pr(p_i = t_j) = 1/A$.

Let $T[j:l] \equiv t_j \ldots t_{j+l-1}$ = the substring of T of length l starting at t_j. Alignment j of P refers to comparing P versus $T[j:m]$. Alignment i matches a text substring $T[j:l]$ if $P[j-i:l] = T[j:l]$. We rule out alignment i if we determine that P cannot match $T[i:m]$. We assume a weak unit cost RAM model of computation in which all arithmetic for numbers of size $\leqslant m^2$ can be done in constant time. The notation $\log^j x$ means $(\log x)^j$, and $\ln x$ means $\log_e x$.

The algorithms in this paper operate on a sliding window of size m into the text and sample the window backwards from right to left. We note that an algorithm that slides the window by at least s positions and requires time $\Theta(t_{\text{win}})$ between slides requires $O(nt_{\text{win}}/s)$ time to search T. For the rest of this paper, $T(=t_0 \ldots t_{m-1})$ refers to the text in this window; all alignments are relative to this window. (See Fig. 1.)

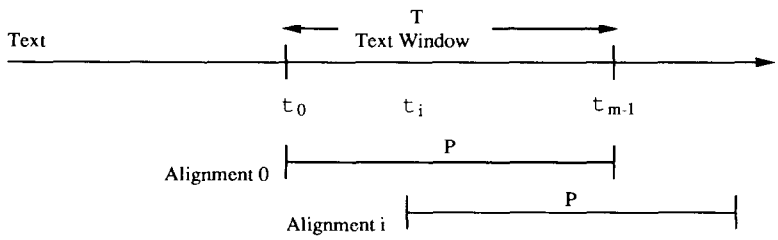

Fig. 1. Sliding window of size m.

3. A naive algorithm

A general approach to pattern searching is to keep a value on how well each alignment has matched the sampled text. We can use a table $\Lambda[0 \ldots m-1]$ of size m with $\Lambda[i]$ containing the number of matches for alignment i. When we are finished sampling, the alignment with the largest value is the best match. Updating $\Lambda[0 \ldots m-1]$ requires $\Theta(m)$ time.

For string matching, we sample the last N characters of T. For $t = t_{m-1}$, $t_{m-2}, \ldots, t_{m-N}$, we increment $\Lambda[i]$ if alignment i matches t. Alignment i *survives* if $\Lambda[i] \geqslant \tau$, where τ is a predetermined threshold. For the k-mismatches problem, we choose $N > k$ and, hence, $\tau = N - k$. If alignment i survives past the sampling, we check if the rest of P actually matches $T[i:m]$ with $\leqslant k$ mismatches. Because $\Pr(p_i = t_j) = 1/A$ for random i, j, after N samples, $E[\Lambda[i]] = N/A$. If alignment i survives the sampling but is not a true match, we call alignment i a *false survivor.*

For $A \geqslant 2$, setting $N = (2k + 4 \ln m + 4\sqrt{k \ln m + \ln^2 m})$ (see Appendix A) reduces the probability of a false survivor to $o(1/m^2)$, so that checking a false survivor does not affect the average-case running time. We can rule out alignments $0, 1, \ldots, m-N$ and can slide the text window past these alignments. We show that the running time is $\Theta(n \max(\log m, k))$ for $k \leqslant m/3$. For $k \leqslant \ln m$, we have $N \leqslant (6 + 4\sqrt{2})(\ln m) < 12 \ln m$ so that for large m, we sample $\Theta(\log m)$ characters between slides and we slide T by at least $(m-N) = \Theta(m)$ positions. For $\ln m < k \leqslant m/3$, and large enough m, we have $2k \leqslant N < 9k/4 \leqslant 3m/4$, which gives a minimum slide of $m - N \geqslant m - 3m/4 \geqslant m/4$. The time t_{win} between slides is mN giving a running time of $\Theta(Nn) = \Theta(n \max(k, \ln m))$ for $k \leqslant m/3$.

4. Multiplexing sparse tables

Our approach modifies the naive approach in two ways.

- Sample the text in blocks of size $b \equiv \lceil \log_A m \rceil$, so that matches between blocks are sparse. We give *points* to alignment i depending on how well it matches the sampled block. For the rest of this paper, we give one point to alignment i if and only if it completely matches the sampled block.
- Multiplex $\Lambda[\ ]$ as a series of smaller *subtables* $\Lambda_1[\ ], \Lambda_2[\ ], \ldots, \Lambda_\xi[\ ]$, where ξ is a function of m. Let $\lambda_j \equiv |\Lambda_j[\ ]| =$ the size of $\Lambda_j[\ ]$. We require $(\prod_{j=1}^{\xi} \lambda_j) \geqslant m$, and that all λ_j are mutually prime to one another. The points for alignment i are stored in $\Lambda_j[i \bmod \lambda_j]$ for all subtables $\Lambda_j[\ ]$. We note that $\Lambda_j[i \bmod \lambda_j] \geqslant \Lambda[i]$ (the naive table) for all i, j.

Define $\lambda_{\text{SUM}} = \sum_{j=1}^{\xi} \lambda_j$, and $\lambda_{\text{PROD}} = \prod_{j=1}^{\xi} \lambda_j$. For example, if $m = 300$ and $A = 26$, then the blocksize $b = 2$, and we might choose 3 subtables of sizes 7, 8, 9 so that $\xi = 3$, $\lambda_1 = 7$, $\lambda_2 = 8$, $\lambda_3 = 9$, $\lambda_{\text{SUM}} = 24$, and $\lambda_{\text{PROD}} = 504$. The points for alignment 123 are stored in $\Lambda_1[4(123 \bmod 7)]$, $\Lambda_2[3(123 \bmod 8)]$ and $\Lambda_3[6(123 \bmod 9)]$. $\Lambda_1[0]$ contains the points for alignments $0, 7, 14, \ldots, 294$.

Claim 1. *On the average, less than one alignment receives a point per sample.*

Proof. Let β represent a block of b text characters, and let $X_i(\beta)$ and X_P be the random variables (r.v.) representing the number of points alignment i and the entire pattern P, respectively, receives on the sample $\beta = T[m-b:b]$. There are $A^b \geqslant m$ different possible blocks; there are $m-b+1$ different alignments each of which matches one block, so that

$$\mathbf{E}[X_P] = \frac{1}{A^b} \sum_{\beta} \sum_{i=0}^{m-b} X_i(\beta) = \frac{m-b+1}{A^b} < \frac{m-b+1}{m} < 1. \qquad \square$$

Claim 2. *Each alignment i occupies a unique set of subtable entries.*

Proof. By the Chinese remainder theorem, as $\lambda_{\text{PROD}} > m$ and all λ_j are mutually prime, there can be only one alignment i that satisfies $i \equiv i_j \bmod \lambda_j$ for a particular set of subtable indices $(i_1, \ldots, i_\xi)$. $\square$

As will be shown in Section 4, setting $\lambda_j =$ the jth prime gives asymptotic values of $\lambda_{\text{SUM}} \approx (\ln^2 m)/(2 \ln \ln m)$, and (number of subtables) $\xi \approx \ln m/(\ln \ln m)$. If alignment i matches, we must have $\Lambda[i] \geqslant \tau$ and, hence, $\Lambda_j[i \bmod \lambda_j] \geqslant \tau$ for all $j = 1, \ldots, \xi$. After N samples $\mathbf{E}[\Lambda_j[i]] = N(m-b+1)/(A^b \lambda_j) < (N/\lambda_j)$, which grows as N/λ_j. But $\tau = N-k$, so that if we sample enough blocks, τ grows as N and $\tau \gg \mathbf{E}[\Lambda_j[i]]$, reducing the chance of a false survivor.

Our algorithm is as follows, given T, P, k and m.

1. Determine the subtable sizes λ_j and ξ, the number of subtables. (From a practical standpoint, start the subtable sizes at 5 or larger as explained at the end of Appendix A.) Set $N = 1.25k + 2 \ln \lambda + 3\sqrt{k \ln \lambda + \ln^2 \lambda}$, for $\lambda = \max(\lambda_j)$.
2. Precompute the tables $\Lambda\Lambda_j[i][\beta]$ which indicates how to update entry $\Lambda_j[i]$ of subtable j upon finding the sample $\beta = T[m-b:b]$, for all j, i, β. Our assumption that the sample β is the last b characters in T will be corrected as we sample leftward into T. (For our present scheme of giving alignment i one point iff it matches β, increment $\Lambda\Lambda_j[i \bmod \lambda_j][P[m-i-b:b]]$, for $1 \leqslant j \leqslant \xi$, and $0 \leqslant i < m - bN$. This phase requires $\Theta((m-N)\xi) = \Theta(m \ln m/\ln \ln m)$ time.)
3. Initialize the subtables, by setting all entries $\Lambda_j[i] = 0$, for all i, j. The variables $\max_j$ will represent the maximum value of any entry in $\Lambda_j[\ ]$. Set all $\max_j$ to 0.
4. Sample N adjacent blocks of b characters from the rightmost end of T. For the qth block, $\beta_q = T[m-(q+1)b:b]$, $(q = 0, 1, \ldots, N-1)$, increment $\Lambda_j[i]$ by the value $\Lambda\Lambda_j[(i-qb) \bmod \lambda_j][\beta_q]$. The term $(i-qb) \bmod \lambda_j$ reflects the fact that the qth block is qb positions from the end of T, whereas we precomputed $\Lambda\Lambda_j[i][\beta_q]$ to assume the sample was at the end of the text. In essence, we "left rotate" the entries of $\Lambda\Lambda_j[\ ][\ ]$ by $-qb$ positions before updating $\Lambda_j[\ ]$.
5. Check if every subtable $\Lambda_j[\ ]$ has at least one entry with $\geqslant \tau$ points. If "yes", determine the corresponding alignments (see [11, Ch. 4.3.2]) and check that they

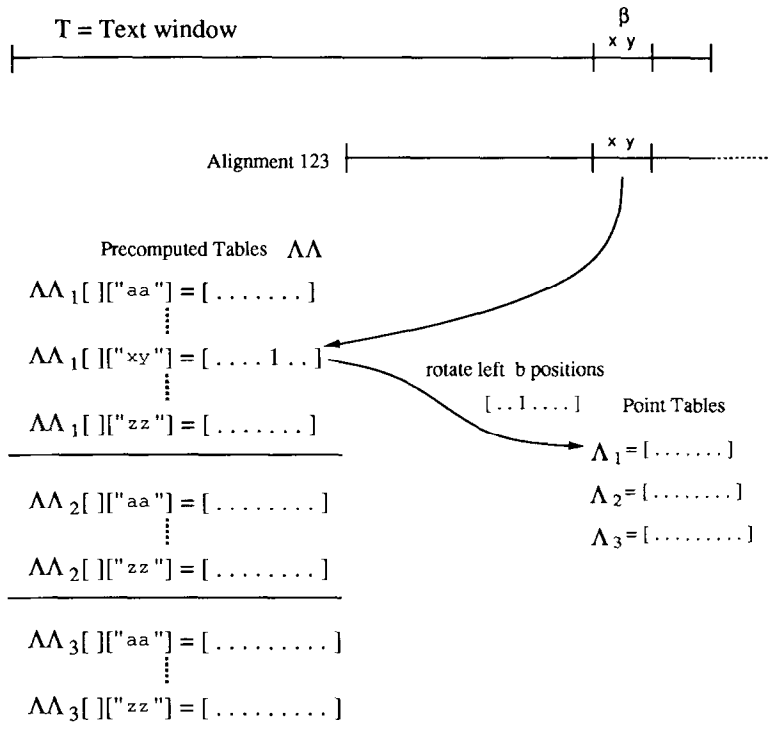

Fig. 2. Updating the point table (alignment 123 matches 2nd to last block).

actually match T. Print the match if one exists. In either case, slide T the text window $(m-Nb)$ positions to the right.

Figure 2 illustrates the algorithm for the previous example for $A=26$ (english letters), $m=300$, $b=2$, $\xi=3$, $\lambda_1=7$, $\lambda_2=8$, $\lambda_3=9$, when alignment 123 matches the sampled block $\beta=$"xy". Figure 2 shows $\Lambda\Lambda_1[4][xy]=1$. As $\beta=$the second-to-last block, we left rotate $\Lambda\Lambda_1[\ldots][xy]$ by 2 positions before adding it to $\Lambda_1[\ldots]$. Figure 3 gives pseudo-code for the last three steps of our algorithm.

The update tables $\Lambda\Lambda_j[\ ][\ ]$ require $\Theta(m\log^2 m/\log\log m)$ space in total and require $\Theta(m\xi)=\Theta(m\log m/(\log\log m))$ time to precompute. As described in [11, p. 270], we can determine i from its subtable indices $(i_1,\ldots,i_\xi)$ (step 9 of Fig. 3) by using a precomputed table of ξ numbers M_j, where $M_j\equiv 1 \bmod \lambda_j$ and $M_j\equiv 0 \bmod \lambda_{j'}$ for all $j'\neq j$. If the surviving entry in table $\Lambda_j[\ ]$ is i_j, then $i=(\sum_{j=1}^{\xi} M_j i_j) \bmod \lambda_{\text{PROD}}$. Computing i from $(i_1,\ldots,i_\xi)$ requires $O(\xi)$ time and precomputing the M_j entries requires $o(m)$ time.

The running time of our algorithm is $\Theta(n\lambda_{\text{SUM}}/m)$. Each sample is of size b, which requires $\Theta(\log_A m)$ time to read. For each sample, we update λ_{SUM} subtable entries requiring $\Theta(\lambda_{\text{SUM}})$ time. After sampling we slide T by $n-bN$ positions. Appendix A shows that for large enough k, m and if $k\gg\ln m$, we

```
    SparseMult (T, P)        // checks if alignments 0...m − Nb match
1     set Λ_j[ ] = max_j = 0, for all j = 0,...,ξ
2     for q = 0 to N − 1
3       β = T[m − (q + 1)b : b]
4       for each subtable Λ_j[ ]
5         for 0 ≤ i < λ_j
6           increment Λ_j[(i_y − qb) mod λ_j] by ΛΛ_j[i][β]
7           max_j = max(max_j, Λ_j[(i_y − qb) mod λ_j]);
          end
        end
      end
8     if (max_j ≥ τ) for all j = 0,...,ξ
9       determine the matching alignment i        // survivor
10      check if an actual match by comparing P and T[i : m]
11        print i if match
      end
12    slide the text window T to the right by m − Nb characters
```

Fig. 3. Algorithm for updating the subtables.

have $N=1.25k+O(\sqrt{k}\ln\ln m)<4k/3$. For $k<m/(2b)=m/(2\log_A m)$, we have $N<2m/(3\log_A m)$, and we can slide T by $T-bN>m/3$ positions. Thus, $t_{\text{win}}=\Theta(N\lambda_{\text{SUM}})=\Theta(k\log^2 m/(\log\log m))$ and we slide by $\Theta(m)$, for a total running time of $\Theta(nk\log^2 m/(m\log\log m))$.

5. An example

We give an explicit example where $k=0$ for simplicity. Although having $k=0$ results in the exact string matching problem, the preceding ideas of our method still apply. We note that there are better algorithms for this particular problem. Let $m=24$, and

$$P=000\quad 001\quad 010\quad \underline{011}\quad \underline{100}\quad 10\underline{1}\quad \underline{110}\quad 111$$

so that P is the concatenation of the binary numbers 0 to 7. The spacing in P is for clarity alone. We shall use $N=2$, $b=4$, and two subtables of sizes $\lambda_1=5$ and $\lambda_2=6$. Then $\tau\equiv N-k=2$. Assume we sample $T[20:4]$ and find $\beta=$"1110". There are matches at alignments 3 and 10, underlined above in P. Thus, $\Lambda\Lambda_1[3][1110]=\Lambda\Lambda_2[3][1110]=1$, and $\Lambda\Lambda_1[10 \bmod 5][1110]=\Lambda\Lambda_2[10 \bmod 6][1110]=1$. All other entries in $\Lambda\Lambda_{1,2}[\ ][1110]$ are 0. For alignment 3, we give a point to $\Lambda_1[3 \bmod 5]$ and $\Lambda_2[3 \bmod 6]$; for alignment 10, we give a point to $\Lambda_1[10 \bmod 5]$ and $\Lambda_2[10 \bmod 6]$. (See Fig. 4.)

Thus, $\Lambda_1[\ ]=[10010]$ and $\Lambda_2[\ ]=[000110]$. Assume, we next find $\beta=$"1011"$=T[16:4]$. If "1011" was the last block in T alignments 1 and 5 would be

```
                0123 4567 890  12   16   20     subtable 1      subtable 2
          T = **** **** **** **** **** 1110   0 1 2 3 4      0 1 2 3 4 5
Align. (3) =    0 0000 1010 0111 0010 1110        1              1
Align.(10) =             00 0001 0100 1110   1                     1
```

Fig. 4.

```
          T = **** **** **** **** **** 1011   0 1 2 3 4      0 1 2 3 4 5
Align. (1) =  000 0010 1001 1100 1011 1011      1              1
Align. (5) =       000 0010 1001 1100 1011   1                       1
```

Fig. 5.

```
          T = **** **** **** **** 1011 ****  0 1 2 3 4      0 1 2 3 4 5
Align.(-3) = 0010 1001 1100 1011 1011 1          1              1
Align. (1) =  000 0010 1001 1100 1011 1011     1              1
```

Fig. 6.

matches as shown in Fig. 5. (Remember, we precompute $\Lambda\Lambda[\ ][\beta]$ as if the β were the last block in T, $T[20:4]$.)

However, "1011" is actually the second to last block, so that we must adjust all alignments by $-b$ or -4, which corresponds to a cyclic left shift of 4 positions in each table. (See Fig. 6.)

Thus, the points added to $\Lambda_1[\ ]$ and $\Lambda_2[\ ]$ are [01100] and [010100] for $T[16:4]$="1011". The resulting subtables are [11110] and [010210]. Because $\Lambda_1[\ ]$ has no entries $\geqslant 2 \equiv \tau$, we know there are no matches among alignments 0–15. In this example, rather than update a table of size 16 we "only" had to update 11 entries.

6. Choosing subtable sizes

Determining the subtable sizes reduces to the problem of finding mutually prime numbers $\lambda_1, \ldots, \lambda_\xi$ whose product exceeds m and whose sum is minimized. In this section, let $p_i \equiv$ the ith prime. We note that $x+y<xy$ for $2 \leqslant x<y$, so that if $\lambda_i = xy$ where x and y are mutually prime, splitting λ_i into two tables of size x and y decreases λ_{SUM}. It follows that each λ_i should be of the form p^r for some prime p and $r \geqslant 1$.

In the continuous version of this problem, where each λ_i can be any real number, setting $\lambda_i = \sqrt[\xi]{N}$ minimizes λ_{SUM}. In particular $\xi = \ln n$ and $\lambda_i = e$ is optimal over all ξ showing that small λ_i are best. In our problem, we cannot choose ξ too small

otherwise there will not be enough primes $\leqslant \sqrt[\xi]{N}$ to use. We believe setting $\lambda_i = p_i$ is asymptotically optimal. Using $p_i \approx i(\ln i + \ln\ln i)$ [10] we solve for ξ.

$$\ln \lambda_{\text{PROD}} = \sum_{i=1}^{\xi} \ln[i(\ln i + \ln\ln i)] \approx \int_1^{\xi} \ln x + \ln(\ln x + \ln\ln x)\,dx.$$

Making the change of variable $\ln x = y$, integrating by parts, and taking the dominant terms gives

$$\ln \lambda_{\text{PROD}} \approx x(\ln x - 1) + x\ln(\ln x + \ln\ln x)|^{\xi}$$

$$= \xi(\ln \xi + \ln(\ln \xi + \ln\ln \xi) - 1).$$

For $\ln \lambda_{\text{PROD}} = \ln m$, we get $\xi \approx (\ln m)/(\ln\ln m)$, and $p_{\xi} \approx \ln m$. Then,

$$\lambda_{\text{SUM}} = \sum_{i=1}^{\xi} \lambda_i \approx \sum_{i=1}^{\xi} i(\ln i + \ln\ln i) < \sum_{i=1}^{\xi} i(\ln \xi + \ln\ln \xi)$$

$$\approx \frac{\xi^2(\ln \xi + \ln\ln \xi))}{2} \approx \frac{(\ln^2 m)/(\ln\ln m)}{2}.$$

7. False survivors

Because we multiplex points for many alignments in a single subtable entry, false survivors are possible. In a subtable of size λ, we force the probability that an entry falsely survives to be $o(1/\lambda^3)$. In Appendix A, we show that choosing $N \geqslant (1.25k + 2\ln\lambda + 3\sqrt{k\ln\lambda + \ln^2\lambda})$ meets this criterion (choose N by plugging in the largest λ).

For any subtable of size λ, the probability of no false survivors in the table is $\Pr(\text{FS}=0) = (1-\lambda^{-3})^{\lambda} > 1-\lambda^{-2}$. The probability of a false survivor is $\Pr(\text{FS} \geqslant 1) = 1 - \Pr(\text{FS}=0) < \lambda^{-2}$. The probability of at least one false survivor in every subtable $A_j[\ ]$ of size λ_j is $\prod \Pr(\text{FS in } A_j[\ ]) < \prod(\lambda_j^{-2}) \leqslant 1/m^2$, as we have $\lambda_{\text{PROD}} = \prod(\lambda_j) \geqslant m$. Thus, the probability of a nonmatching alignment surviving all subtables is $o(1/m^2)$. A full check on this alignment requires $\Theta(m)$ time and adds a negligible factor of $o(1/m)$ to the total running time.

If the pattern P or a large portion of P is periodic with a small period u, our method can fail badly if there are matches, because many alignments might match the sampled blocks. In this case it is best to modify the point tables so that only the first u alignments for the first period receive points. If an alignment i survives, all other alignments $i+ku$ are known to survive.

8. Variations

We outline how to extend our technique for different variations of pattern finding. These extensions are mutually independent of one another and can be applied simultaneously.

Our method handles context sensitive mismatches, where the cost of a mismatch depends on the mismatch itself and possibly the surrounding data. The choice of b, and the how to update alignment i on sampling β will depend on the cost function.

In *generalized string matching* [1], where $P = p_1 \ldots p_m$, each *pattern element* p_i in P can match an arbitrary set of characters. For example, the pattern element [+abc] matches a, b or c, and the pattern element [−abc] matches any character but a, b or c. The effective alphabet size A' is $1/\Pr(p_i \sim t_j)$, where $p_i \sim t_j$ means p_i matches t_j. Use $b = \log_{A'} m$ which increases the space usage by $\Theta(m^\gamma)$, where $\gamma = \log_A A'$.

If *transpositions* are considered a single mismatch, sample the N blocks of the text $T[m-b:b], \ldots, T[m-Nb:b]$ as before, and then sample the same text, but shift the blocks by $b/2$ positions, for $T[m-b-b/2:b], \ldots, T[m-Nb-b/2:b]$. By appropriate precomputation of the subtables, a transposition that occurs inside a block counts as one mismatch. A transposition that occurs at the seam between two blocks, will be compensated for in the overlapping block.

If the pattern is *two-dimensional* (m_r rows by m_c columns) number the alignments column-by-column from the bottom left corner. Let $m = (m_r m_c)$ and $b = \log_A m$. Form a block by sampling the top b characters in a column. Sample a block in each of the rightmost N columns. Alignments for block q must be adjusted by qm_r positions. We slide the T text window row by row from right-to-left moving upwards by $\Theta(m_r)$ rows at the end of the row scan. For exact matching, we get an average-case running time of $\Theta(n(\log_A m)^2/m)$. This technique can be extended for arbitrary dimensions.

For a *set of patterns* $P = \{P_0, \ldots, P_v\}$, let $m_i = |P_i|$, and $m_- = \min_i(P_i)$, the length of the shortest pattern. Give the first m_- alignments for each P_i a unique id. Use these id's in place of the alignments. For the k-mismatches problem we get a running time of $\Theta(kn \log_A(vm_-)/m_-)$. This approach is practical only if m_- is large.

8.1. Alternative strategies

Point schemes: For the k-mismatches problem, we have used a very simple point scheme of 1 point on a complete match. Clearly, other schemes exist. We note that limiting the precomputed table entries $\Lambda\Lambda_j[i][\beta]$ to 1 further reduces the probability of a false survivor, without affecting the correctness of the algorithm.

Other multiplexing schemes: Our current multiplexing scheme stores points for alignment i in $\Lambda_j[i \bmod \lambda_j]$. In the event of more than one simultaneous matches, each subtable has two survivors. There is no method to determine which set of entries belong to a given alignment, so that many alignments must be checked via trial and error. For example, if two alignments match, in the worst case $O(2^\xi) = O(m^{1/\log\log m})$ alignments would have to be explicitly checked. (With our current scheme, if more

than one alignment survives, it is best to continue sampling for, say, another $\approx\Theta(N)$ blocks to hopefully rule out some or all the survivors.)

One method to alleviate the problem of multiple survivors is to rearrange the alignments, so that similar alignments occupy the same entries on many subtables. Let π be a permutation of $(0, 1, \ldots, \lambda_{\text{PROD}})$. We try to choose π such that if alignments i_1 and i_2 are similar, then $\pi(i_1) \equiv \pi(i_2) \bmod \lambda_j$ for as many λ_j as possible. We leave computing π as an open question.

9. Practical aspects

If a subtable λ_j fits in a single computer word, we can save space and time by storing the entries in parallel bit by bit in each a word as shown in Fig. 7. For example, the least significant bit (LSB) for entry i would be bit i in word 0; in general, bit bb for entry i would be the ith bit in word bb. In Fig. 7, $\Lambda_j[0]$ contains the value $3 = 011_2$; note only three words are needed to contain $\Lambda_j[\ldots]$ compared with the normal λ_j words. We update the table in parallel via logical bit operations. Updating the subtable requires $\Theta(\lg k)$ time, as we need $\Theta(\lg k)$ bits per entry, where $\lg x \equiv \log_2 x$.

In addition, each update table $\Lambda\Lambda_j[\ldots][\beta]$ can be stored in a single computer word. To rotate the update tables, use a bit rotate instruction. The total space needed is $\Theta(A(\lg k)(\ln m)/(\ln\ln m))$ and the running time becomes $\Theta(kn \lg k \log m/(m \log\log m))$. On a 32 bit machine, using subtables of sizes 32, 31, 29, 28, 27, 26, 25, 23, 19, 17, and 11 allows for $m \leqslant 100 \times 10^{12}$, which should accommodate most problems.

For $m = O(10{,}000)$, the overhead of doing bit shifts on each addition is probably slower than simply using three or four mid-sized tables. Table 1 gives execution times of several different string searching algorithms on a SUN SPARCstation 1 (a 10–15 MIP machine) running Sun OS 4.0. In all cases, $n = 250{,}000$, $A = 2$, and the text did not contain the pattern. All running times are in seconds and include file I/O time, preprocessing time and search times. We compiled our program, **sparseMult**, with version 1.37 of the GNU C++ compiler.

For comparison, **egrep** a fast regular expression searching program using a deterministic finite automaton, required 0.4 seconds when $m = 378$. An implementation of the Boyer–Moore algorithm, **BM**, required 0.3 seconds on the same search. We note that **BM** is not particularly well suited for searches of long binary patterns. **R-L-naive**

		LSB	...	7th bit
LS bits	word 0	1 ($\Lambda_j[0]$)	...	$\Lambda_j[6]$
2nd LS bits	word 1	1 ($\Lambda_j[0]$)	...	$\Lambda_j[6]$
3rd LS bits	word 2	0 ($\Lambda_j[0]$)	...	$\Lambda_j[6]$

Fig. 7. If subtable size $\lambda_j \leqslant$ word size, map bits in parallel (here $\lambda_j = 7$).

Table 1

Algorithm	Table size	n	m	k	time (sec)
egrep		250 K	378	0	0.4
Boyer–Moore		250 K	378	0	0.3
Boyer–Moore		250 K	378	0	0.3
R-L-naive		250 K	378	0	3.4
R-L-naive		250 K	378	5	6.2
R-L-naive		250 K	378	15	12.7
sparseMult	(11, 7, 5)	250 K	378	0	0.2
sparseMult	(13, 11, 9)	250 K	378	0	0.3
sparseMult	(11, 7, 5)	250 K	300	5	0.6
sparseMult	(13, 11, 9)	250 K	300	15	2.4
sparseMult	(11, 7, 5)	250 K	300	15	1.8
sparseMult	(13, 11, 9)	250 K	600	5	0.4
sparseMult	(13, 11, 9)	250 K	600	15	0.8
sparseMult	(13, 11, 9)	250 K	600	30	2.3
sparseMult	(11, 7, 5, 2)	250 K	600	30	2.0

is the algorithm from Section 3 that samples individual text characters from right to left ala BM. The above table shows that in all cases, **sparseMult** compared quite favorably with the other programs for both exact and k-mismatches string matching.

10. Concluding remarks

We have illustrated our method of multiplexing sparse matches in a series of small subtables for the k-mismatches problem. For an alignment i, the minimum value in its subtable entries gives an upper bound on how well i matches the sampled text. By modifying the sampling strategy or the point scheme, our technique is applicable to other types of pattern searching and pattern recognition problems. We close by asking for further refinements in sampling strategies, point schemes, and multiplexing methods.

Appendix A (Choosing N)

Naive algorithm

Let $X = \sum_{j=1}^{N} X_j$, where X_j is the r.v. for the number of points given to alignment x on the jth sample. $X_j = 1$ if alignment x matches the ith sample, and is 0 otherwise. Thus, X represents the number of points in $A[x]$ after N samples. For an alphabet size of A, we have a mean of $\mu = \mathbf{E}[X_j] = 1/A$, and a variance of $\sigma^2 = (A-1)/A^2$. In this section, we derive N such that the probability of a false survivor $= \Pr(X \geqslant \tau) \leqslant (1/\varepsilon)$, where $\varepsilon \equiv 1/\Pr(\text{false survivor})$.

As the X_j are i.i.d random variables, we apply Chernoff bound methods for a series of N Bernoulli trials [7]

$$\Pr(X - N\mu \geqslant r) \leqslant \exp\left(\frac{-r^2}{4N\sigma^2}\right) \leqslant 1/\varepsilon.$$

We want $\Pr(X \geqslant \tau \equiv N-k)$; thus, $r = \tau - N\mu = ((A-1)N/A - k)$, giving the exponent $r^2/(4N\sigma^2) = ((A-1)N-k)^2/(4N(A-1))$. Taking the natural logarithm of both sides gives

$$((A-1)N - Ak)^2 \geqslant 4(A-1)N \ln \varepsilon.$$

Expanding the left side gives a quadratic equation in N,

$$(A-1)^2 N^2 - (A-1)(2kA + 4\ln\varepsilon)N + A^2k^2 \geqslant 0.$$

Solving for N gives

$$N \geqslant \frac{Ak}{A-1} + \frac{2\ln\varepsilon}{A-1} + \frac{2\sqrt{kA\ln\varepsilon + \ln^2\varepsilon}}{A-1}.$$

As $A \geqslant 2$, making the substitutions $2 \geqslant A/(A-1)$, $1 > 1/(A-1)$, and $4 \geqslant \sqrt{A}/(A-1)$, and setting $\varepsilon = m^2$ gives the result $N = (2k + 4\ln m + 4\sqrt{k\ln m + \ln^2 m})$ used in Chapter 3.

Multiplexed tables

Let $X_{i,\beta}$ be a random variable which is 1 if the alignment i matches sample β. Then $\mathbf{E}[X_{i,\beta}] = \mu = 1/A^b \leqslant 1/m$, and $\sigma^2 \leqslant (m-1)/m^2$. For an entry $\Lambda[i']$ of a *subtable* of size λ, let

$$X = \sum_{j=1}^{N} \sum_{i \equiv i' (\mathrm{mod}\ \lambda)} X_{i,\beta}.$$

X represents the number of points in $\Lambda[i']$ after N samples. X consists of $N_0 = Nm/\lambda$ identical Bernoulli trials, $X_{i,\beta}$, and we solve

$$\Pr(X - N_0\mu \geqslant r) \leqslant \exp\left(\frac{-r^2}{4N_0\sigma^2}\right) \leqslant \frac{1}{\varepsilon}. \tag{A1}$$

As before, we want $\Pr(X \geqslant \tau \equiv N-k)$; thus, $r = ((\lambda-1)N/\lambda - k)$, giving an exponent of

$$\frac{r^2}{4N_0\sigma^2} = \frac{r^2}{4Nm\sigma^2/\lambda} = \frac{((\lambda-1)N - \lambda k)^2}{4N\lambda(m-1)/m}.$$

Taking the natural logarithm of Equation A.1, plugging in the previous equation, and multiplying through by -1 gives the first and third terms of

$$((\lambda-1)N - \lambda k)^2 \geqslant 4N\lambda \ln\varepsilon \geqslant 4N\lambda \frac{(m-1)}{m} \ln\varepsilon.$$

Expanding the first term and subtracting the second term gives the quadratic equation

$$(\lambda-1)^2 N^2-(2\lambda(\lambda-1)k+4\lambda\ln\varepsilon)N+\lambda^2k^2\geqslant 0.$$

Solve the quadratic for N gives

$$N\geqslant\left\lceil\frac{\lambda k}{\lambda-1}+\frac{2\lambda\ln\varepsilon}{(\lambda-1)^2}+\frac{2\lambda\sqrt{k(\lambda-1)\ln\varepsilon+\ln^2\varepsilon}}{(\lambda-1)^2}\right\rceil.$$

Setting $\varepsilon=\lambda^3$ so that $\ln\varepsilon=3\ln\lambda$, and assuming $\lambda\geqslant 5$, we have (for each term in the preceding equation, respectively) $\lambda/(\lambda-1)<1.25$, $\lambda\ln\varepsilon/(\lambda-1)^2<\ln\lambda$, and $2\lambda/(\lambda-1)<3$, and $(\ln\varepsilon)/(\lambda-1)<\ln\lambda$, which gives

$$N\geqslant 1.25k+2\ln\lambda+3\sqrt{k\ln\lambda+\ln^2\lambda}.$$

For all subtable sizes, we have $\lambda\leqslant p_\xi\approx\ln m$, so that $\ln\lambda$ is $O(\ln\ln m)$. Thus, $N=1.25k+O(\sqrt{k}\ln\ln m)$. We note that the smaller the subtable, the larger N must be, as the $\lambda/(\lambda-1)$ factor in the dominant first term increases. We force $\lambda\geqslant 5$ for this reason.

References

[1] Karl Abrahamson, Generalized string matching, *SIAM J. Comput.* **16** (1987) 1039–1048.

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

[3] M. Atallah, Private communication.

[4] Ricardo Baeza-Yates and Mireille Régnier, Fast algorithms for two-dimensional and multiple pattern matching, in: *Proc. SWAT 90*, Lecture Notes in Computer Science, Vol. 447 (Springer, Berlin, 1990) 332–347.

[5] Robert S. Boyer and J. Strother Moore, A fast string searching algorithm, *Comm. ACM* **20** (10) (1977) 762–772.

[6] William I. Chang and Eugene L. Lawler, Approximate string matching in sublinear expected time, in: *Proc. 31st IEEE Symp. on Foundations of Computer Science* (1990) 116–124.

[7] T. Cormen, C.E. Leiserson and R. Rivest, *Introduction to Algorithms* (McGraw Hill, New York, NY, 1990).

[8] M.J. Fischer and M.S. Paterson, String matching and other products, in: R.M. Karp, ed., *Complexity of Computation* (*SIAM-AMS Proceedings 7*) (American Mathematical Society, Providence, RI, 1974) 113–125.

[9] Z. Galil and K. Park, An improved algorithm for approximate string matching, in: Lecture notes in computer science, Vol. 372 (Springer, Berlin, 1989) 394–404.

[10] R. Graham, D.E. Knuth and O. Patashnik, *Concrete Mathematics* (Addison-Wesley, Reading, MA, 1989).

[11] Donald Knuth, *Seminumerical Algorithms, The Art of Programming Vol. 2* (Addison-Wesley, Reading, MA, 1981).

[12] Donald E. Knuth, James H. Morris and Vaughan R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (2) (1977) 323–350.

[13] Gad M. Landau and Uzi Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in: *Proc. 18th ACM Symp. on Theory of Computing* (1985) 220–230.

[14] E.M. McCreight, A space-economical suffix tree construction algorithm, *J. ACM* **23** (2) (1976) 262–272.

[15] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplification and parallelization, *SIAM J. Comput.* **17** (6) (1988) 1253–1262.

[16] P. Weiner, Linear pattern matching algorithms, in: *Proc. 14th IEEE Symp. on Switching and Automata Theory* (1973) 1–11.

[17] Andrew Chi-Chih Yao, The complexity of pattern matching for a random string, *SIAM J. Comput.* **8** (3) (1979) 368–387.

[18] Rui F. Zhu and Tadao Takaoka, A technique for two-dimensional pattern matching, *Comm. ACM* **32** (1989) 1110–1120.