Dictionary Matching in a Stream

Raphaël Clifford[1], Allyx Fontaine[1], Ely Porat[2],
Benjamin Sach[1], and Tatiana Starikovskaya[1]

[1] University of Bristol, Department of Computer Science, Bristol, UK
[2] Bar-Ilan University, Department of Computer Science, Israel

Abstract. We consider the problem of dictionary matching in a stream. Given a set of strings, known as a dictionary, and a stream of characters arriving one at a time, the task is to report each time some string in our dictionary occurs in the stream. We present a randomised algorithm which takes $O(\log\log(k+m))$ time per arriving character and uses $O(k\log m)$ words of space, where k is the number of strings in the dictionary and m is the length of the longest string in the dictionary.

1 Introduction

We consider the problem of dictionary matching in a stream. Given a set of strings, known as a dictionary, and a stream of characters arriving one at a time, the task is to determine when some string in our dictionary matches a suffix of the growing stream. The dictionary matching problem models the common situation where we are interested in not only a single pattern that may occur but in fact a whole set of them.

The solutions we present will be analysed under a particularly strong model of space usage. We will account for all the space used by our algorithm and will not, for example, even allow ourselves to store a complete version of the input. In particular, we will neither be able to store the whole of the dictionary nor the streaming text. We now define the problem which will be the main object of study for this paper more formally.

Problem 1. In the dictionary-matching problem we have a set of patterns $\mathcal{P}$ and a streaming text $T = t_1 \dots t_n$ which arrives one character at a time. We must report all positions in T where there exists a pattern in $\mathcal{P}$ which matches exactly. More formally, we output all the positions x such that there exists a pattern $P_i \in \mathcal{P}$ with $t_{x-|P_i|+1} \dots t_x = P_i$. We must report an occurrence of some pattern in $\mathcal{P}$ as soon as it occurs and before we can process the subsequent arriving character.

If each of the k patterns in the dictionary had the same length m then we could straightforwardly deploy the fingerprinting method of Karp and Rabin [13] to maintain a fingerprint of a window of length m successive characters of the text. We can then compare this for each new character that arrives to a hash table of stored fingerprints of the patterns in the dictionary. In our notation this

N. Bansal and I. Finocchi (Eds.): ESA 2015, LNCS 9294, pp. 361–372, 2015.
DOI: 10.1007/978-3-662-48350-3_31

approach would require $O(k+m)$ words of space and constant time per arrival. However if the patterns are not all the same length this technique no longer works.

For a single pattern, Porat and Porat [17] showed that it is possible to perform exact matching in a stream quickly using very little space. To do this they introduced a clever combination of the randomised fingerprinting method of Karp and Rabin and the deterministic and classical KMP algorithm [14]. Their method uses $O(\log m)$ words of space and takes $O(\log m)$ time per arriving character where m is the length of the single pattern. Breslauer and Galil subsequently made two improvements to this method. First, they sped up the method to only require $O(1)$ time per arriving character and they also showed that it was possible to eliminate the possibility of false negatives, which could occur using the previous approach [3].

Our solution takes the single-pattern streaming algorithm of Breslauer and Galil [3] as its starting point. If we were to run this algorithm independently in parallel for each separate string in the dictionary, this would take $O(k)$ time per arriving character and $O(k \log m)$ words of space. Our goal in this paper is to reduce the running time to as close to constant as possible without increasing the total space. Achieving this presents a number of technical difficulties which we have to overcome.

The first such hurdle is how to process patterns of different lengths efficiently. In the method of Breslauer and Galil prefixes of power of two lengths are found until either we encounter a mismatch or a match is found for a prefix of length at least half of the total pattern size. Exact matches for such long prefixes can only occur rarely and so they can afford to check each one of these potential matches to see if it can be extended to a full match of the pattern. However when the number of patterns is large we can no longer afford to inspect each pattern every time a new character arrives.

Our solution breaks down the patterns in the dictionary into three cases: short patterns, long patterns with short periods, long patterns with long periods. A key conceptual innovation that we make is a method to split the patterns into parts in such a way that matches for all of these parts can be found and stitched together at exactly the time they are needed. We achieve this while minimising the total space and taking only $O(\log\log(k+m))$ time per arriving symbol.

A straightforward counting argument tells us that any randomised algorithm with inverse polynomial probability of error requires at least $\Omega(k \log n)$ *bits* of space, see for example [5]. Our space requirements are therefore within a logarithmic factor of being optimal. However, unlike the single-pattern algorithm of Breslauer and Galil, our dictionary matching algorithm can give both false positives and false negatives with small probability.

Throughout the rest of this paper, we will refer to the arriving text character as the arrival. We can now give our main new result which will be proven in the remaining parts of this paper.

Theorem 1. *Consider a dictionary $\mathcal{P}$ of k patterns of size at most m and a streaming text T. The streaming dictionary matching problem can be solved in*

$O(\log\log(k+m))$ time per arrival and $O(k\log m)$ words of space. The probability of error is $O(1/n)$ where n is the length of the streaming text.

1.1 Related Work

The now standard offline solution for dictionary matching is based on the Aho-Corasick algorithm [1]. Given a dictionary $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$, and a text $T = t_1 \ldots t_n$, let occ denote the number of matches and M denote the sum of the lengths of the patterns in $\mathcal{P}$, that is $M = \sum_{i=1}^{k} |P_i|$. The Aho-Corasick algorithm finds all occurrences of elements in $\mathcal{P}$ in the text T in $O(M+n+\text{occ})$ time and $O(M)$ space. Where the dictionary is large, the space required by the Aho-Corasick approach may however be excessive.

There is now an extensive literature in the streaming model. Focusing narrowly on results related to the streaming algorithm of Porat and Porat [17], this has included a form of approximate matching called parameterised matching [12], efficient algorithms for detecting periodicity in streams [11] as well as identifying periodic trends [10]. Fast deterministic streaming algorithms have also been given which provided guaranteed worst case performance for a number of different approximate pattern matching problems [7,8] as well as pattern matching in multiple streams [6].

The streaming dictionary matching problem has also been considered in a weaker model where the algorithm is allowed to store a complete read-only copy of the pattern and text but only a constant number of extra words in working space. Breslauer, Grossi and Mignosi [4] developed a real-time string matching algorithm in this model by building on previous work of Crochemore and Perrin [9]. The algorithm is based on the computation of periods and critical factorisations allowing at the same time a forward and a backward scan of the text.

1.2 Definitions

We will make extensive use of Karp-Rabin fingerprints [13] which we now define along with some useful properties.

Definition 1. *Karp-Rabin fingerprint function ϕ. Let p be a prime and r a random integer in $\mathbb{F}_p$. We define the fingerprint function ϕ for a string $S = s_1 \ldots s_\ell$ such that:*

$$\phi(S) = \sum_{i=1}^{\ell} s_i r^i \bmod p.$$

The most important property is that for any two equal length strings U and V with $U \neq V$, the probability that $\phi(U) = \phi(V)$ is at most $1/n^2$ if $p > n^3$. We will also exploit several well known arithmetic properties of Karp-Rabin fingerprints which we give in Lemma 1. All operations will be performed in the word-RAM model with word size $\Theta(\log n)$.

Lemma 1. *Let U be a string of size ℓ and V another string, then:*

- $\phi(UV) = \phi(U) + r^{\ell}\phi(V) \ mod \ p$,
- $\phi(U) = \phi(UV) - r^{\ell}\phi(V) \ mod \ p$,
- $\phi(V) = r^{-\ell}(\phi(UV) - \phi(U)) \ mod \ p$.

For a non-empty string x, an integer p with $0 < p \leq |x|$ is called a *period* of x if $x_i = x_{i+p}$ for all $i \in \{1, \ldots, |x| - p\}$. The period of a non-empty string x is simply the smallest of its periods. We will also assume that all logarithms are base 2 and are rounded to the nearest integer.

We describe three algorithms: $\mathcal{A}_1$ in Section 2 which handles short patterns in the dictionary, and $\mathcal{A}_{2a}$ and $\mathcal{A}_{2b}$ in Section 3 which deal with the long patterns. Theorem 1 is obtained by running all three algorithms simultaneously.

2 Short Patterns

Lemma 2. *There exists an algorithm $\mathcal{A}_1$ which solves the streaming dictionary matching problem and runs in $O(\log\log(k+m))$ time per arrival and uses $O(k \log m)$ space on a dictionary of k patterns whose maximum length is at most $2k \log m$.*

For very short patterns, shorter than $2 \log m$, we can straightforwardly construct an Aho-Corasick automaton [1]. The automaton occupies $O(k \log m)$ space and reports occurrences of short patterns in constant time per arrival. As the input alphabet may not be constant, at each node of the automaton we store the transition function using a static perfect hash table allowing constant time transitions. From now on, we continue under the assumption that all patterns are longer than $2 \log m$.

Our solution partitions each of the patterns into prefix/suffix pairs in multiple ways. For each pattern there is one partition for each $\ell \in [\log m, 2 \log m]$. Each suffix has length ℓ and is referred to as the tail. The prefix makes up the rest of the pattern and is referred to as the head. We partition each pattern into at most $\log m$ head/tail pairs, making a total of at most $k \log m$ heads overall.

The overall idea is to insert all heads into a data structure so that we can find potential matches in the stream efficiently. We will only look for potential matches every $\log m$ arrivals. We use the remaining at least $\log m$ arrivals before a full match can occur both to de-amortise the cost of finding head matches as well as to check whether the relevant tails match as well.

In order to look for matches with heads of patterns efficiently we put them into a compacted trie. To make use of the compacted trie that we build, we will need to be able to find the *exit node* of any string x efficiently. This is the deepest node in the trie which represents a prefix of x. Given a string x and the fingerprints of all its prefixes, the exit node of x can be found using a variant of binary search in $O(\log m)$ time. This variant of binary search is called 2-fattest binary search and was introduced by Djamal Bellazougui et al. [2] for this purpose.

The basic idea is to perform binary search on the length of the longest prefix of x which matches a string in the set using fingerprint comparisons. The problem

with this approach is that traditional binary search would require us to store the fingerprint of every prefix of every string corresponding to a node in the trie. The 2-fattest binary search preprocessing avoids this by carefully selecting a single prefix length for each node. These are chosen in such a way that for any x the binary search only makes comparisons with these preselected prefix lengths. Therefore we only store $O(k \log m)$ fingerprints. The details of 2-fattest binary search can be found in Section 4.1 of [2]. In their work a general 'signature function' is supported which in our work is implemented as the Karp-Rabin fingerprint function.

We can now describe Algorithm $\mathcal{A}_1$ assuming that all patterns are longer than $2 \log m$ but no longer than $2k \log m$. As a preprocessing step, we build the compacted trie for the reverse of the at most $k \log m$ heads and preprocess it to allow efficient computation of exit nodes. For regularly spaced indices of the text, we will use the compacted trie to find the longest head that matches at each of these locations.

We will also augment the compacted trie during preprocessing so that we can support a second operation which will allow us to extend head matches into full matches. We mark each node labelled by a head with a colour representing the fingerprint of the corresponding tail. In the end, each node may be marked by several colours, and the total number of colours will be $k \log m$. On top of the trie we build a coloured-ancestor data structure [16]. This occupies $O(k \log m)$ space and supports $\mathsf{Find}(u, c)$ queries in $O(\log \log(k \log m)) = O(\log \log(k + m))$ time, where $\mathsf{Find}(u, c)$ is the lowest ancestor of a node u marked with colour c. We will use the coloured-ancestor queries to extend a matching head into the longest possible match with a whole pattern by using the fingerprints of different tails as queries.

At all times we maintain a circular buffer of size $2k \log m$ which holds the fingerprints of the most recent $2k \log m$ prefixes of the text. Let i be an integer multiple of $\log m$. For each such i, we query the trie with a string $x = t_i \dots t_{i-2k \log m+1}$. Note that for each prefix of x we can compute its fingerprint in $O(1)$ time with the help of the buffer. The query returns the exit node $e(x)$ of x in $O(\log m)$ time, which is used to analyse arrivals in the interval $[i + \log m, i + 2 \log m]$. This exit node corresponds to the longest head that matches ending at index i. The $O(\log m)$ cost of performing the query is deamortised during the interval $(i, i + \log m]$.

For each arrival t_ℓ, $\ell \in (i + \log m, i + 2 \log m]$ we compute the fingerprint ϕ of $t_{i+1} \dots t_\ell$. This can be done in constant time as we store the last $2k \log m \geq 2 \log m$ fingerprints. If $\mathsf{Find}(e(x), \phi)$ is defined, ℓ is an endpoint of a whole pattern match and we report it. Otherwise, we proceed to the next arrival. The overall time per arrival is therefore dominated by the time to perform the coloured-ancestor queries which is $O(\log \log(k + m))$.

We remark that the algorithm can be extended to permit patterns of length at most $4k \log m$ (instead of $2k \log m$) without affecting the time or space complexity. Moreover, if there are several possible patterns that match for a given

arrival, the algorithm reports the longest such pattern. These two properties will be needed when we describe Algorithm $\mathcal{A}_{2b}$ in Section 3.2.

3 Long Patterns

We now assume that all the patterns have length greater than $2k \log m$. We distinguish two cases according to the periodicity of those patterns: those with short period and those with long period. Hereafter, to distinguish the cases, we use the following notation. Let $m_i = |P_i|$ and Q_i be the prefix of P_i such that $|Q_i| = m_i - k \log m$. Let ρ_{Q_i} be the period of Q_i. The remaining patterns are then partitioned in two disjoint groups of patterns, those with $\rho_{Q_i} < k \log m$ and those with $\rho_{Q_i} \geq k \log m$. We describe two algorithms: $\mathcal{A}_{2a}$ and $\mathcal{A}_{2b}$, one for each case respectively. Finally, the overall solution is then to run all three algorithms $\mathcal{A}_1$, $\mathcal{A}_{2a}$, $\mathcal{A}_{2b}$ simultaneously to obtain Theorem 1.

3.1 Algorithm $\mathcal{A}_{2a}$: Patterns with Short Periods

This section gives an algorithm for a dictionary of patterns $\mathcal{P} = P_1, \ldots, P_k$ such that $m_i \geq 2k \log m$ and $\rho_{Q_i} < k \log m$. Recall that Q_i is the prefix of P_i of length $m_i - k \log m$ and ρ_{Q_i} is the period of Q_i. The overall idea for this case is that if we can find enough repeated occurrences of the period of a pattern then we know we have almost found a full pattern match. As the pattern may end with a partial copy of its period we will have to handle this part separately. The main technical hurdle we overcome is how to process different patterns with different length periods in an efficient manner.

We define the tail of a pattern P_i to be its suffix of length $2k \log m$. Observe that a P_i match occurs if and only if there is a match of Q_i followed by a match with the tail of P_i.

Let K_i be the prefix of Q_i of length $k \log m$. Further observe that Q_i can only match if there is a sequence of $\left\lfloor \frac{|Q_i|-|K_i|}{\rho_{Q_i}} + 1 \right\rfloor$ occurrences of K_i in the text, each occurring exactly ρ_{Q_i} characters after the last. This follows immediately from the fact that K_i has length $k \log m$ and Q_i has period $\rho_{Q_i} < k \log m$.

We now describe algorithm $\mathcal{A}_{2a}$ which solves this case. At all times we maintain a circular buffer of size $2k \log m$ which holds the fingerprints of the most recent $2k \log m$ prefixes of the text. That is, if the last arrival is t_ℓ, then the buffer contains the fingerprints $\phi(t_1 \ldots t_{\ell-2k \log m+1})$, $\ldots$, $\phi(t_1 \ldots t_\ell)$.

To find K_i matches, we store the fingerprint $\phi(K_i)$ of each distinct K_i in a static perfect hash table. By looking up $\phi(t_{\ell-k \log m+1} \ldots t_\ell)$ we can find whether some K_i matches in $O(1)$ time. For each distinct K_i we maintain a list of recent matches stored as an arithmetic progression. Each time we find a new match with K_i we check whether it is exactly ρ_{Q_i} characters from the last match. If so we include it in the current arithmetic progression. If not, then we delete the current progression and start a new progression containing only the latest match. Note that $K_i = K_j$ implies that $\rho_{Q_i} = \rho_{Q_j}$ and therefore there is no ambiguity in the description.

We store the fingerprint of each tail in another static perfect hash table. For each arrival t_ℓ we use this hash table to check whether $\phi(t_{\ell-2k\log m+1}\dots t_\ell)$ matches the fingerprint of some tail. This takes $O(1)$ time per arrival.

Assume that the tail of some P_i matched. We will justify below that we can assume that each tail corresponds to a unique P_i. It remains to decide whether this is in-fact a full match with P_i. This is determined by a simple check, that is whether the current arithmetic progression for K_i contains at least $\left\lfloor \frac{|Q_i|-|K_i|}{\rho_{Q_i}} + 1 \right\rfloor$ occurrences.

Lemma 3. *Algorithm $\mathcal{A}_{2a}$ takes $O(1)$ time per character and uses $O(k\log m)$ space.*

Proof. The algorithm stores two hash tables, each containing $O(k\log m)$ fingerprints as well as $O(k)$ arithmetic progressions. The total space is therefore $O(k\log m)$ as claimed. The time complexity of $O(1)$ per character follows by the use of static perfect hash tables (which are precomputed and depend only on $\mathcal{P}$).

We first prove the claim that each tail corresponds to a unique P_i. To this end, we assume in this section that no pattern contains another pattern as a suffix. In particular, any such pattern can be deleted from the dictionary during the preprocessing stage as it does not change the output. This implies the claim that each P_i has a distinct tail because the tail contains a full period of P_i.

The correctness follows almost immediately from the algorithm description via the observation that each Q_i is formed from $\left\lfloor \frac{|Q_i|-|K_i|}{\rho_{Q_i}} + 1 \right\rfloor$ repeats of K_i followed by a prefix of K_i. We check explicitly whether there are sufficient repeats of K_i in the text stream to imply a Q_i match. While we do not check explicitly that either final prefix of K_i is a match or that the full P_i matches, this is implied by the tail match. This is because the tail has length $2k\log m$ and hence includes the final prefix of K_i and the last $k\log m$ characters of P_i (those in P_i but not in Q_i). □

3.2 Algorithm $\mathcal{A}_{2b}$: Patterns with Long Periods

Consider a dictionary $\mathcal{P}$ in which the patterns are such that $m_i \geq 2k\log m$ and $\rho_{Q_i} \geq k\log m$. Let us define k to be number strings in this dictionary. We can now describe Algorithm $\mathcal{A}_{2b}$. Recall that Q_i is the prefix of P_i s.t. $|Q_i| = m_i - k\log m$. For each pattern P_i, we define $P_{i,j}$ to be the prefix of P_i with length 2^j, $1 \leq 2^j \leq m_i - 2k\log m$.

We will first give an overview of an algorithm that identifies $P_{i,j}$ matches in $O(\log m)$ time per arrival. With the help of $\mathcal{A}_1$ and $\mathcal{A}_{2a}$ we will speed it up to achieve an algorithm with $O(\log\log(k+m))$ time per arrival. The algorithm will identify the matches with a small delay up to $k\log m$ arrivals. We then show how to extend $P_{i,j}$ to Q_i matches. This stage will still report the matches after they occur. Finally we show how to find whole pattern matches in the stream using the Q_i matches while also completely eliminating the delay in the reporting of these matches. In other words, any matches for whole patterns will be reported as soon as they occur and before the next arrival in the stream as desired.

$O(\log m)$-time Algorithm. We define a logarithmic number of *levels*. Level j will represent all the matches for prefixes $P_{i,j}$. We store only *active* prefix matches, that still have the potential to indicate the start of full matches of a pattern in the dictionary. This means that any match at level j whose position is more than 2^{j+1} from the current position of an arrival is simply removed. We will use the following well-known fact.

Fact 2 (Lemma 3.2[3]). *If there are at least three matches of a string U of length 2^j in a string V of length 2^{j+1}, then positions of all matches of U in V form an arithmetic progression. The difference of the progression is equal to the length of the period of U.*

It follows that if there are at least three active matches for the same prefix at the same level, we can compactly store them as a *progression* in constant space. Consider a set of distinct prefixes of length 2^j of the patterns in $\mathcal{P}$. For each of them we store a progression that contains:

(1) The position `fp` of the first match;
(2) The fingerprint of $t_1 \dots t_{\mathtt{fp}}$;
(3) The fingerprint of the period ρ of the prefix;
(4) The length of the period ρ of the prefix;
(5) The position `lp` of the last match.

With this information, we can deduce the position and the fingerprint of the text from the start to the position of any active match of the prefix. Moreover, we can add a new match or delete the first match in a progression in $O(1)$ time.

We make use of a perfect hash table $\mathcal{H}$ that stores the fingerprints of all the prefixes of the patterns in $\mathcal{P}$. The keys of $\mathcal{H}$ correspond to the fingerprints of all the prefixes and the associated value indicates whether the prefix from which the key was obtained is a proper prefix of some pattern, a whole pattern itself, or both. Using the construction of [18], for example, the total space needed to store all the fingerprints and their corresponding values is $O(k \log m)$.

When a character t_ℓ of the text arrives, we update the current position and the fingerprint of the current text. The algorithm then proceeds by the progressions over $\log m$ levels. We start at level 0. If the fingerprint $\phi(t_\ell)$ is in $\mathcal{H}$, we insert a new match to the corresponding progression at level 0.

For each level j from 0 to $\log m$, we retrieve the position p of the first match at level j. If p is at distance 2^{j+1} from t_ℓ, we delete the match and check if the fingerprint $\phi(t_p \dots t_\ell)$ is in $\mathcal{H}$. If it is and the fingerprint is a fingerprint of one of the patterns, we report a match (ending at t_ℓ, the current position of the text). If the fingerprint is in $\mathcal{H}$ and if it is a fingerprint of a proper prefix, then p is a plausible position of a match of a prefix of length 2^{j+1}. We check if it fits in the appropriate progression π at level $j+1$. (Which might not be true if the fingerprints collided). If it does, we insert p to π. If p does not match in π, we discard it and proceed to the next level.

As updating progressions at each level only takes $O(1)$ time, and there are $\log m$ levels, the time complexity of the algorithm is $O(\log m)$ per arrival. The space complexity is $O(k \log m)$.

$O(\log\log(k+m))$-time Algorithm. We will follow the same level-based idea. To speed up the algorithm, we will consider prefixes $P_{i,j}$ with short and long periods separately. The number of matches of the prefixes with short periods can be big, but we will be able to compute them fast with the help of $\mathcal{A}_1$ and $\mathcal{A}_{2a}$. On the other hand, matches of the prefixes with long periods are rare, and we will be able to compute them in a round robin fashion.

Let $\rho_{i,j}$ be the period of $P_{i,j}$. We first build a dictionary D_1 containing at most one prefix for each P_i. Specifically, containing the largest $P_{i,j}$ with the period $\rho_{i,j} < k\log m$ and $2k\log m \leq |P_{i,j}| \leq m_i - 2k\log m$. If no such $P_{i,j}$ exists we do not insert a prefix for P_i. This dictionary is processed using a modification of algorithm $\mathcal{A}_{2a}$ which we described in Section 3.1. The modification is that when a text character t_ℓ arrives, the output of the algorithm identifies the longest pattern in D_1 which matches ending at t_ℓ or 'no match' if no pattern matches. This is in contrast to $\mathcal{A}_{2a}$ as described previously where we only outputted whether some pattern matches. The modification takes advantage of the fact that prefixes in D_1 all have power-of-two lengths and uses a simple binary search approach over the $O(\log m)$ distinct pattern lengths. This increases the run-time of $\mathcal{A}_{2a}$ to $O(\log\log m)$ time per arrival.

Whenever a match is found with some pattern in D_1, we update the match progression of the reported pattern (but not of any of its suffixes that might be in D_1). Importantly, we will still have at most two progressions of active matches per prefix because of the following lemma and corollary.

Lemma 4. *Let $P_{i,j}, P_{i',j'}$ be two prefixes in D_1 and suppose that $P_{i,j}$ is a suffix of $P_{i',j'}$. The periods of $P_{i,j}, P_{i',j'}$ are equal.*

Proof. Assume the contrary. Then $P_{i,j}$ has two periods: $\rho_{i,j}$ and $\rho_{i',j'}$ (because it is a suffix of $P_{i',j'}$). We have $\rho_{i,j} + \rho_{i',j'} < 2k\log m \leq |P_{i,j}|$. By the periodicity lemma (see, e.g., [15]), $\rho_{i,j}$ is a multiple of $\rho_{i',j'}$. But then $P_{i,j}$ is periodic with period $\rho_{i',j'} < \rho_{i,j}$, a contradiction. □

Corollary 1. *Let $P_{i,j}$, $P_{i',j'}$, and $P_{i'',j''}$ be prefixes in D_1. Suppose that $P_{i,j}$ is a suffix of $P_{i',j'}$ and simultaneously is a suffix of $P_{i'',j''}$. Then $P_{i',j'}$ is a suffix of $P_{i'',j''}$ (or vice versa).*

We now consider any P_i for which we did not find a suitable small period prefix. In this case it is guaranteed that there is a prefix $P_{i,j}$ with period longer than $k\log m$ but length at most $4k\log m$. We build another dictionary D_2 for each of these prefixes. We apply algorithm $\mathcal{A}_1$ and for each arrival t_ℓ return the longest prefix $P_{i,j}$ in D_2 that matches at it in $O(\log\log(k+m))$ time. We then need to update the match progression of $P_{i,j}$ as well as the match progressions of all $P_{i',j'} \in D_2$ that are suffixes of $P_{i,j}$. Fortunately, each of the prefixes in D_2 can match at most once in every $k\log m$ arrivals, because the period of each of them is long, meaning that we can schedule the updates in a round robin fashion to take $O(1)$ time per arrival.

We denote a set of all $P_{i,j}$ such that $\rho_{i,j} \geq k\log m$ by S. Any of these prefixes can have at most one match in $k\log m$ arrivals. Because of that and because

$|S| \leq k \log m$, we will be able to afford to update the matches in a round robin fashion.

We will have two update processes running in parallel. The first process will be updating matches of prefixes $P_{i,j} \in S$ such that $P_{i,j-1} \in S \cup D_2$. We consider one of these prefixes per arrival. If there is a match with $P_{i,j}$ in $[t_\ell - k \log m, t_\ell]$ then there must be a corresponding match with $P_{i,j-1}$ ending in $[t_{\ell-2^{j-1}-k \log m}, t_{\ell-2^{j-1}}]$. As $P_{i,j-1} \in S$, $\rho_{i,j} \geq k \log m$ so there is at most one match. We can determine whether this match can be extended into a $P_{i,j}$ match using a single fingerprint comparison as described in the $O(\log m)$-time algorithm. This is facilitated by storing a circular buffer of the fingerprints of the most recent $k \log m$ text prefixes.

The second process will be updating matches of prefixes $P_{i,j} \in S$ such that $P_{i,j-1} \in D_1$. Again, if there is a match with $P_{i,j}$ in $[t_\ell - k \log m, t_\ell]$ then there must be a corresponding match with $P_{i,j-1}$ ending in $[t_{\ell-2^{j-1}-k \log m}, t_{\ell-2^{j-1}}]$. However, the second process will be more complicated for two reasons. First, $P_{i,j-1}$ has a small period so there could be many $P_{i,j-1}$ matches ending in this interval. Second, the information about $P_{i,j-1}$ matches can be stored not only in the progressions corresponding to $P_{i,j-1}$, but also in the progressions corresponding to prefixes that have $P_{i,j-1}$ as a suffix. The first difficulty can be overcome because of the following lemma.

Lemma 5. *Consider any $P_{i,j}$ such that $\rho_{i,j-1} < k \log m \leq \rho_{i,j}$. Given a match progression for $P_{i,j-1}$, only one match could also correspond to a match with $P_{i,j}$.*

Proof. Let U be the prefix of $P_{i,j-1}$ of length $\rho_{i,j-1}$. That is, the substrings bounded by consecutive matches in the match progression for $P_{i,j-1}$ are equal to U. Suppose that $P_{i,j}$ starts with exactly r copies of U. Then we have $P_{i,j} = U^r V$ for some string V. Note that as $\rho_{i,j-1} < k \log m \leq \rho_{i,j}$, the string V cannot be a prefix of U. Then the only match in the progression which could match with $P_{i,j}$ is the r-th last one. □

To overcome the second difficulty, we use Corollary 1. It implies that prefixes in D_1 can be organized in chains based on the "being-a-suffix" relationship. We consider prefixes in each chain in a round robin fashion again. We start at the longest prefix, let it be $P_{i,j}$. At each moment we store exactly one progression initialized to the progression of $P_{i,j}$. If the progression intersects with $[t_{\ell-2^{j-1}-k \log m}, t_{\ell-2^{j-1}}]$, we identify the 'interesting' match in $O(1)$ time with the help of Lemma 5 and try to extend it as in the first process. We then proceed to the second longest prefix $P_{i',j'}$. If the stored progression intersects with $[t_{\ell-2^{j'-1}-k \log m}, t_{\ell-2^{j'-1}}]$, we proceed as for $P_{i,j}$. Otherwise, we update the progression to be the progression of $P_{i',j'}$ and repeat the previous steps for it. We continue this process for all prefixes in the chain.

From the description of the processes it follows that the matches for each $P_{i,j}$ (in particular, for the longest $P_{i,j}$ for each i) are outputted in $O(\log \log(k+m))$ time per arrival with a delay of up to $k \log m$ characters (i.e. at most $k \log m$ characters after they occur).

Finding Q_i Matches. We now show how to find Q_i matches using $P_{i,j}$ matches. If there is a match with Q_i in $[t_\ell - k\log m, t_\ell]$, there must be a match with the longest $P_{i,j}$ in $[t_\ell - 2^j - k\log m, t_\ell - 2^j]$. Because $|P_{i,j}| \leq m_i - 2k\log m$, this match has been identified by the algorithm and it is the first match in the progressions. We can determine whether this match can be extended into a Q_i match using a single fingerprint comparison.

Therefore the Q_i matches are outputted in $O(\log\log(k+m))$ time with a delay of up to $k\log m$ characters (i.e. at most $k\log m$ characters after they occur). We can then remove this delay using coloured ancestor queries in a similar manner to algorithm $\mathcal{A}_1$ as described below.

Finding Whole Pattern Matches and Removing the Delay. Up to this point, we have shown that we can find each Q_i match in $O(\log\log(k+m))$ time per arrival with a delay of at most $k\log m$ characters. Further we only report one Q_i match at each time. We will show how to extend these Q_i matches into P_i matches using coloured ancestor queries in $O(\log\log(k+m))$ time per arrival.

Build a compacted trie of the reverse of each string Q_i. The edges labels are not stored. The space used is $O(k)$. For each i we can find the reverse of Q_i in the trie in $O(1)$ time (by storing an $O(k)$ space look-up table).

The tail of each P_i is its $(k\log m)$-length suffix, i.e. the portion of P_i which is not in Q_i. Each distinct tail is associated with a colour. As there are at most $k\log m$ patterns, there are at most $k\log m$ colours. Computing the colour from the tail is achieved using a standard combination of fingerprinting and static perfect hashing. For each node in the tree which represents some Q_i we colour the node with the colour of the tail of P_i.

Whenever we find a Q_i match, we identify the place in the tree where the reverse of Q_i occurs. Recall that these matches may be found after a delay of at most $k\log m$ characters. A Q_i match ending at position $\ell - k\log m$ implies a possible P_i match at position ℓ. We remember this potential match until t_ℓ arrives.

More specifically when t_ℓ arrives we determine the node u in the trie representing the reverse of the longest Q_i which has a match at position $\ell - k\log m$. This can be done in $O(1)$ time by storing a circular buffer of fingerprints.

We now need to decide whether Q_i implies the existence of some P_j match. It is important to observe that as we discarded all but the longest such Q_i, we might find a P_j with $j \neq i$.

For each arrival t_ℓ, we compute the fingerprint ϕ of $t_{\ell-k\log m+1}\ldots t_\ell$. This can be done in constant time as we store the last $k\log m$ fingerprints. If $\mathsf{Find}(u,\phi)$ is defined, t_ℓ is an endpoint of a pattern match and we report it. Otherwise, we proceed to the next arrival.

Lemma 6. *Algorithm $\mathcal{A}_{2b}$ takes $O(\log\log(k+m))$ time per character. The space complexity of the algorithm is $O(k\log m)$.*

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(8), 333–340 (1975)
2. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: Searching a sorted table with O(1) accesses. In: SODA 2009: Proc. 20th ACM-SIAM Symp. on Discrete Algorithms, pp. 785–794 (2009)
3. Breslauer, D., Galil, Z.: Real-time streaming string-matching. ACM Transactions on Algorithms 10(4), 22 (2014)
4. Breslauer, D., Grossi, R., Mignosi, F.: Simple real-time constant-space string matching. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 173–183. Springer, Heidelberg (2011)
5. Broder, A.Z., Mitzenmacher, M.: Survey: Network applications of bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2003)
6. Clifford, R., Jalsenius, M., Porat, E., Sach, B.: Pattern matching in multiple streams. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 97–109. Springer, Heidelberg (2012)
7. Clifford, R., Sach, B.: Pseudo-realtime pattern matching: Closing the gap. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 101–111. Springer, Heidelberg (2010)
8. Clifford, R., Sach, B.: Pattern matching in pseudo real-time. Journal of Discrete Algorithms 9(1), 67–81 (2011)
9. Crochemore, M., Perrin, D.: Two-way string matching. J. ACM 38(3), 651–675 (1991)
10. Crouch, M.S., McGregor, A.: Periodicity and cyclic shifts via linear sketches. In: Goldberg, L.A., Jansen, K., Ravi, R., Rolim, J.D.P. (eds.) APPROX and RANDOM 2011. LNCS, vol. 6845, pp. 158–170. Springer, Heidelberg (2011)
11. Ergun, F., Jowhari, H., Sağlam, M.: Periodicity in streams. In: Serna, M., Shaltiel, R., Jansen, K., Rolim, J. (eds.) APPROX and RANDOM 2010. LNCS, vol. 6302, pp. 545–559. Springer, Heidelberg (2010)
12. Jalsenius, M., Porat, B., Sach, B.: Parameterized matching in the streaming model. In: STACS 2013: Proc. 30th Annual Symp. on Theoretical Aspects of Computer Science, pp. 400–411 (2013)
13. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260 (1987)
14. Knuth, D.E., Morris, J.H., Pratt, V.B.: Fast pattern matching in strings. SIAM Journal on Computing 6, 323–350 (1977)
15. Lothaire, M.: Algebraic Combinatorics on Words. Cambridge University Press (2002)
16. Muthukrishnan, S., Müller, M.: Time and space efficient method-lookup for object-oriented programs. In: SODA 1996: Proc. 7th ACM-SIAM Symp. on Discrete Algorithms, pp. 42–51 (1996)
17. Porat, B., Porat, E.: Exact and approximate pattern matching in the streaming model. In: FOCS 2009: Proc. 50th Annual Symp. Foundations of Computer Science, pp. 315–323 (2009)
18. Ružić, M.: Constructing efficient dictionaries in close to sorting time. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 84–95. Springer, Heidelberg (2008)