Episode Matching

Gautam Das[1] Rudolf Fleischer[2] Leszek Gąsieniec[2]
Dimitris Gunopulos[3] Juha Kärkkäinen[4]

[1] Dept. of Mathematical Sciences, The University of Memphis, Memphis TN 38152, USA; dasg@mathsci.msci.memphis.edu
[2] Max-Planck Institut für Informatik, Im Stadtwald, Saarbrücken D-66123, Germany; {rudolf,leszek}@mpi-sb.mpg.de
[3] IBM Almaden RC k55/B1, 650 Harry Rd, CA 95120, USA; gunopulo@almaden.ibm.com
[4] Dept. of Computer Science, P.O. Box 26, FIN-00014 University of Helsinki, Finland; juha.karkkainen@cs.helsinki.fi

Abstract. Given two words, text T of length n and episode P of length m, the episode matching problem is to find all minimal length substrings of text T that contain episode P as a subsequence. The respective optimization problem is to find the smallest number w, s.t. text T has a subword of length w which contains episode P.
In this paper, we introduce a few efficient off-line as well as on-line algorithms for the entire problem, where by on-line algorithms we mean algorithms which search from left to right consecutive text symbols only once. We present two alphabet independent algorithms which work in time $O(nm)$. The off-line algorithm operates in $O(1)$ additional space while the on-line algorithm pays for its property with $O(m)$ additional space. Two other on-line algorithms have subquadratic time complexity. One of them works in time $O(nm/\log m)$ and $O(m)$ additional space. The other one gives a time/space trade-off, i.e., it works in time $O(n + s + nm \log\log s/\log(s/m))$ when additional space is limited to $O(s)$.
Finally, we present two approximation algorithms for the optimization problem. The off-line algorithm is alphabet independent, it has superlinear time complexity $O(n/\epsilon + n \log\log(n/m))$ and it uses only constant space. The on-line algorithm works in time $O(n/\epsilon + n)$ and uses space $O(m)$. Both approximation algorithms achieve $1+\epsilon$ approximation ratio, for any $\epsilon > 0$.

1 Introduction

In [6], Mannila et al. introduced the problem of finding frequent episodes in event sequences. An *episode* is a collection of events that occur within short time interval. Given a long sequence of events, e.g. alarms from a telecommunication network, it can be useful to know what episodes occur frequently in the sequence.

A simplified version of this problem in string matching terms is: Given a text T, a window width w and a frequency threshold t, find all strings (episodes) P that satisfy the condition:

Episode Condition 1. The text T has (at least) t different substrings of length w that contain P as a subsequence.

The problem of finding frequent episodes was also considered in [5], but with a different definition of "frequent" based on minimal substrings. A substring containing P is *minimal* if no proper substring of it contains P. The corresponding episode condition is

Episode Condition 2. The text T has (at least) t different *minimal* substrings of length at most w that contain P as a subsequence.

Obviously, Condition 2 implies Condition 1, but not vice versa.

In this paper, we consider problems rising from the episode conditions when both T and P are given. We call these episode matching problems. Our main problem is the following.

Problem 1. Given a text T and an episode P, find all minimal substrings of T that contain P as a subsequence.

From the set of minimal substrings it is easy to compute the smallest w for any t or the largest t for any w that satisfies the episode condition. Only linear time is needed whether the first or the second condition is used. We will give four algorithms for this problem. In addition, we will give two *approximation* algorithms for the problem of finding a minimal w when $t = 1$:

Problem 2. Given a text T and an episode P, find the smallest w such that T has a substring of length w that contains P as a subsequence.

Some of our algorithms are alphabet independent, that is, the only character operation needed is the comparison for equality or inequality. The other algorithms may need an alphabet transformation. Let $\langle P \rangle$ denote the number of different characters in the episode P. The alphabet transformation maps characters appearing in P bijectively into the set $\{1, 2, \ldots, \langle P \rangle\}$ and all other characters into 0. Applying this transformation to P and T changes the alphabet into $\{0, \ldots, \langle P \rangle\}$ but does not change the solutions to our problems. With general alphabets allowing only equality comparisons, the transformation needs $O(nm)$ time, where $n = |T|$ and $m = |P|$. With ordered alphabets allowing order comparisons, the transformation can be done in $O(n \log m)$ time. In practice, array indexing, hashing or other techniques can be used for making the transformation in linear time.

The properties of our algorithms for Problem 1 are summarized below.

Algorithm A Alphabet independent, $O(nm)$ time, $O(1)$ additional space.
Algorithm B Alphabet independent, on-line, $O(nm)$ time, $O(m)$ additional space.
Algorithm C On-line, $O(nm/\log m)$ time, $O(m)$ additional space.
Algorithm DB On-line, $O(n+s+nm \log\log s/\log(s/m))$ time, $O(s)$ additional space.

The approximation algorithms for Problem 2, both achieving $1+\epsilon$ approximation ratio for any $\epsilon > 0$, are summarized below.

Algorithm AA Alphabet independent, $O(n/\epsilon + n \log\log(n/m))$ time, $O(1)$ additional space.

Algorithm AB On-line, $O(n/\epsilon + n)$ time, $O(m)$ additional space.

The time and space requirements for the alphabet dependent algorithms do not include the alphabet transformation but assume that it has been made. No other assumptions are made about the alphabet.

As was mentioned in the beginning, the episode conditions are a simplification of the situation described in [6]. The text T is actually a sequence of events, each of which has a type and an occurrence time. The window width w refers to a time window instead of the number of characters in a substring. Our algorithms for Problem 1 are easily modified to handle this generalization, but the approximation algorithms for Problem 2 lose either their approximation guarantee or their running time guarantee when the events are unevenly distributed in time.

Another generalization given in [6] concerns the episodes. The episode P as described above is called a *serial* episode because the characters (or events) must occur in a fixed order in the substring or time window. A *parallel* episode gives a (multi)set of characters that can occur in any order. A *general* episode specifies an arbitrary partial order on the characters. At least Algorithms A, B, AA and AB can be modified to handle general episodes. Further generalizations to the concept of episodes are described in [5].

An even more general class of patterns are regular expressions. The problem of finding minimal substrings matching a regular expression was described and solved in [1, Sect. 9.2.]. Our Algorithms A, B, AA and AB can be generalized even for regular expressions. In fact, the generalization of Algorithm B is exactly the solution given in [1].

Another related pattern matching problem is approximate string matching which looks for those substrings of the text T that can be transformed into the pattern P with at most k edit operations. When deletion is the only edit operation allowed and we choose $k = w - m$, the problem is equivalent to finding all substrings of T of length at most w that contain P as a subsequence. Our Algorithm B is closely related to the classical $O(nm)$ time dynamic programming algorithm for approximate string matching [8,9].

There are more advanced variations of the dynamic programming algorithm, including $O(nk)$ time [3,2] and $O(nm/\log n)$ time [7,13] algorithms. Due to different properties of the dynamic programming table, most of these algorithms are not directly applicable to episode matching. The general techniques in those algorithms, however, can be useful. Our algorithm D is based on the "Four Russians" technique also used in [7,13]. In [11], Ukkonen describes an automaton approach to approximate string matching and suggests that a part of the dynamic programming table could be computed with the automaton and the rest with a simpler method. This is very similar to the idea of our Algorithm DB.

There are a couple of reasons why episodes have not drawn attention before. One is that while episodes are very useful in event sequences, it is not easy to find natural applications in normal strings. The other is the apparent triviality of subsequence matching. For example, the *agrep* package [12] can find all lines or records that contain the given pattern as a subsequence, which is trivial, indeed. The new twist that makes the problem nontrivial is to look for matching substrings whose length is minimal or limited.

2 Minimal Substrings

Let $T = t_1 t_2 \dots t_n$ be the text and $P = p_1 p_2 \dots p_m$ the episode. A substring $T[i..j] = t_i \dots t_j$ contains P if there exist a sequence $i \le i_1 < i_2 < \dots < i_m \le j$ such that $t_{i_k} = p_k$ for all $k = 1, \dots, m$. The sequence $i_1, \dots, i_m$ is called an occurrence of P in $T[i..j]$. The substring $T[i..j]$ is *minimal* if $i_1 = i$ and $i_m = j$ for all occurrences of P in $T[i..j]$, i.e., no proper substring of $T[i..j]$ contains P. The following lemma gives an important property of occurrences in minimal substrings.

Lemma 3. *Let $i_1, \dots, i_m$ and $i'_1, \dots, i'_m$ be occurrences of an episode P in minimal substrings $T[i_1..i_m]$ and $T[i'_1..i'_m]$ with $i_1 < i'_1$ or $i_m < i'_m$. Then $i_{j+1} \le i'_j$ for $j = 1, \dots, m-1$.*

Proof. Suppose $i_{j+1} > i'_j$ for some j. Then $i'_1, \dots, i'_j, i_{j+1}, \dots, i_m$ is an occurrence, which contradicts $T[i_1..i_m]$ and $T[i'_1..i'_m]$ being minimal. □

Let $ms(P, T)$ denote the number of minimal substrings of T that contain P. Particularly, with a single character c, $ms(c, T)$ is the number of occurrences of c in T. We will need the following results in the analysis of our algorithms.

Lemma 4. *For any $j = 1, \dots, m$, $ms(P, T) \le ms(p_j, T)$.*

Proof. If $i_1, \dots, i_m$ and $i'_1, \dots, i'_m$ are occurrences in two different minimal substrings, then by Lemma 3, $i_j \ne i'_j$. That is, the same text character cannot serve as the jth character of an occurrence in two different minimal substrings. Therefore, the number of p_j's in T must be at least $ms(P, T)$. □

Lemma 5. *The sum of the lengths of the minimal substrings of T containing P is at most nm.*

Proof. Let $1 \le i \le n$ and $1 \le j \le m$. By Lemma 3, there exists at most one minimal substring with an occurrence $i_1, \dots, i_m$ such that $i_j \le i < i_{j+1}$ (where $i_{m+1} = i_m + 1$). Therefore, at most m different minimal substrings can contain the text character t_i. Summing over all text characters gives the result since the length of a substring is the number of characters it contains. □

3 Algorithm A

The following simple algorithm finds all minimal substrings containing P.

Algorithm A
Let $i \leftarrow 0$ and repeat until T ends during the forward scan:

1. [Forward Scan] Starting from position $i+1$ scan text T first looking for p_1, then the next p_2, and so on until p_m is found at position j.
2. [Backward Scan] Starting from position j scan text T *backwards* first looking for p_m, then the next p_{m-1}, and so on until p_1 is found at position i.
3. Report a minimal occurrence at $T[i..j]$.

The algorithm is alphabet independent and requires only $O(1)$ additional space. The running time of the algorithm is clearly $\Theta(n+s)$, where s is the total length of the minimal substrings, which by Lemma 5 is $O(nm)$.

Theorem 6. *The time complexity of Algorithm A is $O(nm)$.*

Algorithm A can be improved by removing unnecessary characters from the text during the scan. When a forward scan is looking for p_{i+1} or a backward scan is looking for p_i, all encountered characters not in $\{p_1, \ldots p_i\}$ can be removed. This makes later scans faster, although the algorithm still needs $\Theta(nm)$ time in the worst case.

Algorithm A is easily modified to handle other kinds of patterns, including general episodes and regular expressions, by doing the forward and backward scans with suitable automatons for the pattern.

4 Algorithm B

Our second algorithm for finding all minimal substrings is closely related to the basic dynamic programming algorithm for approximate string matching [8,9]. The algorithm could also be derived from a minimal substring algorithm for regular expressions [1]. Essentially the same algorithm was also described in [6], and in more detail in [10], as a part of an algorithm for finding all frequent episodes.

The algorithm computes a table $S[0..n, 0..m]$, where $S[i,j]$ is the largest value k such that $T[k..i]$ contains $P[1..j]$. Then, for every i and j such that $k = S[i,j] > S[i-1,j]$, $T[k..i]$ is a minimal substring containing $P[1..j]$. In particular, $T[S[i,m]..i]$ is a minimal substring containing P if and only if $S[i,m] > S[i-1,m]$. The table S can be computed by dynamic programming using the recurrence relation

$$S[i,j] = \begin{cases} S[i-1,j-1] \text{ if } t_i = p_j \\ S[i-1,j] \text{ otherwise.} \end{cases}$$

with the initialization $S[0,j] = 0$ for $j = 1, \ldots, m$ and $S[i,0] = i+1$ for $i = 0, \ldots, n$. The recurrence relation in the basic dynamic programming algorithm for approximate string matching reduces to exactly this form when deletion is the only edit operation allowed.

In practice, the algorithm maintains a table $s[0..m]$ while scanning the text. After reading t_i, $s[j] = S[i, j]$ for all j. To quickly find the entries of s that need to be updated, the algorithm maintains for every character c the set $\textsc{waits}[c]$ of those entries that need to be updated if the next text character is c. The resulting algorithm is as follows.

Algorithm B

1 **for** $j \leftarrow 0$ **to** m **do** $s[j] \leftarrow 0$
2 **for each** $c \in \Sigma$ **do** $\textsc{waits}[c] \leftarrow \emptyset$
3 **for** $i \leftarrow 1$ **to** n **do**
4     $s[0] \leftarrow i$; $\textsc{waits}[p_1] \leftarrow \textsc{waits}[p_1] \cup \{1\}$
5     $Q \leftarrow \textsc{waits}[t_i]$; $\textsc{waits}[t_i] \leftarrow \emptyset$
6   **for each** $j \in Q$ in descending order **do**
7         $s[j] \leftarrow s[j-1]$
8   **if** $j = m$ **then** report minimal substring $T[s[m]..i]$
9   **else** $\textsc{waits}[p_{j+1}] \leftarrow \textsc{waits}[p_{j+1}] \cup \{j+1\}$

Theorem 7. *Algorithm B works correctly and has time complexity* $O\big(n + |\Sigma| + pms(P, T)\big)$*, where*

$$pms(P, T) = \sum_{j=1}^{m} ms(P[1..j], T).$$

Proof. The following invariant holds just before executing line 5.

> For $j = 1, \ldots, m$, $s[j]$ is the largest value k such that $T[k..i-1]$ contains $P[1..j]$ and j is in $\textsc{waits}[p_j]$ if and only if $s[j] > s[j-1]$.

The invariant can be proven by induction on m; we leave the details to the reader. The first part of the invariant shows that the value of $s[j]$ changes when t_i is processed if and only if a minimal substring containing $P[1..j]$ ends at i. The second part shows that $s[j]$ changes every time line 7 is executed. Therefore, lines 6–9 are executed exactly $pms(P, T)$ times which proves the time complexity. The algorithm works correctly because a minimal substring is reported exactly when $s[m]$ changes. □

The value $pms(P, T)$ is bounded by nm. As such, Algorithm B is alphabet dependent and requires $O(m + |\Sigma|)$ additional space. Combined with the alphabet transformation described in the introduction, the algorithm is alphabet independent, and works in $O(nm)$ time and $O(m)$ additional space. Unlike Algorithm A, Algorithm B works on-line, that is, it reads each text character only once and never needs more than $O(m)$ time to process it.

The value $pms(P,T)$ can be further analyzed using Lemma 4. For each prefix $P[1..j]$, we can choose an integer $1 \leq \ell(j) \leq j$. By Lemma 4,

$$pms(P,T) \leq \sum_{j=1}^{m} ms(p_{\ell(j)},T) = \sum_{c \in \Sigma} \big|\{j \mid p_{\ell(j)} = c\}\big| ms(c,T) \tag{1}$$

no matter how we choose the $\ell(j)$'s. The following examples show some of the applications of this result.

- If the character distribution of T is even over the alphabet Σ, $pms(P,T) \leq n|P|/|\Sigma|$ for all P.
- If no character appears more than k times in P, $pms(P,T) \leq k|T|$ for all T.
- If $P = \texttt{aabcaa}$, $pms(P,T) \leq 2\big(ms(\texttt{a},T) + ms(\texttt{b},T) + ms(\texttt{c},T)\big) \leq 2|T|$ for all T. (Choose $\ell(5) = 3$, $\ell(6) = 4$ and $\ell(j) = j$ for $j = 1, \ldots, 4$.)

The value in entry $s[j]$ depends only on the prefix $P[1..j]$. When there are several episodes to be matched to the same text, we can build a trie of them and combine the computation of the entries that depend only on the common prefixes. The result can be a significant saving in computation since the entries $s[j]$ with small j are updated more often than entries with large j. Algorithm B for tries is given in detail below.

Algorithm B for tries
Input: Trie $TR = (V,E)$, V is the set of nodes, E is the set of edges
Egde $e \in E$ is from node $\textsc{from}[e]$ to node $\textsc{to}[e]$ and is labeled by $\textsc{label}[e]$.

for each $c \in \Sigma$ **do** $\textsc{waits}[c] \leftarrow \emptyset$
for each $v \in V$ **do** $\textsc{sleeps}[v] \leftarrow \emptyset$; $s[v] \leftarrow 0$
for each $e \in E$ **do** add e into $\textsc{sleeps}[\textsc{from}[e]]$
for $i \leftarrow 1$ **to** n **do**
    $s[root] \leftarrow i$
    for each $e \in \textsc{sleeps}[root]$ **do**
        move e from $\textsc{sleeps}[root]$ into $\textsc{waits}[\textsc{label}[e]]$
    $Q \leftarrow \textsc{waits}[t_i]$; $\textsc{waits}[t_i] \leftarrow \emptyset$
    for each $e \in Q$ (in FIFO order) **do**
        add e into $\textsc{sleeps}[\textsc{from}[e]]$
        $s[\textsc{to}[e]] \leftarrow s[\textsc{from}[e]]$
        for each $f \in \textsc{sleeps}[\textsc{to}[e]]$ **do**
            move f from $\textsc{sleeps}[\textsc{to}[e]]$ into $\textsc{waits}[\textsc{label}[f]]$
        if $\textsc{to}[e]$ represents P_j **then**
            report minimal substring $T[s[\textsc{to}[e]]..i]$ containing P_j

The values $s[\cdot]$ are stored with the nodes of the trie. An edge e labelled with c is in the set $\textsc{waits}[c]$ if the next occurrence of c will cause an update of $s[\textsc{to}[e]]$. Otherwise, e is stored in the set $\textsc{sleeps}[\textsc{from}[e]]$. Let

$$pms(TR,T) = \sum_{v \in V \setminus \{root\}} ms(\textsc{str}(v),T) \quad ,$$

where $\text{STR}(v)$ is the concatenation of the labels on the path from the root to the node v.

Theorem 8. *Algorithm B for tries works correctly and has time complexity* $O(n + |\Sigma| + pms(TR, T))$.

The proof is essentially the same as the proof of Theorem 7. Otherwise, too, the algorithm has the same properties as the basic algorithm, including time and space requirements with m replaced by the size of the trie.

The partial order of a general episode can be represented with a directed acyclic graph (DAG). The idea of associating a value $s[v]$ with each node v generalizes for DAG's, too. Similarly, the value $s[v]$ can be associated with each state v in an automaton that recognizes a regular expression. This is exactly the idea of the method in [1, Sect. 9.2.].

5 Algorithm C

Algorithm B performs well when the episode has a lot of variation. For example, if every character of the episode is different, Algorithm B runs in $O(n)$ time. On the other hand, if the episode has little variation, for example $P = \texttt{aaa}\ldots$, the algorithm might require $\Theta(nm)$ time. In this section, we describe an algorithm that takes advantage of, on one hand, the good performance of Algorithm B for episodes of high variation, and on the other hand, the repetitiveness of an episode of low variation.

Algorithm C

1. Divide the episode into k distinct pieces $P_1, P_2, \ldots, P_k$.
2. Build a trie TR of the pieces and use Algorithm B for tries to find the minimal substrings containing the pieces.
3. Combine the minimal substrings containing the pieces to find the minimal substrings containing the whole P.

The key to the algorithm is a right choice of the pieces. Our solution is to make each piece P_i as long as possible under the constraint

$$|P_i| \leq \log_{\langle P_i \rangle} m - \log_{\langle P_i \rangle} \log_{\langle P_i \rangle} m,$$

where $\langle P_i \rangle$ is the number of different characters in P_i. The "as long as possible" gives the lower bound

$$|P_i| > \log_{\langle P_i \rangle + 1} m - \log_{\langle P_i \rangle + 1} \log_{\langle P_i \rangle + 1} m - 1$$

for the length of a piece P_i (except possibly P_k). From this we get

$$\begin{aligned} &|P_i| \log \langle P_i \rangle = \Omega(\log m), \text{ if } \langle P_i \rangle \geq 2 \\ &|P_i| = \Omega(\log m), \text{ if } \langle P_i \rangle = 1. \end{aligned} \tag{2}$$

Step 2 of Algorithm C was already described in the previous section. One additional detail is the handling of pieces P_i with $\langle P_i \rangle = 1$. They are not included in the trie TR. Instead, their minimal occurrences are found by an $O(n)$ time algorithm that scans the text and keeps account of the positions of the last $|P_i|$ occurrences of the only character of P_i. By (2), there are at most $O(m/\log m)$ such pieces, and therefore, handling them separately takes $O(nm/\log m)$ time. The running time of the rest of Step 2 is given by the following theorem. The proof is given in the appendix.

Theorem 9. *Algorithm B for trie TR has time complexity $O(nm/\log m)$.*

Step 3 is done with a modified Algorithm B. In Algorithm B, the value $s[j]$ is the starting position of the latest minimal substring containing the prefix $P[1..j]$. Two changes are needed in the accounting of the minimal substrings. First, it is not enough to keep only the starting position, but also the ending position is needed. Second, keeping only the latest minimal substring is not enough. Instead, the minimal substrings containing $P_1 \cdots P_j$ are stored into a queue $Q[j]$. The algorithm is given below.

Algorithm B for sequence of pieces (Step 3 of Algorithm C)

1 **for** $j \leftarrow 0$ **to** k **do** $Q[j] \leftarrow \emptyset$
2 **for** $i \leftarrow 1$ **to** n **do**
3   BEGIN$(R) \leftarrow i$; END$(R) \leftarrow i - 1$; APPEND$(Q[0], R)$
4   **for each** P_j with minimal substring S ending at i **do**
5   **if not** EMPTY$(Q[j-1])$ **then**
6   **while not** EMPTY$(Q[j-1])$ **and**
                    END(TOP$(Q[j-1])) <$ BEGIN(S) **do**
7                 $R \leftarrow$ POP$(Q[j-1])$
8   **if** $j = k$ **then** report minimal substring $T[$BEGIN$(R)..i]$
9   **else** END$(R) \leftarrow i$; APPEND$(Q[j], R)$

The minimal substrings ending at i (line 4) are found by Step 2 of Algorithm C. Therefore, Steps 2 and 3 are best run in parallel, so that Step 3 immediately processes any minimal substrings found by Step 2.

Theorem 10. *Step 3 of Algorithm C has time complexity $O(nm/\log m)$.*

Proof. Step 3 works in time $\Theta(n + \sum_{h=1}^{k} ms(P_h, T))$. This is clear but for the innermost loop. Every round of that loop removes an item from a queue, so the total time spend in the loop cannot be more than the time spend elsewhere adding items to the queues. We still need to show that

$$\sum_{h=1}^{k} ms(P_h, T) = O\left(\frac{nm}{\log m}\right) .$$

Let $P_h = P[a_h..b_h]$ and let $j_1, \ldots, j_{\langle P_h \rangle}$ be a subsequence of $a_h, \ldots, b_h$ such that $\{p_{j_1}, \ldots, p_{j_{\langle P_h \rangle}}\}$ is the set of $\langle P_h \rangle$ characters appearing in P_h. Let $T[i_{a_h}..i_{b_h}]$ be a minimal substring containing an occurrence $i_{a_h}, \ldots, i_{b_h}$ of P_h. The minimal substring will contribute 1 to the above sum. This cost will be evenly distributed among the pairs (i, j) where $i \in \{i_{j_1}, \ldots, i_{j_{\langle P_h \rangle}}\}$ and $j \in \{a_h, \ldots, b_h\}$. The number of such pairs is $\langle P_h \rangle |P_h|$. Given a pair (i, j) that gets a share of a cost, j uniquely identifies the piece P_h. By the condition $t_i = p_{j_l}$, i identifies the j_l with $i = i_{j_l}$. By Lemma 3, the minimal substring is then unique to (i, j). Therefore, no pair (i, j) can get more than one share of the cost of one minimal substring, giving an upper bound $1/\min_h(\langle P_h \rangle |P_h|)$ for the total cost assigned to a pair. By (2) and the fact that $\langle P_h \rangle |P_h| > |P_h| \log \langle P_h \rangle$ when $\langle P_h \rangle \geq 2$, the cost is $O(1/\log m)$. Since the number of different pairs (i, j) is nm, the total cost is $O(nm/\log m)$. □

The additional space requirement of Step 3 can be $\Theta(n)$ because the queues can grow large. However, from the trie TR, we can get all potential starting positions of minimal substrings containing P_j up to the current position. The queue $Q[j-1]$ needs to keep only those minimal substrings containing $P_1 \cdots P_{j-1}$ which best match the potential starting positions. By purging the unneeded minimal substrings from the queues when necessary, the additional space requirement can be kept at $O(m)$.

To summarize, Algorithm C works in $O(nm/\log m)$ time and $O(m)$ additional space. Like Algorithm B, it is an on-line algorithm and needs only $O(m)$ time to process each character.

6 Algorithms D and DB

The algorithms in the previous sections work in $O(m)$ extra space. Faster episode matching is possible if more space is available. Even $O(n)$ text scanning time can be achieved but this may need exponential space. In this section, we will describe an algorithm that works already in moderate space but can take advantage of what space is available. It is based on the "Four Russians" technique that is well-known in approximate string matching [7,13].

Algorithm B computes the full table $s[0..m]$ for every text position, even though knowing just $s[m]$ would be enough. The first algorithm of this section, Algorithm D, computes the full table $s[0..m]$ only at every hth position (for a suitably chosen h) and just $s[m]$ at other positions.

Let $s_i[j]$ denote the value of $s[j]$ at position i. Assume that $s_{kh}[0..m]$ is known for some k. The value $s_{(k+1)h}[j]$ must be one of the values in the nondecreasing sequence $\sigma_j[1..h+1] = (s_{kh}[j], s_{kh}[j-1], \ldots, s_{kh}[1], kh+1, kh+2, \ldots, (k+1)h-j+1)$, and which it is depends only on the substring $T[kh+1..(k+1)h]$. Similarly, assuming $i \leq h \leq m$, $s_{kh+i}[m]$ must be one of the values $\sigma^i[1..i+1] = (s_{kh}[m], s_{kh}[m-1], \ldots, s_{kh}[m-i])$ depending only on the substring $T[kh+1..kh+i]$.

Algorithm D starts by building a full trie of height h, that is, a trie containing all strings of length h in the transformed alphabet $\{0, \ldots, \langle P \rangle\}$. Each leaf representing a string S stores a table $\delta_S[1..m]$ such that $s_{(k+1)h}[j] = \sigma_j[\delta_S[j]]$

for $j = 1, \ldots, m$. Similarly, each internal node at depth i representing a string S stores the value δ^S such that $s_{kh+i}[m] = \sigma^i[\delta^S]$. The size of the augmented trie is $O(mc^h)$, where $c = \langle P \rangle + 1$ is the size of the transformed alphabet. The trie can be computed in $O(mc^h)$ time.

Armed with the trie, Algorithm D then scans the text and computes the table $s[0..m]$ at every hth position and $s[m]$ at every position reporting a new minimal substring whenever $s[m]$ changes. The scan works on-line and needs $O(n + nm/h)$ time.

Algorithm D

1. Build a full trie of height h.
2. Augment the trie with the δ values.
3. Using the trie, scan the text computing $s[0..m]$ at every hth position and $s[m]$ at every position.

The algorithm works in $O(n + mc^h + nm/h)$ time and $O(mc^h)$ additional space. If enough space is available, we can choose $h = \log_c n - \log_c \log_c n$, thus getting an algorithm that works in $O(n + nm/\log_c n))$ time and $O(nm/log_c n)$ additional space. If the additional space is limited to $s = o(nm/log_c n)$, the algorithm works in $O(n + nm/\log_c(s/m))$ time.

A large alphabet size c is bad for Algorithm D while it is good for Algorithm B. Algorithm DB combines these two to achieve lesser dependence on the alphabet size and better worst case time complexity. For suitably chosen l, Algorithm D does the matching for the prefix $P[1..l]$ and Algorithm B for the suffix $P[l+1..m]$.

Algorithm DB

for $i \leftarrow 1$ **to** n **do**

1. Execute a step of Algorithm D to compute $s_i[l]$.
2. Execute a step of Algorithm B to compute $s_i[l+1..m]$.

Algorithm B updates the entries $s[l+1..m]$ exactly as often as if it was computing the whole table. Its time complexity is, therefore,

$$O\left(n + \sum_{j=l+1}^{m} ms(P[1..j], T)\right) .$$

Let $c_l = \langle P[1..l] \rangle$. Using the technique of Inequality 1 to analyze the sum, we can distribute the costs evenly among the c_l different characters of $P[1..l]$. This gives $O(n(m-l)/c_l)$ as Algorithm B's contribution to the time complexity of Algorithm DB.

The combined algorithm works in $O(n + lc_l^h + nl/h + n(m-l)/c_l)$ time and $O(lc_l^h)$ additional space. With the time-optimal choice of h, the time is $O(n + nl/\log_{c_l} n + n(m-l)/c_l)$. We will choose l to minimize the time. In the worst case, we have $l \approx m/2$ and $c_l \approx \log n/\log\log n$. This gives an algorithm working in $O(n + nm \log\log n/\log n)$ time and $O(nm \log\log n/\log n)$ additional space. Similar analysis for the case where additional space is limited to $O(s)$ gives the running time $O(n + nm \log\log s/\log(s/m))$.

7 Approximation Algorithms for Problem 2

The problem considered in this section is to find the length w of the shortest substring of T that contains P. Both of the algorithms of this section return a value $\widehat{w}$ with the approximation guarantee $w \le \widehat{w} \le (1+\epsilon)w$, for an arbitrary $\epsilon > 0$.

The first algorithm, Algorithm AA, uses the simple text scan that was also used in Algorithm A. The basic operation is $\textsc{scan}(h, l)$ which works as follows. Starting from every hth position of the text T, the operation scans l characters forward looking for the episode P. The operation returns the smallest number of characters scanned from a starting position before finding an occurrence of P. If $\textsc{scan}(h, l)$ returns $\widehat{w}$, it is known that $\widehat{w} - h < w \le \widehat{w}$. If no occurrences were found, it must be that $w > l - h + 1$. The procedure works in $O(nl/h)$ time.

In the first part of Algorithm AA, a binary search is used for finding the $j \in \{0, \ldots, \lfloor \log(n/m) \rfloor\}$ for which $\textsc{scan}(2^j m, 2^{j+1}m - 1)$ finds an occurrence but $\textsc{scan}(2^{j-1}m, 2^j m - 1)$ does not. This requires $O(n \log\log(n/m))$ time. As a result, it is known that $2^{j-1}m < w \le 2^{j+1}m$.

Finally, Algorithm AA does $\textsc{scan}(h, 2^{j+1}m + h - 1)$. The return value $\widehat{w}$ then satisfies

$$2^{j-1}m < w \le \widehat{w} < w + h.$$

Selecting $h = 2^{j-1}m\epsilon$, we get $w \le \widehat{w} < (1+\epsilon)w$. This part requires $O(n/\epsilon)$ time.

Overall, Algorithm AA works in $O(n \log\log(n/m) + n/\epsilon)$ time. Like Algorithm A, it is alphabet independent, runs in $O(1)$ additional space, and is easily modified for other kinds of patterns.

The other approximation algorithm, Algorithm AB, is a simple modification of Algorithm B. Let $h = 1 + \lfloor m\epsilon \rfloor$. Line 4 of Algorithm B is replaced with

4′ **if** $(i-1) \bmod h = 0$ **then** $s[0] \leftarrow i$; $\textsc{waits}[p_1] \leftarrow \textsc{waits}[p_1] \cup \{1\}$

In other words, $s[0]$ is only changed at every hth position. These lines are the only place where new values enter the table s; otherwise values are just copied from one entry to another. Each entry $s[j]$ is the starting position of the latest minimal substring containing $P[1..j]$. With the modifications, the starting positions can be up to $h-1$ smaller than they would be in the original algorithm. As a result, the reported minimal substrings are longer than actual minimal substrings by at most $h-1$. Thus, the algorithm has $(1+\epsilon)$ approximation ratio.

Algorithm AB has the running time $O(n+u)$, where u is the number of times an entry of s changes. With the modifications, only $O(n/h)$ different values are entered into the table, which means that no entry can change more than $O(n/h)$ times. Therefore, $u = O(nm/h)$ and the running time is $O(n + n/\epsilon)$.

8 Concluding Remarks

We have presented a new class of string patterns, called episodes, and given several algorithms for two different episode matching problems. Despite their simple formulation, episode matching problems can be quite nontrivial as our algorithms demonstrate. Indeed, we have just scratched the surface of episode matching problems. In addition to the various problems that can be derived from the basic episode conditions, there are the generalizations, such as occurrence times and general episodes, that rise from the applications to event sequences [6,5]. Other aspects that we have mostly ignored in this paper include average case analysis and lower bounds.

9 Acknowledgements

Discussions with Mordecai Golin, Heikki Mannila and Esko Ukkonen have been helpful in writing this paper.

References

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. Z. Galil and K. Park: An improved algorithm for approximate string matching. *SIAM J. Comp.*, **19**(6) (Dec. 1990), 989–999.
3. G. M. Landau and U. Vishkin: Fast parallel and serial approximate string matching. *J. Algorithms*, **10**(2) (June 1989), 157–169.
4. J. H. van Lint and R. M. Wilson: *A Course in Combinatorics*. Cambridge University Press, 1992.
5. H. Mannila and H. Toivonen: Discovering frequent episodes in sequences. *Proc. 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, 146–151. AAAI Press 1996.
6. H. Mannila, H. Toivonen and A. I. Verkamo: Discovering frequent episodes in sequences. *Proc. 1st International Conference on Knowledge Discovery and Data Mining (KDD'95)*, 210–215. AAAI Press 1995.
7. W. J. Masek and M. S. Paterson: A faster algorithm for computing string edit distances. *J. Comput. System Sci.*, **20** (1980), 18–31.
8. S. B. Needleman and C. D. Wunsch: A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Molecular Biol.* **48** (1970), 443–453.
9. P. H. Sellers: The theory and computation of evolutionary distances: pattern recognition. *J. Algorithms*, **1**(4) (Dec. 1980), 359–373.
10. H. Toivonen: *Discovery of Frequent Patterns in Large Data Collections*. Ph.D. Thesis, Report A-1996-5, Department of Computer Science, University of Helsinki, 1996.
11. E. Ukkonen: Finding approximate patterns in strings. *J. Algorithms*, **6**(1) (May 1985), 132–137.
12. S. Wu, U. Manber: Agrep – a fast approximate pattern-matching tool. *Proc. Usenix Winter 1992 Technical Conference*, 153–162. Jan. 1992.
13. S. Wu, U. Manber and G. Myers: A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, **15**(1) (Jan. 1996), 50–67.

Appendix. Proof of Theorem 9

Let $P_1, P_2, \ldots, P_k$ be strings over alphabet Σ satisfying the following conditions.

Condition 1.

$$|P_1| + |P_2| + \cdots + |P_k| \leq m$$

Condition 2. For all $i = 1, \ldots, k$,

$$\langle P_i \rangle \geq 2 \quad \text{and} \quad |P_i| \log\langle P_i \rangle - \log\log\langle P_i \rangle \leq \log m - \log\log m$$

or

$$\langle P_i \rangle = 1 \quad \text{and} \quad |P_i| \leq \log m - \log\log m .$$

The pieces of P in the trie of Algorithm C satisfy these conditions.

Let TR be the trie build from the strings $P_1, P_2, \ldots, P_k$. We shall prove that Algorithm B for trie TR has time complexity $O(nm/\log m)$, where n is the length of the text. By Theorem 8, Algorithm B for trie TR has time complexity $O(n + |\Sigma| + pms(TR, T))$. After alphabet transformation, $|\Sigma| \leq m + 1$. It is, therefore, enough to show that $pms(TR, T) = O(nm/\log m)$.

We will start by bounding the size $|TR|$ of the trie.

Lemma 11. *$|TR| = O(\langle TR \rangle m/\log m)$, where $\langle TR \rangle$ is the number of different characters in TR.*

Proof. Let Σ_{TR} be the set of characters in TR and let Γ be a subset of Σ_{TR}. Consider those strings P_i that contain at least one of each of the characters in Γ and no characters in $\Sigma_{TR} \setminus \Gamma$. Let m_Γ be the total length of these strings and let TR_Γ be the trie build of them. Based on Condition 2, it can be shown that

$$|TR_\Gamma| = O\left(\frac{m_\Gamma \left(1 + \log \frac{m}{m_\Gamma}\right)}{\log m} \right) .$$

Clearly, $|TR| \leq \sum_{\Gamma \subseteq \Sigma_{TR}} |TR_\Gamma|$. The sum is maximized when

$$m_\Gamma = \frac{m}{2^{|\Sigma_{TR}|}} = \frac{m}{2^{\langle TR \rangle}}$$

for all Γ, giving the result. □

Next, we will analyze $pms(TR, T)$ using the amortized analysis technique from Section 4. Since there is one-to-one correspondence between nodes $V \setminus \{root\}$ and edges E in the trie TR, we can redefine $pms(TR, T)$ as

$$pms(TR, T) = \sum_{e \in E} ms(\textsc{str}(e), T) ,$$

where $\text{STR}(e)$ is the concatenation of the labels on the path from the root to e including the label of e itself. Let ℓ be a relabeling of the edges such that, for all $e \in E$, $\ell(e) \in \text{STR}(e)$. Then, following (1),

$$pms(TR,T) \leq \sum_{e \in E} ms(\ell(e),T) = \sum_{c \in \Sigma} |\{e \in E \mid \ell(e) = c\}| ms(c,T) \ . \quad (3)$$

Let L be the set of all such relabelings ℓ. For all $\Gamma \subseteq \Sigma_{TR}$, let $L_\Gamma \subseteq L$ be the set of relabelings such that $\ell(e) \in \Gamma \cup \{\text{LABEL}[e]\}$ for all $e \in E$. Let $L_\Gamma^{max} \subseteq L_\Gamma$ be the set of relabelings that maximize the number of edges e with $\ell(e) \in \Gamma$.

By Lemma 11, there exists a constant b such that $|TR| \leq b\langle TR \rangle m/\log m$ for large enough m. Let $q = \lceil bm/\log m \rceil$. Then

$$|TR| \leq \langle TR \rangle q \ . \quad (4)$$

We will prove the following lemma at the end of the appendix.

Lemma 12. *There exists a nonempty subset Γ of Σ_{TR} and a relabeling $\ell \in L_\Gamma^{max}$ such that no character of Γ appears more than q times in the relabelled trie TR.*

Consider now the trie TR relabelled with ℓ of Lemma 12. Since $\ell \in L_\Gamma^{max}$, every path from the root to a leaf can be divided into two parts, the first having only labels in $\Sigma_{TR} \setminus \Gamma$ and the second only labels in Γ. We call the latter parts the Γ-tails of trie TR.

Removing the Γ-tails from TR and the corresponding suffixes from the strings $P_1, \ldots, P_k$, we get a trie TR' build from strings $P'_1, \ldots, P'_k$. The strings $P'_1, \ldots, P'_k$ satisfy Conditions 1 and 2 since $|P'_i| \leq |P_i|$ and $\langle P'_i \rangle \leq \langle P_i \rangle$. Therefore, Lemmas 11 and 12 apply to TR', too, and there exist Γ' and ℓ' with the properties stated in Lemma 12.

Applying this procedure recursively until TR is empty, we get a partition $\Gamma, \Gamma', \Gamma'', \ldots$ of Σ_{TR} and the corresponding sequence of relabelings $\ell, \ell', \ell'', \ldots$. Combining the relabelings by restricting each ℓ to the corresponding Γ-tail, we have a relabeling of TR such that no character appears more than q times in the relabelled trie. Therefore, by (3),

$$pms(TR,T) \leq q \sum_{c \in \Sigma} ms(c,T) \leq qn \ .$$

Since $q = O(m/\log m)$, this completes the proof of the theorem.

The only thing left is to prove Lemma 12.

Proof (of Lemma 12). This proof is closely related to the maximum bipartite matching problem. Let G be a bipartite graph $(X \cup Y, F)$, where X contains a vertex x_e for each edge e in TR, and Y contains q vertices $y_c^1, y_c^2, \ldots, y_c^q$ for each character $c \in \Sigma_{TR}$. Vertex x_e is connected to the vertices $y_c^1, \ldots, y_c^q$ iff e can be relabelled with c. Note that, by (4), $|X| \leq |Y|$.

A matching M is a subset of F such that no vertex is an endpoint of more than one edge in M. Matching M is complete if it matches every $x \in X$. Given a matching M, let ℓ_M be the relabeling of trie TR with $\ell_M(e) = c$ if x_e is matched to one of the vertices $y_c^1, \ldots, y_c^q$, and $\ell_M(e) = \textsc{label}[e]$ otherwise. If M is complete, ℓ_M satisfies the conditions of the lemma when we choose $\Gamma = \Sigma_{TR}$.

Given a matching M, an alternating path is a path from a vertex $x \in X$ to a vertex $y \in Y$, where every even-numbered edge is in M and every odd-numbered edge (including the first and last) are in $F \setminus M$. An alternating path is an augmenting path if neither x nor y is matched in M. The classic result on bipartite matching is that a matching M is maximal if and only if there are no augmenting paths (see e.g. [4, Sect. 5]).

Suppose a maximal matching M is not complete. Let Y' be the set of vertices $y \in Y$ such that there is no alternating path to y from a non-matched vertex $x \in X$. All non-matched $y \in Y$ must be in Y'. Y' is nonempty, because there are non-matched vertices in Y, since $|X| \leq |Y|$.

Let X' be the set of $x \in X$ that are connected to some $y \in Y'$. By definition of Y', all $x \in X'$ are matched in M. Assume there is $x' \in X'$ that is matched to $y \in Y \setminus Y'$. By definition of X', x' is also connected to some $y' \in Y'$, and by definition of $Y \setminus Y'$, there is an alternating path from a non-matched vertex $x \in X \setminus X'$ to y. But then we could extend the alternating path from y to x' to y'. This is a contradiction. Therefore every $x \in X'$ is matched to $y \in Y'$.

Let G' be the graph G restricted to $X' \cup Y'$ and let M' be the matching M restricted to G'. By the above, M' is a complete matching for G'. For any $c \in \Sigma_{TR}$, $y_c^1, \ldots, y_c^q$ have the same connections in G and are, therefore, either all in Y' or all in $Y \setminus Y'$. Let Γ be the set of $c \in \Sigma_{TR}$ with y_c^i in Y'. Then X' represents the edges of TR that can be labelled with a character in Γ. Since all $x \in X'$ are matched in M', $\ell_{M'} \in L_\Gamma^{max}$, and no $c \in \Gamma$ appears more than q times in the trie TR relabelled with $\ell_{M'}$. $\square$