Inverse Pattern Matching

Amihood Amir* Alberto Apostolico† Moshe Lewenstein‡
Georgia Tech and Bar-Ilan University
Purdue and Università di Padova
Bar-Ilan University

August 4, 1996

Abstract

Let a textstring T of n symbols from some alphabet Σ and an integer $m < n$ be given. A pattern P of length m over Σ is sought such that P minimizes (alternatively, maximizes) the total number of pairwise character mismatches generated when P is compared with all m-character substrings of T. Two additional variants of the problem are obtained by adding the constraint that P be (respectively, not be) a substring of T. Efficient sequential algorithms are proposed in this paper for the problem and its variants.

Key Words: Design and analysis of algorithms, combinatorial algorithms on words, pattern matching, inverse pattern matching, Hamming distance, digital signature.

1 Introduction

Inverse pattern matching refers to the task of inferring from a given textstring T a short pattern string P such that P is, by some measure, most typical (or, alternatively, most anomalous) in the context of T. This problem arises in a wide variety of applications and takes up numerous flavors, among which most common is probably the one based on frequencies of pattern occurrences. When such occurrences need not be exact, alternative measures of typicality can be based on some notion of similarity among string, such as the Hamming [7] or Levenshtein [9] distances. Given a textstring T and an integer m, for example, one might ask for a pattern P that scores the smallest (or largest) total number of mismatches when aligned with all substrings of T. Noteworthy variants of the

*Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8770; amir@bimacs.cs.biu.ac.il; Partially supported by NSF grant CCR-92-23699 and the Israel Ministry of Science and the Arts grant 6297.

†Dipartimento di Elettronica e Informatica, Università di Padova, Via Gradenigo 6/A, 35131 Padova, Italy, (39-49)828-7710; axa@art.dei.unipd.it; partially supported by NSF grant CCR-92-01078, by NATO grant CRG 900293, by the National Research Council of Italy, and by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM).

‡Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; moshe@bimacs.cs.biu.ac.il.

problem arise when the constraint is added that P must be a substring of T, or, symmetrically, that P must not have any occurrence in T. Efficient (occasionally, optimal) sequential algorithms for the problem and its variants are provided in this paper. Computations of these and similar "distance preserving signatures" (see e.g. [6]) find use in disparate contexts, including information retrieval, data compression, computer security and molecular biology. In the two latter fields, in particular, highly anomalous patterns are also often sought, e.g., in intrusion [11] or plagiarism detection, in the synthesis of molecular probes in genome sequencing by hybridization [3], in designing control (inactive) antisense oligonucleotides [12], etc. Throughout the paper, we treat the inverse pattern matching problem theoretically. In section 11 we consider a specific biological problem that motivated this research, the sequencing by hybridization problem. A detailed description of that particular problem appears in section 11.

2 Problem Definition – Inverse Pattern Matching

Definition: Let $T = t_1 \cdots t_n$ be a *text* string over alphabet Σ and $P = p_1 \cdots p_m$ be a *pattern* string over Σ. The *hamming distance between P and text location i* is defined as:

$$ham(P, T_i) = \sum_{i=1}^{m} c(p_j, t_{i+j-1}), \qquad \text{where}$$

$$c(a, b) = \begin{cases} 0, & \text{if } a = b; \\ 1, & \text{if } a \neq b. \end{cases}$$

The *average hamming distance between P and T* is

$$ham(P, T) = \frac{\sum_{i=1}^{n-m+1} ham(P, T_i)}{n - m + 1}.$$

The *(min) Inverse Pattern Matching Problem* is the following:
INPUT: Text string $T = t_1 \cdots t_n$ and positive integer $m \leq n$.
OUTPUT: A pattern string $P_{min} = p_1 \cdots p_m$ (of length m) where $ham(P_{min}, T) \leq ham(P, T)$ for all strings $P \in \Sigma^m$, where Σ is the set of distinct elements of T.

The symmetric *(Max) Inverse Pattern Matching Problem* seeks instead a pattern P_{Max} such that $ham(P_{Max}, T) \geq ham(P, T)$ with respect to all $P \in \Sigma^m$. Both versions of the problem are solved by the same basic strategy. To fix the ideas, we will deal with the "min" version till noted otherwise.

3 The Algorithm

The naive algorithm for the inverse pattern matching problem is computing the hamming distance for every possible substring of length m, and choosing the minimum. This algorithm is clearly bad since it takes exponential time. We present an optimal algorithm for solving the problem. The algorithm adds up the number of appearances of the various alphabet symbols in the text, and uses these sums to determine the best symbol for each pattern location. Let $\Sigma = \{a_1, ..., a_\sigma\}$ be the set of distinct elements of T.

Algorithm:

1. { *Initialize* }
   for $i = 1$ to σ do: $sum[i] \leftarrow 0$; end
2. { *find sums for first pattern symbol* }
   for $i = 1$ to $n - m + 1$ do
       if $t_i = a_j$ then $sum[j] \leftarrow sum[j] + 1$
   end
3. { *Choose first pattern symbol.* }
   Let j be such that $sum[j] \geq sum[k]; \quad k = 1, ..., \sigma$.
   $p_1 \leftarrow a_j$
4. { *Choose rest of pattern symbols.* }
   for $i = 2$ to m do
       if $t_{i-1} = a_{j_1}$ and $t_{n-m+i} = a_{j_2}$ then
           $sum[j_1] \leftarrow sum[j_1] - 1$
           $sum[j_2] \leftarrow sum[j_2] + 1$
       Let j be such that $sum[j] \geq sum[k]; \quad k = 1, ..., \sigma$.
       $p_i \leftarrow a_j$
   end

end Algorithm

4 Algorithm Correctness

Consider $p_1, ..., p_m$ as variables whose values need to be chosen so that they minimize the hamming distance. Following Fischer and Paterson [5], we define the following functions for every $a \in \Sigma$.

$$\chi_a(b) = \begin{cases} 1, & \text{if } a = b; \\ 0, & \text{if } a \neq b. \end{cases} \qquad \chi_{\overline{a}}(b) = 1 - \chi_a(b).$$

It is not hard to see that

$$ham(P, T_i) = \sum_{a \in \Sigma} \sum_{j=1}^{m} \chi_a(t_{i+j-1}) \chi_{\overline{a}}(p_j).$$

Therefore the sum we are trying to minimize is

$$\sum_{i=1}^{n-m+1} \sum_{a \in \Sigma} \sum_{j=1}^{m} \chi_a(t_{i+j-1}) \chi_{\overline{a}}(p_j) = \sum_{j=1}^{m} \sum_{a \in \Sigma} \left(\chi_{\overline{a}}(p_j) \sum_{i=1}^{n-m+1} \chi_a(t_{i+j-1}) \right).$$

Since the p_j's can be chosen independently of each other, then in order to minimize the sum, it is sufficient to minimize, for each $j = 1, ..., m$ the sum:

$$\sum_{a \in \Sigma} \left(\chi_{\overline{a}}(p_j) \sum_{i=1}^{n-m+1} \chi_a(t_{j-1+i}) \right).$$

Depending on the choice of p_j, this sum will end up being the summation of $\sigma - 1$ sums. To minimize it, we need to discard the largest sum. This will happen if we choose $p_j = a$ where $\sum_{i=1}^{n-m+1} \chi_a(t_{j-1+i}))$ is largest.

5 Algorithm Time Complexity

For bounded alphabet the time is clearly $O(n + m) = O(n)$.

For unbounded alphabets, two operations need added consideration.

- Comparison: Determining what alphabet symbol t_i is (done n times).
- Maximum: Determining the maximum of $sum[i], \quad i = 1, ..., \sigma$ (done m times).

We consider three possibilities.

1. If $\Sigma \subseteq \{1, ..., n\}$, then the comparisons can be done in constant time by table lookup. The maximum can be chosen by keeping linked lists of the sums of size $i \quad i = 1, ..., n$, with a pointer to the largest nonempty list. Updating sums and choosing maxima can be easily done in constant time. The total time in this case is still $O(n)$.
2. If the elements of Σ are ordered, then in time $O(n \log \sigma)$ we can translate the problem to a problem over alphabet $\Sigma \subseteq \{1, ..., n\}$, making the total time $O(n \log \sigma)$.
3. If the elements of Σ are incomparable (a highly theoretical case, impossible when the inputs are computer data), we can still translate the problem to one over alphabet $\Sigma \subseteq \{1, ..., n\}$ in time $O(n\sigma)$, and this is then the total time.

6 Algorithm Optimality

We show a linear reduction from the *element distinctness problem* to the inverse pattern matching problem.

Definition: The *element distinctness problem* has as its input a list of n elements. The output is a decision whether every element appears *exactly* once.

From Borodin et. al. ([4]) it can be seen that the element distinctness problem has lower bounds $O(n)$, $O(n \log \sigma)$ and $O(n\sigma)$ depending on a range of elements of type 1., 2., or 3. as above. Therefore a linear reduction from element distinctness to our problem will assure us of our algorithm's optimality.

The Reduction: Given a list of n elements, take the list as the text string, and take $m = 1$. The minimum hamming distance is $(n-1)/n$ iff all elements are distinct.

7 Problem Definition and Algorithm – Internal Inverse Pattern Matching

We now consider the case where the pattern of length m we desire is a *substring* of the text with the minimum average hamming distance.

The *Internal (min) Inverse Pattern Matching Problem* is the following:
INPUT: Text string $T = t_1 \cdots t_n$ and positive integer $m \leq n$.
OUTPUT: A pattern string $P_{min} = p_1 \cdots p_m$ (of length m) where P_{min} is a substring of T and $ham(P_{min}, T) \leq ham(P, T)$ for all strings P that are substrings of T, i.e. $P \in \{t_i t_{i+1} \cdots t_{i+m-1} | i = 1, ..., n-m+1\}$.

The naive algorithm for solving this problem is for each of the $n-m+1$ substrings to find its average hamming distance, and choose the substring with the minimum average hamming distance. The time is thus $O(n^2 m)$. Using more sophisticated techniques for computing the hamming distance, such as the Abrahamson algorithm [1], this ends up being $O(n^2\sqrt{m}\log^2 m)$. We will reduce the running time by a factor of n.

We start by showing an algorithm whose running time is $O(nm)$. We will then refine it to achieve time $O(n\sqrt{m}\log^2 m)$.

The idea is similar to the inverse pattern matching problem. We count appearances of the various characters, and use them for a fast computation of the average hamming distance.

The algorithm has two stages, a preprocessing stage and a pattern construction stage. Let $\Sigma = \{s_1, ..., s_\sigma\}$ be the alphabet. $(\sigma \leq n)$

Algorithm:

Preprocessing:

construct an $m \times \sigma$ table C such that entry $C[i, j]$ is the number of occurrences of symbols *different from* s_j in substring $t_i t_{i+1} \cdots t_{n-m+i}$.

Pattern Construction:

1. for $i = 1$ to $n - m + 1$ do

   compute the average hamming distance h_i of $t_i t_{i+1} \cdots t_{i+m-1}$ in T by using table C as follows:

$$h_i \leftarrow \frac{\sum_{j=1}^{m} C[j, I(i,j)]}{n - m + 1}$$

   where $t_{i+j-1} = s_{I(i,j)} \in \Sigma, \ j = 1, ..., m.$

   end

2. choose $t_k ... t_{k+m-1}$ where h_k is the smallest of the h_i's

end Algorithm

8 Algorithm Correctness and Time Complexity

Correctness: Observe that table entry $C[i,j]$ supplies the number of mismatches that symbol s_j incurs if it is the i-th pattern character. Thus, the formula for computing h_i is correct.

Preprocessing Time: In reality, we can compute all relevant values of the table in time $O(n)$. It would have been possible to use sparse table techniques and restrict the preprocessing time to $O(n)$. However, since the pattern construction takes time $O(nm)$, we may as well use the full table, and the time is $O(\sigma m) \leq O(nm)$.

Pattern construction Time: $O(nm)$.

9 Faster Algorithm

Consider step 1 of the algorithm's pattern construction stage. Computing h_i takes time m. We would like to use h_i in order to produce h_{i+1} in constant time.

Note that almost all mismatches introduced by substring $t_{i+1} \cdots t_{i+m}$ were also mismatches in substring $t_i \cdots t_{i+m-1}$, since they share $m - 1$ symbols. In order to compute h_{i+1} from h_i, before dividing both by $n - m + 1$, we need to make the following four changes:

1. Subtract the mismatches introduced by having t_i as the first symbol. This can be done in constant time since it is $C[1, I(i,1)]$, where $t_i = s_{I(i,1)} \in \Sigma$.

2. Add the mismatches introduced by having t_{i+m} as the m-th symbol. This can be done in constant time since it is $C[m, I(i, m+1)]$, where $t_{i+m} = s_{I(i,m+1)} \in \Sigma$.

3. Subtract the mismatches introduced by matching $t_{i+1} \cdots t_{i+m-1}$ against the last symbols in T. This is $ham(t_{i+1} \cdots t_{i+m-1}, t_{n-m+2} \cdots t_n)$.
4. Add the mismatches introduced by matching $t_{i+1} \cdots t_{i+m-1}$ against the first symbols in T. This is $ham(t_{i+1} \cdots t_{i+m-1}, t_1 \cdots t_{m-1})$.

We will add $O(n\sqrt{m}\log^2 m)$ to the preprocessing, to enable computing the above two hamming distances in constant time. Notice that the hamming distances of all $m-1$ length substrings, are computed against the first $m-1$ elements of T and the last $m-1$ elements of T. The Abrahamson algorithm [1] has as its input a text T (of length n) and pattern P (of length m). It constructs an array of length $n-m+1$ that has in location i the value $ham(P, T_i)$, in time $O(n\sqrt{m}\log^2 m)$. This algorithm is exactly what we need, with P being the first $m-1$ elements of T and with P being the last $m-1$ elements of T.

We therefore add to the preprocessing the following two steps:

1. Construct Array H_{prefix} where $H_{prefix}[i] = ham(t_{i+1} \cdots t_{i+m-1}, t_1 \cdots t_{m-1})$.
2. Construct Array H_{suffix} where $H_{suffix}[i] = ham(t_{i+1} \cdots t_{i+m-1}, t_{n-m+2} \cdots t_n)$.

The pattern construction stage now looks as follows:

Pattern Construction:

1. $$h_1 \leftarrow \sum_{j=1}^{m} C[j, I(1,j)]$$
2. for $i = 1$ to $n-m+1$ do

   $$h_{i+1} \leftarrow h_i - C[1, I(i,1)] + C[m, I(i, m+1)] - H_{suffix}[i+1] + H_{prefix}[i+1]$$

   end
3. for $i = 1$ to $n-m+1$ do

   $h_i \leftarrow \frac{h_i}{n-m+1}$

   end
4. choose $t_k ... t_{k+m-1}$ where h_k is the smallest of the h_i's

The problem is that table C is still $m \times \sigma$ which may be as large as mn. In order to improve this, note that within the loop of the pattern construction all we use from the table is the first and m-th row. Therefore we will indeed save only rows C_1 and C_m. (We denote the $i-th$ row from table C with C_i and $C[i,j]$ with $C_i[j]$.) C_1 and C_m are of size $O(\sigma)$ and can be calculated in $O(n)$ time by adding the following to the preprocessing:

3a. **for** $j = 1$ **to** σ **do**

$$C_1[j] \leftarrow n - m + 1$$
$$C_m[j] \leftarrow n - m + 1$$

end

3b. **for** $j = 1$ **to** $n - m + 1$ **do**: $C_1[t_j] \leftarrow C_1[t_j] - 1$; **end**

3c. **for** $j = m$ **to** n **do**: $C_m[t_j] \leftarrow C_m[t_j] - 1$; **end**

This still leaves us with the calculation of h_1 in step 1 which seemingly requires all rows C_j. We make one last change to the algorithm to solve this problem.

In the table C the difference between two consecutive rows can be in at most two columns (since in row i we are looking at $t_i \cdots t_{n-m+i}$ and in row $i+1$ at $t_{i+1} \cdots t_{n-m+i+1}$). Therefore we can calculate row $i+1$ from row i in constant time and row m from row 1 in time $O(m)$.

This suggests the following strategy. Initialize C_m to C_1 and have it simulate the change to C_2 and then to C_3 etc. till it is C_m. During the change use the temporary information to calculate h_1.

3b. **for** $j = 1$ **to** $n - m + 1$ **do**

$$C_1[t_j] \leftarrow C_1[t_j] - 1$$
$$C_m[t_j] \leftarrow C_m[t_j] - 1$$

end

3c. $h_1 \leftarrow C_m[t_1]$

3d. **for** $j = 2$ **to** m **do**

$$C_m[t_{j-1}] \leftarrow C_m[t_{j-1}] + 1$$
$$C_m[t_{n-m+j}] \leftarrow C_m[t_{n-m+j}] - 1$$
$$h_1 \leftarrow h_1 + C_m[t_j]$$

end

We can now sum up our algorithm's time.

$O(n)$ for computing C_1, C_m and h_1.

$O(n\sqrt{m}\log^2 m)$ for computing H_{prefix} and H_{suffix}.

$O(n)$ for computing $h_i,\ i = 2, ..., n - m + 1$.

Total Time: $O(n\sqrt{m}\log^2 m)$.

Note that the bottleneck in this algorithm is pattern matching with mismatches. Karloff [8] showed an approximation algorithm that computes all mismatches in time $O(n \log m)$. Using his algorithm, we can compute a substring inverse pattern in time $O(n \log m)$, that is within ϵ of the minimum average hamming distance.

10 Reduction from the All Mismatches problem

As we noted the substring inverse pattern can be computed as fast as computing all mismatches. Conversely we can compute all mismatches as fast as we can compute all $h_i's$. Let $t_1...t_n$ be the text and $p_1...p_m$ the pattern for which we are to compute all mismatches. Let $p_1...p_m t_1...t_n \$^m$, where $\$ \notin \Sigma$, be the text of the substring inverse pattern matching problem and $m' = m + 1$ the positive integer. Note that the number of mismatches between $p_1...p_m$ and $t_i...t_{i+m-1}$ is exactly $H_{prefix}[i+m]$ of the string $p_1...p_m t_1...t_n \$^m$. From line 2 in the pattern construction $H_{prefix}[i+1] = h_{i+1} - h_i + C_1[i_1] - C_{m'}[i_{m'+1}] + H_{suffix}[i+1]$. If we denote with $M[i]$ the number of mismatches between $p_1...p_m$ and $t_i...t_{i+m-1}$ then $M[i] \leftarrow h_{i+m} - h_{i+m-1} + C_1[i_1] - C_{m'}[i_{m'+1}] - H_{suffix}[i+m]$. C_1 and $C_{m'}$ can be preprocessed in linear time and $H_{suffix}[i+1]$ is m for $i \leq n$. Therefore computing all mismatches is as fast as computing all h_i's.

Muthukrishnan [10] showed that the all-mismatches problem can not be solved in a convolution model using less than $O(\sqrt{m})$ convolutions. The above reduction means that we are computing the h_i's optimally in the convolution model.

11 External Inverse Pattern Matching

Through the remainder of our discussion, we will assume a change in terms of "Max" inverse pattern matching. As noted earlier, a solution to either one of the "Max" or "min" version of the problem extends trivially to the other. We change our terms here mainly to accomodate the main motivation for this version of the problem [12]. We briefly describe this application.

As is well known, a single strand of DNA is a string written on an alphabet of four "letters" where each letter corresponds to one of four different "nucleotides". The task of determining the exact letter sequence inherent to a given string of DNA is called "sequencing". One sequencing technique, called "by hybridization", is performed in two main stages, as follows. First, a matrix (also called a "chip") is prepared, the entries of which correspond each to a distinct possible k-tuple. When the sample single-stranded DNA to be sequenced is presented to this matrix, each k-tuple present in the matrix gets chemically "marked" by a corresponding k-tuple in the sample. Next, the task is faced to reconstruct the sequence starting from the set of the observed marked k-tuples. Of course, the larger the k, the easier and less prone to errors will be the task of assembling k-tuples into a sequence. However, building chips for all k-tuples become quickly unmanageable as k grows. To complicate matters, there may be also errors (mismatches) in the detected k-tuples, as some nucleotides "stick" to the matrix better than others. Assume though that a chip with only part of the k-tuples has led to successfully sequence some portions of a DNA sequence. At that point, the chances to sequence successfully the unknown regions are enhanced by setting up chips that contain k-tuples that are as distant as possible from the ones already identified.

This application required precisely the maximum inverse pattern matching. The case considered here is when the desired pattern (1) never occurs in the text and (2) has maximum average hamming distance from it. Note that the pattern symbols must come from the text alphabet, otherwise the problem is vacuous. Moreover, there might be no solution even when this condition is met, since the text could contain every possible string of the specified length.

Formally, the *External (Max) Inverse Pattern Matching Problem* is defined as follows:
INPUT: Text string $T = t_1 \cdots t_n$ over Σ and positive integer $m \leq n$.
OUTPUT: A pattern string $P_{Max} = p_1 \cdots p_m$ (of length m) over Σ, if it exists, where P_{Max} is not a substring of T and $ham(P_{Max}, T) \geq ham(P, T)$ for all strings P over Σ that are not substrings of T.

Unlike the internal problem, the number of candidate patterns in the external problem does not seem at first to be bounded by a polynomial of n. Yet the following notion, somewhat reminiscent of that of a "position identifier" (cf. [2], ch.9), gives a handle that leads quickly to such a bound.

Definition: A string $X = x_1 \cdots x_{g+1}$ $(0 < g < m)$ over Σ is an *m-stem* for $T = t_1 \cdots t_n$ if X is not a substring of $T^{(g+1)} = t_1 \cdots t_{n-m+g+1}$ but its longest proper prefix $X' = x_1 \cdots x_g$ is a substring of $T^{(g)} = t_1 \cdots t_{n-m+g}$.

First, we apply the general (Max) inverse pattern matching of Section 3. If we're lucky the solution is external and we have a desired result. Externality can easily be checked by regular pattern matching techniques. If the solution is internal then the following fact holds.

Fact 1: If there is an internal solution $P = p_1 ... p_m$ then every external solution $Q = q_1 ... q_m$ has an m-stem.

Proof: Since Q is an external solution, Q is not a substring of T and therefore it suffices to show that q_1 appears at least once in $T^{(1)}$. Since P is internal, p_1 appears in $T^{(1)}$. If q_1 does not appear in $T^{(1)}$ then $\text{ham}(q_1 p_2 p_3 ... p_m, T) > \text{ham}(P,T)$ a contradiction to maximality of $\text{ham}(P,T)$. Therefore q_1 appears in $T^{(1)}$ and hence Q has an m-stem. $\square$

Fact 1 shows that the search for an optimal pattern may be confined among strings that are extensions of m-stems, which are in turn unit extensions of certain substrings of T.

Fact 2: Let $P_{Max} = p_1 \cdots p_m$ be an optimal solution and $p_1 \cdots p_{g+1}$ its corresponding m-stem. Then, the average hamming distance between $p_1 \cdots p_{g+1}$ and $T^{(g+1)}$ is maximum among all m-stems for T of length $g+1$, and the average hamming distance between $p_{g+2} \cdots p_m$ and $t_{g+2} \cdots t_n$ is maximum among all strings in $\Sigma^{[m-(g+1)]}$.

Proof:. Clearly, any string produced by appending some symbols to an m-stem for T is still not a substring of T. Therefore, once g is fixed, the m-stem $p_1 \cdots p_{g+1}$ and the string $p_{g+2} \cdots p_m$, as they are both found in an optimal pattern P_{Max}, can be chosen independently of each other. $\square$

Let $\mathcal{P}_d$ $(1 < d \leq m)$ denote the set of the best solutions among those that can be built using an m-stem of length d.

Fact 3: Let $p_f \cdots p_m$ be a suffix of an element of $\mathcal{P}_g$. Then, in any $\mathcal{P}_h \not\equiv \emptyset$ with $h < g$, there is some element of $\mathcal{P}_h$ of which $p_f \cdots p_m$ is a suffix, too.

Proof:. Same basic argument as in Fact 2. $\square$

Observe that, once the value of g for an optimum pattern P_{Max} is known, then the choice of symbols $p_{g+2} \cdots p_m$ of P_{Max} is done by trivial adaptation of the algorithm of Sec. 3 and within the corresponding time bounds. Actually, Fact 3 suggests that an optimal selection of the symbols in $p_{g+2} \cdots p_m$ may be computed for all values of g by running that algorithm just once. In fact, the consecutive proper suffixes of the output of that algorithm are also the suffixes of optimal solutions relative to m-stems of length 2, 3, etc. In conclusion, we only need to shown how an optimal m-stem is found. The following brief discussion explains how this is done.

We consider first that, for any value of g in $[1, m-1]$, applying the algorithm of Section 9 to $T^{(g)} = t_1 \cdots t_{n-m+g}$ yields the average hamming distance between $T^{(g)}$ and each of its substrings of length g. For every such substring, we would need then to find out whether or not it can be extended into an m-stem of length $g+1$ for T and, in the affirmative, we need to compute a corresponding m-stem of maximum cost. However, the overall cost of the applications of the algorithm of Sec. 9 alone would be already $O(nm\sqrt{m}\log^2 m)$. We present a much simpler approach that takes only $O(nm \log \sigma)$ time.

12 Algorithm for External Inverse Pattern Matching

We start by building a digital search tree (*trie*) containing all substrings of m symbols in T.

Notation: For each g in $[1, m]$, every substring of $T^{(g)}$ is associated with a unique leafward path from the root of the trie. The node on this path at which a string X ends is called the *locus* of X.

Algorithm:

Preprocessing:

1. construct the $m \times \sigma$ table C as in the algorithm for internal inverse pattern matching. Sort each row of C by descending order of symbol frequency.

2. Construct the trie of all m-length substrings of T. Sort all sons of a node in level g by descending order of symbol frequency in row $C[g+1, 1..\sigma]$.

3. Assign a weight to every locus of string X whose length is less than m as follows:

   (a) Every level 1 node is a locus of some symbol $s_j \in \Sigma$. Give such a node the weight $C[1, j]$. (i.e., the number of mismatches generated by s_j if this were the first symbol in a pattern of length m).

   (b) Let l be a level $g+1$ node that is the locus of $X = x_1 x_2 \cdots x_{g+1}$, where $x_{g+1} = s_j \in \Sigma$, and where l' is the locus of $x_1 x_2 \cdots x_g$. Then
   $weight(l) \leftarrow weight(l') + C[g+1, j]$.

4. For every node l of depth g, set $next(l)$ to be the $s \in \Sigma$ with the highest frequency in $C[g+1, 1..\sigma]$ and that does not appear as a son of l. If l has σ sons, then $next$ indicates this fact.

5. Compute $P_{Max} = p_1 p_2 \cdots p_m$ using the algorithm of section 3.

6. For $g = 3, \ldots, m$ compute W_g, the average hamming distance of $p_g p_{g+1} \cdots p_m$ in $t_g \cdots t_n$.

We now have a weighted trie where the weight of the locus of string $X = x_1 \cdots x_g$ is the average hamming distance between X and $T^{(g)}$. We also have the best average hamming distance that one can get by appending a suffix to an m-stem, for all m-stems.

Pattern Construction:

Scan the trie. For each node l in level g compute $weight(l) + C[g+1, next] + W_g$. The maximum score provides the desired pattern.

end Algorithm

13 Algorithm Correctness and Time Complexity

Correctness: Facts 1 and 2 tell us to look for the maximal m-stem. Consider the locus of $X = x_1 x_2 \cdots x_g$. Clearly, X can be extended into an m-stem if and only if there is some symbol $s \in \Sigma$ such that $x_1 x_2 \cdots x_g s$ does not have a locus in the trie. Among such strings, moreover, the one(s) corresponding to symbols maximizing $C[g+1, j]$ yield the best m-stem among such stems that are unit extensions of X. The algorithm detects whether such an s exists, and chooses the maximal. Computing these optimal extensions for all loci in the trie, while keeping track in the process of the best one for each g, gives the best m-stem for each length.

Finally, pairing up these stems each with the appropriate suffix of the string produced by algorithm of Sec. 3, gives the desired solution pattern, because of fact 3.

Time: Building the trie is done by standard procedures, and it charges $O(nm \log \sigma)$ time. Building the table C can be done in $O(m\sigma)$ and sorting its rows for the required order takes $O(m\sigma \log \sigma)$ therefore preprocessing table C costs $O(nm \log \sigma)$.

Finding *next* for all vertices in the trie takes $O(nm)$ time by the following implementation. For every depth g node, check all its children (that are sorted by symbol frequency) against the row $C[g+1, 1..\sigma]$ (also sorted by symbol frequency) until a symbol is encountered in C that is not a son of this node. This element is the *next*. Clearly the time is proportional to the number of edges, i.e. $O(nm)$.

Computing W_g takes $O(n+m)$=$O(n)$ time for all g, since we compute $P_{Max} = p_1 p_2 ... p_m$ in step 5 in $O(n)$ time and W_{g+1} can be computed from W_g in constant time by lookup in table C.

For all practical cases the algorithm of Sec. 3 takes $O(n \log \sigma)$ time and therefore the total cost of the algorithm is $O(nm \log \sigma)$.

14 Conclusion

We have shown an optimal algorithm for the inverse pattern matching problem. When restricted to the internal inverse pattern matching problem, we provided a $O(n\sqrt{m} \log^2 m)$ algorithm. There is evidence to support the equivalence of this problem and pattern matching with mismatches. We proved that any improvement in the time of the mismatch problem will improve our algorithm accordingly, but improving the mismatch problem seems difficult.

Surprisingly, the hardest variant turned out to be the external inverse pattern matching problem. There our time complexity is $O(nm \log \sigma)$ and we do not know anything about the lower bounds.

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Computing*, 16(6):1039–1051, 1987.

[2] A. V. Aho, J.E. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.

[3] B. Alberts, D. Bray, J. lewis, M. Raff, K. Roberts and J.D. Watson. *Molecular Biology of the Cell.* Garland Publishing, N.Y., 1989.

[4] A. Borodin, F. Fich, F. Meyer auf der Heide, E. Upfal, and A. Wigderson. A time-space tradeoff for element distinctness. *SIAM J. Computing*, 16(1):97–99, 1987.

[5] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[6] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. *Proc. 35th Annual Symposium on Foundations of Computer Science*, pages 722–731, 1994.

[7] R.W. Hamming. Error detecting and error correcting codes. *Bell. Sys. Tech. Journal* 26(2): 147–160, 1950.

[8] H. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, 1993.

[9] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10: 707–710, 1966.

[10] S. Muthukrishnan. Non-standard Stringology: Algorithms and Complexity. *Proc. 26th Annual Symposium on the Theory of Computing*, pages 770–779, 1994.

[11] D. Russel and G.T. Gangemi, Sr. *Computer Security Basics*. O'Reilly and Associates, Inc., Sebastopol, California, 1991.

[12] L. Weith. Personal communication, 1995.