Real-Time Streaming String-Matching

Conference Paper *in* ACM Transactions on Algorithms · June 2011

DOI: 10.1007/978-3-642-21458-5_15 · Source: DBLP

2 authors, including:

Zvi Galil
Georgia Institute of Technology
232 PUBLICATIONS **7,987** CITATIONS

SEE PROFILE

Real-Time Streaming String-Matching *

Dany Breslauer[1] and Zvi Galil[2]

[1] University of Haifa, Haifa, Israel
[2] Georgia Institute of Technology, Atlanta, Georgia

Abstract. This paper presents a real-time randomized streaming string-matching algorithm that uses $O(\log m)$ space. The algorithm only makes one-sided small probability false-positive errors, possibly reporting phantom occurrences of the pattern, but never misses an actual occurrence.

1 Introduction

The string-matching problem is concerned with finding all exact occurrences of a pattern string $\mathcal{P}[1..m]$ in a text string $\mathcal{T}[1..n]$. Numerous algorithms exist, including algorithms that solve the problem in linear time, in real time, and even using only constant auxiliary space in addition to the input strings [3,5,6,7]. However, all these algorithms, including the on-line algorithms, require repeated access to the pattern or the text. In fact, if the pattern is considered part of the streamed input, without sufficient state space to remember the pattern or associated information, it is impossible to precisely identify occurrences of the pattern in the text.

The string matching problem is often viewed as a candidate elimination problem where initially all text positions are candidate occurrences of the pattern and an algorithm's task is to eliminate candidates and to verify which of the remaining text positions are actual occurrences. The classical Knuth-Morris-Pratt [7] algorithm proceeds by scanning the text and matching subsequent text symbols against the pattern. If a mismatch occurs, then the algorithm shifts the pattern ahead to the next viable text occurrence candidate. The shift is the smallest number of text positions that would align the pattern prefix that was matched thus far with the text, with another matching pattern prefix, skipping candidate occurrences that can be ruled out by the transitivity of the pattern's prefix self-overlap (also called *period*). The lengths of all such shifts are pre-computed in the pattern preprocessing phase and take up $O(m)$ space.

The Karp-Rabin [6] randomized string-matching algorithm deploys an entirely different approach. The algorithm computes a so-called *fingerprint* of a sliding text window of size m, the same length as the pattern, and compares this fingerprint to the fingerprint of the pattern, eliminating candidate occurrences with different fingerprints than the pattern's fingerprint. Their algorithm, however, requires access to the last m text symbols to slide the fingerprint window

* Partially supported by the European Research Council (ERC) project SFEROT and by the Israeli Science Foundation grants 35/05 and 347/09.

along the text. While the fingerprint functions always identify equal strings, different strings are usually mapped to different fingerprints, but with small probability, to identical fingerprints, possibly introducing erroneous *false-positive phantom* occurrences. Such phantom occurrences can be later verified against the text if both the pattern and text are readily accessible in memory.

Porat and Porat [8] recently gave a streaming-model string-matching algorithm that uses a combination of both the periodicity and the fingerprint approaches. Their one-pass streaming algorithm takes $O(\log m)$ time per symbol, or $O(n \log m)$ time overall, and uses only $O(\log m)$ space. Throughout this paper space refers to the number of $O(\log n)$ bit registers and neither the pattern nor any text segment is accessible after appearing in the input stream.

In addition to possibly reporting false-positive phantom occurrences inherent in fingerprinting, Porat and Porat's [8] algorithm may also commit with small probability *false-negative* errors, omitting actual occurrences of the pattern in the text (two-sided errors). Their algorithm also requires the period lengths and period fingerprints of the pattern and various pattern prefixes to be computed in the pattern preprocessing phase. However, no details were provided about how this information is computed. Note that while period lengths are often computed via straighforward application of string matching algorithms to match the pattern against itself, the streaming model's limitation present some obstacles.

In fact, independently of our work, Ergun, Jowhari and Salgan [1] recently studied the problem of computing the period length of a string in the streaming model. They describe an $O(m \log m)$ time one-pass streaming algorithm to compute the period length of a string using $O(\log m)$ space. Their algorithm, that finds the period only if the input string is *periodic* (the period is no longer than half of the string's length), builds on a simplified streaming string-matching algorithm with simpler pattern preprocessing requirements than Porat and Porat's [8] algorithm (still two-sided errors). Moreover, Ergun, Jowhari and Salgan [1] prove that $\Omega(m)$ space is required by any one-pass streaming algorithm that computes the period length of *non-periodic strings*, but two-passes suffice to reduce the space to $O(\log m)$. They also prove that $\Omega(\log m)$ space is required by any streaming string-matching algorithm, for certain choices of the pattern and text lengths.

This paper presents two streaming string-matching algorithms. The first, like Porat and Porat's [8] algorithm, takes $O(\log m)$ time per symbol and uses $O(\log m)$ space, but is conceptually much simpler and has two important advantages: (1) the algorithm only commits small probability false-positive errors and no false-negative errors; in particular it never misses an occurrence (one-sided errors), and, (2) the pattern preprocessing phase is a trivial real-time streaming algorithm that does not compute period lengths. The second algorithm is a *real-time* algorithm, namely worst-case constant-time per symbol, using the same $O(\log m)$ space, while maintaining the one-sided error and the simple real-time streaming pattern preprocessing. Our techniques can be used to speed up Ergun, Jowhari and Salgan's [1] periodicity streaming algorithm to $O(m)$ time.

2 Fingerprints and Periods

Porat and Porat [8] used Karp and Rabin's [6] fingerprints, defining for a prime p, a random integer $r \in \mathcal{F}_p$, and a string $s = s_1 s_2 \cdots s_l$ over the alphabet $\mathcal{F}_p$ ($\mathcal{F}_p$ is the field of integers modulo the prime p) the fingerprint function $\phi_{r,p}(s) = \sum_{i=1}^{l} s_i r^i \ mod\ p$. The *error probability*, the probability that two different strings share the same fingerprint, can be bounded as follows.

Theorem 1. *Let u and v be two different strings of length l, where $l \leq n$ and $p \in \theta(n^{2+\alpha})$, for some $\alpha \geq 0$. Then, the probability that fingerprints $\phi_{r,p}(u) = \phi_{r,p}(v)$, for a random $r \in F_p$, is smaller than $\frac{1}{n^{1+\alpha}}$.*

The fingerprint function can be arithmetically manipulated to compute the fingerprint of two concatenated strings, requiring only the fingerprints and string lengths and not the concatenated strings themselves. (The powers r^k and r^{-k} can be maintained together with the corresponding fingerprints and updated with the fingerprint operations, and do not need to be computed every time.)

Lemma 1. *One can compute the fingerprint of concatenated strings u and v as:*

$$\phi_{r,p}(uv) = \phi_{r,p}(u) + r^k \phi_{r,p}(v) \ mod\ p \qquad uv = u_1 u_2 \cdots u_k v_1 v_2 \cdots v_l. \tag{1}$$

The last Lemma can be used to *cancel out* the fingerprint of the prefix u from the fingerprint of the concatenated string uv to obtain the fingerprint of $v = u^{-1}(uv)$, an operation referred to as *sliding* by Porat and Porat [8]. Similarly, one can also cancel out the fingerprint of the suffix v to get the fingerprint of $u = (uv)v^{-1}$.

Corollary 1. *To extract the fingerprints of u or v from the fingerprint of uv:*

$$\phi_{r,p}(v) = r^{-k}(\phi_{r,p}(uv) - \phi_{r,p}(u)) \ mod\ p. \tag{2}$$

$$\phi_{r,p}(u) = \phi_{r,p}(uv) - r^k \phi_{r,p}(v) \ mod\ p. \tag{3}$$

The formulae above will be used to maintain the running fingerprints of multiple text blocks starting at various locations of interest and ending at the current text symbol, without having to update all such fingerprints with every text symbol. Instead, the algorithm will maintain one running fingerprint for the text prefix from the very beginning of the text and up to the current text symbol, and only update this one fingerprint with each input text symbol. Now, by keeping for each location of interest the static fingerprint for the text prefix up to that location, the algorithm can obtain the running fingerprint of the text block starting at that location up to and including the current text symbol:

Lemma 2. *The fingerprint of the text block starting at some location of interest and ending at the current text symbol can be computed in constant time whenever needed.*

Properties of periodic strings are often used in efficient string algorithms. A string u is *a period* of a string w if w is a prefix of u^k for some k, or equivalently if w is a prefix of uw. The shortest period of w is called *the period* of w and w is called *periodic* if it is at least twice as long as its period. Consider prefixes of the pattern of increasing length. If u is a prefix and v is a longer prefix, the period of u is said to continue in v if u and v have the same period and otherwise the period of u terminates in v. The following Theorem is due to Fine and Wilf [2]. The next two lemmas are proved using this theorem.

Theorem 2. *If a string u has periods of length p and q, and its length $|u| \geq p + q - \gcd(p, q)$, then u also has a period of length $\gcd(p, q)$.*

3 The $O(n \log m)$ Time Algorithm

This section describes an $O(n \log m)$ time on-line streaming string-matching algorithm, introducing the basic concepts which are refined in the next section to obtain a real-time algorithm. The algorithm runs $\lceil \log_2 m \rceil$ simultaneous stages that filter the remaining viable occurrences. Each stage requires constant space, and takes constant time per input symbol, adding up to $O(\log m)$ time per input symbol, $O(n \log m)$ time overall, and $O(\log m)$ total space. The pattern preprocessing is trivial. It computes a sequence $\mathcal{P}_i$ of $\lceil \log_2 m \rceil$ increasing prefixes of the pattern $\mathcal{P}[1..m]$, and records their fingerprints, where $|\mathcal{P}_i| = 2^i$, and if m is not a power of 2, adding the last $\mathcal{P}_k = \mathcal{P}[1..m]$, $k = \lceil \log_2 m \rceil$. The $\lceil \log_2 m \rceil$ fingerprints are stored in $O(\log m)$ space.

The algorithm maintains and updates viable occurrences of the pattern while the text is being streamed on-line. A *viable occurrence (VO)* is a position in the text were an occurrence has not been ruled out. The *block of the VO* is the block that starts at the VO and ends at the currently last symbol of the text. Each VO belongs to some stage number i, such that the algorithm has verified earlier that the fingerprint of the text block of length $|\mathcal{P}_i|$ starting at the VO is equal to the fingerprint of pattern prefix $\mathcal{P}_i$, but there are insufficient text symbols yet to verify if this VO belongs to stage number $i + 1$. As soon as there are sufficient text symbols, $|\mathcal{P}_{i+1}|$ to be precise, to promote a VO to the next stage (always the first, longest VO in the stage), the fingerprint of the block of the VO is compared to the pre-computed fingerprint of the pattern prefix $\mathcal{P}_{i+1}$ and the VO either gets promoted to the next stage or is eliminated. Clearly, an occurrence of each of the pattern prefixes $\mathcal{P}_i$ must start at each occurrence of the pattern and VOs eliminated this way cannot be occurrences of the whole pattern.

When the algorithm maintains a VO it maintains the position and the fingerprint of the block of the VO. The $O(n \log m)$ algorithm maintains this fingerprint directly. The real-time algorithm will maintain the fingerprint of the prefix of the text up to the VO. This fingerprint is the running fingerprint of the text when this position was reached. When needed the real-time algorithm gets the fingerprint of the block of the VO from the running fingerprint of the text and the static fingerprint of the VO using Corollary 1.

Each text position is initially considered a VO. As soon as the next position is reached, the one text symbol fingerprint is verified against the fingerprint of the pattern prefix $\mathcal{P}_0$, before the new VO may enter stage number 0. Note that VOs that start earlier in the text always correspond to longer text blocks, and therefore belong to the same or higher numbered stages. One can envision the VOs climbing the stage ladder from one stage to the next or falling off the ladder in case of fingerprint mismatch, up to the ultimate stage that verifies the fingerprint of the full pattern $\mathcal{P}_k = \mathcal{P}[1..m]$. Since all text positions are considered VOs and are only eliminated as a consequence of fingerprint mismatch, the algorithm commits no false-negative errors.

Similarly to Galil's [4] parallel string-matching algorithm, multiple VOs that get too crowded in some stage imply that there must be a periodic pattern prefix. Specifically, if there are at least three VOs at the same stage number i, then the pattern prefix $\mathcal{P}_i$ must be periodic, all the VOs in this stage must form an arithmetic progression whose difference is the period length of $\mathcal{P}_i$, and these VOs can be represented compactly and processed efficiently. However, there is one important caveat requiring more caution. The streaming algorithm compares fingerprints and not actual strings, and therefore different strings may be identified by the same fingerprint, conflicting with the periodicity *implied* by string equality. The algorithm may conclude that some fingerprint false-match error must have occurred since the periodicity properties have been violated, without precisely identifying the culprit fingerprint error.

Note that the algorithm only uses periodicity to facilitate the space efficient representation and never to eliminate any VO. In case that the algorithm discovers such low probability false-positive error, it must make some hard choices to remain within its strict space bounds while making sure that only false-positive errors are reported and no actual occurrences are omitted. The algorithm will discard some VOs that can not be compactly represented via the periodic arithmetic progression, essentially throwing them off the stage ladder. However, the algorithm will report all these discarded VOs as occurrences of the pattern so that no occurrences are missed. Note that by allowing extra space to store more individual VOs or arithmetic progressions, the algorithm could continue to examine these VOs. The expected space would still be $O(\log m)$.

Recall that the offending VO that revealed the fingerprint-periodicity inconsistency is in the process of being promoted from some stage to the next. To simplify the presentation and avoid cascading the effects of discarded VOs on higher numbered stages, the algorithm will discard and report all earlier VOs (in equal or higher numbered stages), excluding the offending VO and the last VO in the arithmetic progression, and keep all VOs beyond these two since the limiting factor here is the algorithm's ability to compactly store and process all the VOs. The up to $O(\log m)$ discarded arithmetic progressions will be compactly written to the output rather than spelled out individually to remain within the $O(\log m)$ bounds.

Thus, the algorithm might now report two classes of erroneous pattern occurrences, those *phantom* occurrences that passed through the entire stage lad-

der and eventually had their fingerprint verified against the fingerprint of the whole pattern, and those VOs that were thrown off the stage ladder due to some non-specific fingerprint false-match errors conflicting with the implied periodicity and keeping the algorithm from compactly representing crowded VOs. The error probability in both cases is small, since it is either due to fingerprint false-match of the whole pattern (and some stage prefixes) or a detected fingerprint-periodicity conflict that must be due to fingerprint false-match of some pattern prefix. The algorithm is summarized in Figure 1 where all stages are executed in increasing order for each input symbol.

1. Extend the fingerprints of the blocks of the VOs by the current text symbol. (There is sufficient time to do this directly without Lemma 2.)
2. If the block of the first, longest, VO in the stage has precisely $|\mathcal{P}_{i+1}|$ text symbols, then remove this VO from stage number i, and compare the fingerprint of its block to the fingerprint of the pattern prefix $\mathcal{P}_{i+1}$ as candidate for stage number $i+1$. The next VO in stage number i, if any, becomes the first VO in the stage.
   - If the fingerprints match, promote the VO to stage number $i+1$.
   - VOs that get promoted to the ultimate stage matched the fingerprint of the whole pattern and are reported as occurrences of the whole pattern. They do not need to be stored.
3. To initialize, each text symbol's fingerprint that is equal to the fingerprint of $\mathcal{P}_0$ adds to stage number 0 a new VO starting at that text position.

Fig. 1: Stage number i of the $O(n \log m)$ algorithm.

Lemma 3. *Let u and v be strings such that v contains at least three occurrences of u. Let $t_1 < t_2 < \ldots < t_h$ be the locations of all occurrences u in v, and assume that $t_{i+2} - t_i \leq |u|$, for $i = 1, \cdots, h-2$ and $h \geq 3$. Then this sequence must form an arithmetic progression with difference $d = t_{i+1} - t_i$, for $i = 1, \ldots, h-1$, that is equal to the period length of u.*

To get the required time and space bounds one has to compactly represent and efficiently process the multiple VOs that might accumulate in the same stage. Multiple VOs imply periodicity, and periodicity properties are used to represent the VOs and their associated information. The following Lemma follows from Lemma 3.

Lemma 4. *Suppose that there are at least three VOs in stage number i. If these are actual occurrences of the pattern prefix $\mathcal{P}_i$, then these VOs must form an arithmetic progression with difference equal to the period length of $\mathcal{P}_i$.*

If there are one or two VOs in a stage, these VOs are stored directly (the position and the fingerprint). If there are three or more, these VOs should form an arithmetic progression. The algorithm stores the first and last VO, the difference

between the positions of the first and second VOs (the difference of the arithmetic progression) and the fingerprint of the block between these two positions (which should be equal to the period of $\mathcal{P}_i$), altogether constant space. The locations and the fingerprints of the middle of the arithmetic progression can be verified and reconstructed as follows:

Lemma 5. *Let $t_1, t_2, \ldots, t_h$ be all the VOs in stage number i, and assume that these are all actual occurrences of $\mathcal{P}_i$. Then it is possible to represent the locations and fingerprints of all the text blocks starting at each of these VOs up to the current position by the location and fingerprints of the first and last VOs and the length and the fingerprint of the block between the first and second VOs.*

A progression is generated when a third VO joins the stage. Note that the data maintained for the progression can be easily computed. The only computation needed is to compute the fingerprint of the block between the first and the second VOs using Corollary 1. We can easily maintain the data related to the arithmetic progression in constant time: If a new VO joins the progression when promoted to stage i, it simply replaces the last one. When the first VO of the progression is promoted to stage $i+1$, the second one becomes the first and if there remain only two VOs in the progression, the progression stops to exist.

Recall that the algorithm did not compare actual symbols, but only fingerprints of the pattern prefixes $\mathcal{P}_i$, and there might be spurious VOs resulting from false-positive fingerprint errors. The algorithm only uses periodicity properties to verify the validity of the compact representation in stage number i when new VOs are added to the representation to ensure that the full representation can be faithfully reconstructed. There are two factors that need to be verified: (1) The VOs must form an arithmetic progression, and (2) the fingerprints can be reconstructed by Corollary 1. These properties are verified as soon as the third VO is added and every time another VO is added. If there are any problems during the verification, the algorithm concludes that some of the VOs are not actual occurrences due to small probability false-positive errors of some fingerprints. To stay within the time and space bounds, the algorithm will discard some of the VOs as outlined above, and to err only on the false-positive side, the algorithm will report the discarded VOs as actual occurrences.

1. The algorithm verifies that the VOs form an arithmetic progression. The difference of the arithmetic progression, the *implied period length* of $\mathcal{P}_i$, is set to the difference between the the first two VOs and verified against the rest. Normally one should expect the implied period length to be equal to the real period length of $\mathcal{P}_i$, but the algorithm does not know the real period length of $\mathcal{P}_i$ and only verifies that the VOs fall into some arithmetic progression.
2. The algorithm verifies that the text block that starts at the last VO in the stage and ends at the newly added VO has the same fingerprint of the text block between the first and the second VOs in the arithmetic progression, the *implied period fingerprint* which we maintain. This is required so that the algorithm does not introduce false-negative errors when extracting fingerprints from the compact representation via sliding by the implied period

fingerprint. Note that these text block fingerprints are extracted using either Equation 2 or 3 in Corollary 1 depending on whether VO fingerprints are maintained for text prefixes up to the VOs or for the blocks of the VOs.

Theorem 3. *The algorithm described above reports all occurrences of the pattern in the text in $O(\log m)$ time per text symbol using total $O(\log m)$ space. The algorithm may report false occurrences, and on occasions even detects that it had fingerprint errors, with probability $1/n^{\alpha'}$ for $\alpha' < \alpha$.*

Proof. Each stage number i, whether it has at most two VOs or more, takes constant-time to update one or two fingerprints with the current text symbol, and to discard or promote to stage number $i+1$ at most one VO. The space requirement for each stage is constant. Multiplying by $O(\log m)$ stages we get the desired bounds. The error probability is bounded by multiplying the probability of fingerprint comparison error by the up to $O(n \log m)$ comparisons made.

4 The Real-Time Algorithm

Observe that in the $O(n \log m)$ algorithm above, fingerprints were only used in stage number i when the length of the first (longest) block of a VO in the stage was equal to the length of the next stage's pattern prefix $|\mathcal{P}_{i+1}|$, to verify whether the VO may be promoted to the next stage or eliminated. The key to the *real time* implementation is in (1) eliminating repetitive verification due to small highly repetitive pattern prefixes (e.g. $aa \cdots aaa$), and, (2) evenly spreading the VO stage promotion verification to avoid contentious text locations that might require up to $\lceil \log_2 m \rceil$ verifications. Both problems are solved by using additional $O(\log m)$ space.

Galil's [3] real-time implementation of the Knuth-Morris-Pratt [7] algorithm will be used. Let $f = \lceil \log_2 \log_2 m \rceil + 1$ and consider the pattern prefix $\mathcal{P}_f$, such that $2 \log_2 m < |\mathcal{P}_f| \leq 4 \log_2 m$. The pattern preprocessing of the Knuth-Morris-Pratt [7] real-time variant will start as the pattern appears in the input stream, and will be stopped after the pattern prefix $\mathcal{P}_f$ was processed, having used only $|\mathcal{P}_f| = O(\log m)$ extra space.

Lemma 6. *Let u and v be prefixes of a string w, such that $|u| < |v|$, u is periodic and v is the shortest prefix of w such that the periodicity of u terminates in v. Then the period length of $v > |v| -$ the period length of u, and v is not periodic.*

The failure function of the Knuth-Morris-Pratt [7] algorithm and of Galil's [3] real-time implementation consists of the period of each prefix of the pattern. In the preprocessing we compute the period of each prefix of $\mathcal{P}_f$. If $\mathcal{P}_f$ is periodic, the preprocessing will examine further symbols of $\mathcal{P}$ until either the periodicity ends or the pattern ends. In case the periodicity ends, let π be the prefix up to and including the symbol where the periodicity ends. The period of each prefix of π that is not a prefix of P_f except for π is equal to the period of $\mathcal{P}_f$. Since π can be compressed, also its period can be easily computed by Galil's algorithm with

no additional space. If the pattern ends; i.e., the period of $\mathcal{P}_f$ is the period of $\mathcal{P}$, then Galil's [3] real-time string-matching algorithm can solve the streaming string-matching problem deterministically without any errors using $O(\log m)$ space.

The real-time string matching algorithm will then be used to match in real-time using $O(\log m)$ space occurrences of either a non-periodic $\mathcal{P}_f$ or π, which is non-periodic by Lemma 6. Let $\hat{f}$ be the largest integer such that $\mathcal{P}_{\hat{f}}$ is contained in this non-periodic pattern prefix (if $\mathcal{P}_f$ is not periodic $f = \hat{f}$). The occurrences found must be spaced by more than $|\mathcal{P}_{\hat{f}}|/2 \geq \log_2 m$ text positions apart, must start with $\mathcal{P}_{\hat{f}}$, and will be reported before sufficient symbols are available to verify $\mathcal{P}_{\hat{f}+1}$. These occurrences will be introduced at stage number $\hat{f}$ of the randomize $O(n \log m)$ algorithm in the previous section, skipping all the prior stages. Observe that no arithmetic progressions are forming at stage number $\hat{f}$ because of the spacing between the VOs.

Thus, the randomized real-time algorithm has two parts that are run alongside each other. Galil's [3] real-time string-matching algorithm that feeds stage number $\hat{f}$ of the following real-time adaptation of the $O(n \log m)$ algorithm from the previous section.

The real-time adaptation simulates the $O(n \log m)$ algorithm by maintaining a *cyclic buffer* $\mathcal{FP}[t]$ of size $s = \lceil \log_2 m \rceil$ that gives the running fingerprints of the *last* text prefixes of locations up to and including t. Specifically, the fingerprints for positions $t, t-1, \ldots, t-s+1$ are stored at $\mathcal{FP}[t \bmod s]$. The round robin algorithm rotates through the stages numbered $i = \hat{f}, \cdots, \lceil \log_2 m \rceil - 2$, in increasing order, processing one stage at each text location using the buffer for the correct fingerprints. Note that the stage processing is delayed by less than s steps and the fingerprint needed to test whether to promote the VO to the next stage is available in the buffer $\mathcal{FP}$. The following concerns need attention:

1. The simulated action may happen out of order with respect to the $O(n \log m)$ algorithm, and even in different order depending on the text location. Note that since VOs in the same stage are at least $\log_2 m$ apart and the delay in the test for promotion is less than $\log_2 m$, the order of tests for promotion is maintained inside each stage. The only case that the different order will lead to a non temporary different computation is the following: Assume in the real-time algorithm x, y are the first and second VOs in stage i and z was just promoted from stage $i-1$ to stage i as the third VO in the stage and it reveals an inconsistency with the periodicity. It is possible that in the $O(n \log m)$ algorithm x is promoted from stage i to stage $i+1$ before z is promoted to stage i and therefore in that algorithm y and z are the only VOs in stage i and there is no inconsistency. We can easily fix the order in such case to be the same by deleting x from stage i first, since stage i will be considered immediately after stage $i-1$ in the round robin algorithm and x can be promoted then. But in fact, this is not necessary since the algorithm is still correct with the different order.

2. The real-time on-line algorithm has to output the pattern occurrences immediately at their end, and delayed promotions to the last stage number are not acceptable. Such delays will be avoided by examining the last stage (number $\lceil \log_2 m \rceil - 1$) at every text location. In case P is longer than the P_i of the next to last stage by less than $\log_2 m$, then this stage too receives the same treatment.
3. Discarding and reporting VOs when some fingerprint-periodicity conflict is detected can take time. The simplest solution is to continue the rotation to larger number stages and discard the VOs in each stage until the last stage number $\lceil \log_2 m \rceil - 1$. Discarded arithmetic progressions will be compactly written to the output rather than spelled out individually to remain within the real-time bounds.

Theorem 4. *The algorithm described above reports all occurrences of the pattern in the text in constant time per text symbol using total $O(\log m)$ space. The algorithm may report false occurrences, and on occasion it even detects that it had fingerprint errors with probability $1/n^{\alpha}$.*

Proof. The algorithm updates the running fingerprint buffer with the current text symbol in constant time. Each delayed stage action can be properly done since the $\lceil \log_2 m \rceil$ fingerprint history is available in the buffer $\mathcal{FP}$. The space requirement for each stage is constant or $O(\log m)$ over all stages, and the overall space required for the Galil's [3] real-time string matching algorithm and for the buffer $\mathcal{FP}$ is $O(\log m)$. The error probability is bounded by multiplying the probability of fingerprint comparison error by the up to $O(n)$ comparisons made.

5 The Pattern Preprocessing

The $O(n \log m)$ time algorithm only requires the trivial preprocessing storing the fingerprints of the pattern prefixes $\mathcal{P}_i$. The real-time algorithm needs in addition to store either the short pattern prefix $\mathcal{P}_f$ and its failure function or (if $\mathcal{P}_f$ is periodic) the compressed versions of the longer prefix π and its failure function. The real-time algorithm will need to know the length of the pattern m or its order of magnitude.

Additional pattern preprocessing can be advantageous, though, to try to obtain a "better" fingerprint function that does not cause any fingerprint-periodicity conflict while matching the given pattern with a text string that is exactly equal to the pattern. (Conflicts would repeat in every occurrence of the pattern.) Such fingerprint function can be obtained by trying out several random seeds, either simultaneously while the pattern is streamed in or sequentially if the pattern is available for additional re-processing (i.e. in a nonstreaming fashion).

Theorem 5. *Given the fingerprint function and the pattern, if the pattern is fingerprint-periodicity conflict free, then when the streaming algorithm discards VOs in the text due to fingerprint-periodicity conflict, the discarded VOs do not need to be reported as potential occurrences.*

References

1. F. Ergun, H. Jowhari, and M. Salgan. Periodicity in Streams. Manuscript, 2010.
2. N.J. Fine and H.S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16:109–114, 1965.
3. Z. Galil. String Matching in Real Time. *J. Assoc. Comput. Mach.*, 28(1):134–149, 1981.
4. Z. Galil. Optimal parallel algorithms for string matching. *Inform. and Control*, 67:144–157, 1985.
5. Z. Galil and J. Seiferas. Time-space-optimal string matching. *J. Comput. System Sci.*, 26:280–294, 1983.
6. R.M. Karp and M.O. Rabin. Efficient randomized pattern matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, 1987.
7. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322–350, 1977.
8. B. Porat and E. Porat. Exact And Approxiamate Pattern Matching In The Streaming Model. In *Proc. 50th IEEE Symp. on Foundations of Computer Science*, pages 315–323, 2009.