Real-Time Streaming String-Matching

DANY BRESLAUER, Caesarea Rothchild Institute for Interdisciplinary Applications of Computer Science, University of Haifa, Israel
ZVI GALIL, College of Computing, Georgia Institute of Technology

This article 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 missing an actual occurrence.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Pattern Matching*; G.3 [**Probability and Statistics**]: *Probabilistic Algorithms*; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Analysis of Algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: String matching, streaming algorithms, randomized algorithms

ACM Reference Format:
Dany Breslauer and Zvi Galil. 2014. Real-time streaming string-matching. ACM Trans. Algor. 10, 4, Article 22 (July 2014), 12 pages.
DOI: http://dx.doi.org/10.1145/2635814

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 [Galil 1981; Galil and Seiferas 1983; Karp and Rabin 1987; Knuth et al. 1977]. However, all these algorithms, including the online 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 in which, initially, all text positions are candidate occurrences of the pattern, and an algorithm's task is to eliminate candidates and verify which of the remaining text positions are actual occurrences. The classical Knuth-Morris-Pratt [Knuth et al. 1977] 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

This work was partially supported by the European Research Council (ERC) Project SFEROT and by the Israeli Science Foundation Grants 35/05, 686/07, 347/09 and 864/11.
Author's addresses: D. Breslauer, Caesarea Rothchild Institute for Interdisciplinary Applications of Computer Science, University of Haifa, Israel; email: dany@cs.columbia.edu; Z. Galil, College of Computing, Georgia Institute of Technology; email: galil@cc.gatech.edu.

DOI: http://dx.doi.org/10.1145/2635814

another matching pattern prefix, while skipping candidate occurrences that can be ruled out by the transitivity of the pattern's prefix self-overlap (also called its *period*). The lengths of all such shifts are precomputed in the pattern preprocessing phase and take up $O(m)$ space.

The Karp-Rabin [Karp and Rabin 1987] 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 fingerprints different from the pattern's fingerprint. Their algorithm, however, requires access to the last m text symbols to slide the fingerprint window along the text. Although the fingerprint functions always identify equal strings, different strings are usually mapped to different fingerprints but, sometimes, with small probability, they are mapped 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 [2009] 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 article, 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 [2009] 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 although 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 [Ergün et al. 2010] 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, which 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 [2009] algorithm (still two-sided errors). Moreover, Ergun, Jowhari, and Salgan [Ergün et al. 2010] prove that $\Omega(m)$ space is required by any one-pass streaming algorithm that computes the period length of *nonperiodic strings*, but that 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 article presents two streaming string-matching algorithms. The first, like Porat and Porat's [2009] 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 (the amount of available space $O(\log m)$, which is derived from the pattern length m, must be known

in advance). Our techniques can be used to speed up Ergün et al. [2010] periodicity streaming algorithm to $O(m)$ time.

The new algorithms' candidate elimination resembles Galil's [1985] parallel string-matching algorithm. However, whereas that parallel algorithm works in consecutive stages that simultaneously access the entire pattern and text, execution of the streaming algorithm proceeds in a skewed fashion, partially evaluating all the stages simultaneously when the next input symbol is presented and without accessing the whole pattern and text, which are not available in the streaming model. This has the effect that viable occurrence candidates essentially climb from one stage to the next until full occurrences of the pattern are reported.

This article starts by reviewing some of the basic properties of fingerprint arithmetic and periodic strings needed to describe the algorithms in Section 2. Sections 3–5 describe the two algorithms and their required pattern preprocessing. The article finishes with concluding remarks and open problems in Section 6.

2. FINGERPRINTS AND PERIODS

The new algorithms make use of the Karp and Rabin's [1987] fingerprint functions, which are defined as $\phi_{r,p}(s) = \sum_{i=1}^{l} s_i\, r^i \bmod p$ 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 *error probability*—the probability that two different strings share the same fingerprint—can be bounded as follows:

THEOREM 2.1. *Let u and v be two different string 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 $\frac{1}{n^{1+\alpha}}$.*

PROOF. The fingerprint equality $\sum_{i=1}^{l} u_i r^i = \sum_{i=1}^{l} v_i r^i \bmod p$ means that the polynomial $\sum_{i=1}^{l} (u_i - v_i) x^i = 0 \bmod p$ admits r as its root. Since $\mathcal{F}_p$ is a field, any degree l polynomial has at most l roots. The probability that some random $r \in \mathcal{F}_p$ is one such root is at most $\frac{l}{p} \leq \frac{1}{n^{1+\alpha}}$. □

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; they do not need to be computed at every step). The following primitive fingerprint operations were also used by Karp and Rabin [1987], Porat and Porat [2009] and Ergün et al. [2010].

LEMMA 2.2. *One can compute the fingerprint of concatenated strings u and v as follows:*

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

PROOF. Immediate from the definition of the fingerprint function. □

Lemma 2.2 can be used to remove the fingerprint of the prefix u from the fingerprint of the concatenated string uv to extract the fingerprint of $v = u^{-1}(uv)$. Similarly, one can also remove the fingerprint of the suffix v to extract the fingerprint of $u = (uv)v^{-1}$.

COROLLARY 2.3. *The fingerprint function:*

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

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

Equations (2) and (3) 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 streaming 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 2.4. *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.*

PROOF. Let u be the text prefix up to the location of interest and let v be the continuation text block up to and including the current text symbol. Then, the text prefix up to and including the current text symbol is uv, and, by Corollary 2.3, the fingerprint of v is given by canceling the fingerprint of u from the fingerprint of uv. □

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; otherwise, the period of u terminates in v. The following *Periodicity Theorem* is credited to Fine and Wilf [1965].

THEOREM 2.5. *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 $\gcd(p, q)$.*

Remark. One of the important goals in this work has been to allow only false-positive phantom occurrences and eliminate false-negative omitted occurrence errors. The combined use of fingerprints that introduce low probability errors and periods that rely on the transitivity of symbol equality can be confusing and may quickly lead to two-sided errors. We managed to avoid some of the pitfalls in Porat and Porat [2009] work:

(1) because fingerprint equality of larger strings does not imply fingerprint equality of their respective substrings, canceling out some *pattern* prefixes in the fingerprint of some *text* block may introduce very strange errors, in which the algorithm may wrongly proceed with the wrong fingerprint of some text block and even report false occurrences at text locations where the real text fingerprint does not match the pattern's fingerprint;
(2) using periodicity of pattern prefixes to skip occurrence candidates may cause the algorithm to skip past actual occurrences after a false fingerprint match;
(3) relying on pattern prefix periods that might not be correctly computed can propagate one-sided errors into two-sided errors.

3. THE $O(N \log M)$ TIME ALGORITHM

This section describes the $O(n \log m)$ streaming string-matching algorithm and introduces the basic concepts that are refined in the next section to obtain the real-time algorithm. The algorithm's pattern preprocessing is trivial. It computes the sequence $\mathcal{P}_i$ of $\lceil \log m \rceil$ 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, then for $k = \lceil \log_2 m \rceil$ (i.e., the largest k), $\mathcal{P}_k = \mathcal{P}[1..m]$. These $\lceil \log_2 m \rceil$ fingerprints are stored in $O(\log m)$ space. No period lengths of the pattern or any of its prefixes are required.

(1) If the first, longest, viable occurrence in stage number i has precisely $|\mathcal{P}_{i+1}|$ text symbols (up to and including the current text location), then remove this viable occurrence from stage number i, and compare its fingerprint to the fingerprint of the pattern prefix $\mathcal{P}_{i+1}$ as candidate for stage number $i+1$ promotion. The next viable occurrence in stage number i, if any, becomes the first viable occurrence in the stage.
   — If the fingerprints match, promote the viable occurrence to stage number $i+1$.
   — Viable occurrences that get promoted to the ultimate stage matched the fingerprint of the whole pattern. These viable occurrences are reported as occurrences of the whole pattern and do not need to be stored.
(2) To initialize, each text symbol's fingerprint that is equal to $\mathcal{P}_0$ adds to stage number 0 a new viable occurrence starting at that text position.

Fig. 1. Stages of the $O(n \log m)$ algorithm.

Whenever possible, we give an intuitive verbal description. A *viable occurrence* is a text position that has not been ruled out as a start of an occurrence. The algorithm maintains viable occurrences of the pattern prefixes through $\lceil \log_2 m \rceil$ simultaneous stages that filter the remaining viable occurrences while the text is being streamed online. 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 stages are summarized in Figure 1, where all stages are executed in increasing order for each input symbol.

The viable occurrences are grouped into stages. A viable occurrence belongs to stage number i, if the algorithm has verified earlier that it starts with a text block of length $|\mathcal{P}_i|$ whose fingerprint is equal to the fingerprint of pattern prefix $\mathcal{P}_i$ but there are insufficient text symbols yet to verify if it belongs to stage number $i+1$. As soon as there are sufficient symbols—$|\mathcal{P}_{i+1}|$ to be precise—to promote the viable occurrence to the next stage (always the first, longest viable occurrence in the stage), the appropriate text block fingerprint is compared to the precomputed pattern prefix fingerprint $\mathcal{P}_{i+1}$ and the viable occurrence either gets promoted to the next stage or is eliminated. Clearly, an occurrence of each of the pattern prefixes must start at each occurrence of the pattern, and viable occurrences eliminated in this way cannot be occurrences of the pattern.

Each text position is initially considered a viable occurrence. When 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 viable occurrence may enter stage number 0. Note that viable occurrences that start earlier in the text always correspond to longer text blocks and therefore belong to higher or equal numbered stages. One can envision the viable occurrences 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 $\mathcal{P}_k = \mathcal{P}[1..m]$. Because all text positions are considered viable occurrences and are only eliminated as a consequence of fingerprint mismatch, the algorithm commits no false-negative errors.

The algorithm maintains the fingerprint of each viable occurrence x, which is the fingerprint of the text prefix that ends at x, the running fingerprint of the text when x was reached. When x is ripe for promotion, the fingerprint of the block that starts with x and ends at the current text symbol is computed from the current running fingerprint of the text and the fingerprint of x using Lemma 2.4 and compared to the fingerprint of $\mathcal{P}_{i+1}$.

As in Galil's [1985] parallel string-matching algorithm, multiple viable occurrences that get too crowded in some stage imply that there must be a periodic pattern prefix. Specifically, if there are at least three viable occurrences at the same stage number i, then the pattern prefix $\mathcal{P}_i$ must be periodic, all the viable occurrences in this stage form an arithmetic progression whose difference is the period length of $\mathcal{P}_i$, and, therefore, these viable occurrences can be represented compactly and processed efficiently. We prove this next.

LEMMA 3.1. *Let u and v be strings such that v contains at least three occurrences of u. Let $t_1 < t_2 < \cdots < t_h$ be the locations of all occurrences u in v and assume that $t_{i+2} - t_i \leq |u|$, for $i = 1, \ldots, h-2$ and $h \geq 3$. Then, this sequence forms 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.*

PROOF. Consider any consecutive three elements $t_i, t_{i+1,}$ and t_{i+2} and let $p = t_{i+1} - t_i$ and $q = t_{i+2} - t_{i+1}$. Because of the overlap of these occurrences of u in v, u must have periods of length p and q. Since $p + q = t_{i+2} - t_i \leq |u|$, by Theorem 2.5 u must also have a period of length $\gcd(p, q)$. If r is the period length of u, we have similarly that $\gcd(r, p, q)$ is a period length of u that must be equal to r by the minimality of r. It follows that all occurrences of u between t_i and t_{i+2} are in the form of $t_i + kr$. Also, by aligning occurrences, there must be occurrences at all these locations $t_i + kr$. Therefore $p = q = r$. □

LEMMA 3.2. *Suppose that there are at least three viable occurrences of $\mathcal{P}_i$ in stage number i. If these are actual occurrences of the pattern prefix $\mathcal{P}_i$, then these viable occurrences form an arithmetic progression with difference equal to the period length of $\mathcal{P}_i$.*

PROOF. Since $|\mathcal{P}_{i+1}| \leq 2|\mathcal{P}_i|$ and since viable occurrences that have been compared to $\mathcal{P}_{i+1}$ were either promoted to the next stage or eliminated, the difference between the locations of any viable occurrences of $\mathcal{P}_i$ in stage number i is less than $|\mathcal{P}_i|$. By Lemma 3.1, actual occurrences form an arithmetic progression with difference that is equal to the period length of $\mathcal{P}_i$. The viable occurrences must include all actual occurrences, and, if all viable occurrences are actual occurrences, then the viable occurrences also form the same arithmetic progression. □

Unfortunately, there is one important caveat here. The streaming algorithm compares fingerprints and not actual strings, and therefore different strings may be identified by the same fingerprint, thus conflicting with the periodicity *implied* by string equality, called hereafter *fingerprint-periodicity conflict*. The algorithm may conclude that some fingerprint false-match error must have occurred because the periodicity properties have been violated but without precisely identifying the culprit fingerprint error. Note that the algorithm only uses the periodicity properties to facilitate the space-efficient representation and never to eliminate viable occurrence candidates.

LEMMA 3.3. *The viable occurrences in each stage can be compactly represented in constant space, allowing insertion of the last viable occurrence in the stage and the removal of the first viable occurrence in constant time. The representation faithfully reconstructs the viable occurrences if no fingerprint-periodicity conflict is detected while inserting viable occurrences.*

PROOF. If there are only one or two viable occurrences in stage number i, these viable occurrences and their fingerprints are stored directly. If there are three or more, they should form an arithmetic progression by Lemma 3.2. To compactly and faithfully represent the arithmetic progression, as soon as there are two viable occurrences in the representation, the algorithm computes the *implied period length* of $\mathcal{P}_i$, which is set to the difference between the two viable occurrences, and the *implied period fingerprint*, which is the fingerprint of the text block between the two viable occurrences. This information takes constant space and is maintained together with the text positions and fingerprints of the first and last viable occurrences in the progression.

To insert the last viable occurrence into the compact representation, the algorithm verifies that this new viable occurrence continues the arithmetic progression by

checking that the difference between the last viable occurrence and the new viable occurrence is equal to the implied period length and that the text block fingerprint between the last viable occurrence and the new viable occurrence is equal to the implied period fingerprint. If these two tests succeed, the algorithm concludes that there is no fingerprint-periodicity conflict and updates the last viable occurrence it stores to the new viable occurrence.

To remove the first viable occurrence from the compact representation, it is necessary to update the next first viable occurrence. The position of the first viable occurrence is advanced by the implied period length, and the fingerprint is adjusted using the implied period fingerprint.

The representation is clearly faithful if the insertion tests detected no fingerprint-periodicity conflicts. All fingerprint manipulation is done using Corollary 2.3. □

In Lemma 3.3, the algorithm does not know the real period length of $\mathcal{P}_i$ nor the real period of $\mathcal{P}_i$. The algorithm only verifies that whatever goes into the compact representation will eventually faithfully come out, conforming to some arithmetic progression and some repeated text blocks that share the same fingerprints but might even be different strings.

If the algorithm encounters any fingerprint-periodicity conflicts while inserting a viable occurrence into the compact representation, the algorithm concludes that some of the viable occurrences are not actual occurrences due to a small probability of false-positive fingerprint errors. The algorithm must make hard choices to remain within its strict space bounds while ensuring that only false-positive errors are reported and no actual occurrences are omitted. The algorithm will discard some valid viable occurrences that cannot be compactly represented via the arithmetic progression. However, the algorithm will report all these discarded viable occurrences as occurrences of the pattern so that *no occurrences are missed*.

Recall that the offending viable occurrence that revealed the fingerprint-periodicity conflict is in the process of being promoted from some stage to the next. To simplify the presentation and avoid cascading the effects of discarded viable occurrences on higher numbered stages, the algorithm will discard and report all earlier viable occurrences (in equal or higher numbered stages), excluding the offending viable occurrence and the last viable occurrence in the arithmetic progression, and it will keep all other viable occurrences. The up to $O(\log m)$ discarded arithmetic progressions can be compactly written to the output as arithmetic progressions rather than spelled out individually, to remain within the time bounds (because this is such a low-probability event, the algorithm may even report as occurrences all text locations between the first and last discarded viable occurrences). More details on discarding viable occurrences are given in Section 5.

Thus, the algorithm might report two classes of erroneous pattern occurrences: those *phantom* occurrences that passed through the entire stage ladder and eventually had their fingerprint verified against the fingerprint of the whole pattern and those viable occurrences that were thrown off the stage ladder due to some nonspecific fingerprint false-match errors conflicting with the implied periodicity and keeping the algorithm from compactly representing crowded viable occurrences. The error probability in both cases is bounded, since it is either due to a fingerprint false-match of the whole pattern (and its stage prefixes) or to a detected fingerprint-periodicity conflict that must be due to a fingerprint false-match of some pattern prefix.

THEOREM 3.4. *The algorithm just described 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 occasion, it even detects that it had fingerprint errors, with probability at most* $1/n^{\alpha}$.

PROOF. Each stage number i 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 viable occurrence. 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. □

By allowing extra space to store more individual viable occurrences or arithmetic progressions, the algorithm could continue to examine the viable occurrences. In such a case, the worst-case space will not be $O(\log m)$, but by making the error probability sufficiently small, the expected space would still be $O(\log m)$. In Section 5, we show a way to avoid this undesirable property.

4. THE REAL-TIME ALGORITHM

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

To overcome the first challenge, we find an appropriate prefix of the pattern between $\mathcal{P}_g$ and $\mathcal{P}_{g+1}$, for $g > \log_2 \log_2 m$, and use Galil's [1981] deterministic real-time implementation of the Knuth-Morris-Pratt [Knuth et al. 1977] algorithm, essentially performing all stages $i = 0, \ldots, g-1$ together. 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 will start with Galil's real-time algorithm 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 to store the pattern prefix and its failure function. If $\mathcal{P}_f$ is periodic, then the pattern preprocessing will continue to examine further symbols of $\mathcal{P}$ until either the periodicity ends or the pattern ends. The periodic pattern prefix and its failure function can still be stored using the same $O(\log m)$ space by keeping the basic period and its length and recalling that the Knuth-Morris-Pratt failure function essentially consists of the period length of each pattern prefix, which remains the same after $\mathcal{P}_f$ for as long as the periodicity did not end. There are three cases:

(1) If the pattern prefix $\mathcal{P}_f$ is nonperiodic, then we choose $g = f$. We use Galil's real-time algorithm to find all occurrences of $\mathcal{P}_g$ in the text using $O(\log m)$ space. Since $|\mathcal{P}_g| \geq 2 \log_2 m$, the occurrences of $\mathcal{P}_g$ are at least $\log_2 m$ positions apart.
(2) If the pattern prefix $\mathcal{P}_f$ is periodic, and the periodicity does not end in the pattern $\mathcal{P}$, we choose $g = \lceil \log_2 m \rceil$, and $\mathcal{P}_g$ is the whole pattern. We use Galil's real-time algorithm to find the occurrences of the whole pattern $\mathcal{P}_g$.
(3) If the pattern prefix $\mathcal{P}_f$ is periodic and this periodicity ends in $\mathcal{P}$, let π be the longest pattern prefix with the same period, such that the periodicity terminates at πa. We choose g to be such that $|\mathcal{P}_g| \leq |\pi a| < |\mathcal{P}_{g+1}|$ and use Galil's real-time algorithm to find the occurrences of the pattern prefix πa. The reason we can still use Galil's algorithm is that, to search for the pattern prefix πa, only $O(1)$ additional space is required to keep πa and its failure function over the space that was used for the periodic π, and this is the only information that the algorithm needs.
The following lemma shows that the pattern prefix πa is nonperiodic and therefore its occurrences must be spaced by more than $|\pi a| \geq |\mathcal{P}_g|/2 \geq \log_2 m$ text positions

apart. Moreover, such occurrences must start with $\mathcal{P}_g$ and will be reported by the real-time string-matching algorithm before sufficient symbols are available to verify $\mathcal{P}_{g+1}$. Observe that no arithmetic progressions are forming at stage number g because the spacing between the viable occurrences is too large.

LEMMA 4.1. *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.*

PROOF. Since $v = \hat{u}a$ is the shortest prefix of w where the period of u terminates, the prefix $\hat{u}$ has the same period as u that terminates at the letter a. Let $p \leq |u|/2$ be the period length of $\hat{u}$, and let $q \leq |v|$ be the period length of v and therefore also the period length of $\hat{u}$. All periods of $\hat{u}$ that are multiples of the shortest period length p must also terminate in v. Hence, by Theorem 2.5, q must not be a multiple of p and $q > |\hat{u}| - p + \gcd(p, q) \geq |\hat{u}| - p + 1 = |v| - p > |v|/2$. □

Both cases (1) and (3) above essentially compute all stages numbered $0, \ldots, g-1$ together. In case (1) all occurrences of $\mathcal{P}_g$ become viable occurrences in stage g as in the $O(n \log m)$ algorithm, whereas in case (3) all occurrences of πa become viable occurrences in stage g (these are not all occurrences of $\mathcal{P}_g$, but only those occurrences that can be extended to πa and later to complete occurrences of the whole pattern $\mathcal{P}$).

To overcome the second challenge and avoid too many promotions causing congestion at certain text positions, we adapt the $O(n \log m)$ algorithm that handles the higher numbered stages. Thus, the real-time algorithm has two parts that are run alongside each other. Galil's [1981] real-time string-matching algorithm that feeds viable occurrences to stage number g of the real-time adaptation of the $O(n \log m)$ time algorithm is described next.

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

(1) The simulated action may happen out of order with respect to the $O(n \log m)$ time algorithm and even in different order depending on the text location. Note that because viable occurrences 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 in which the different order will lead to a different computation is the following: Assume that in the real-time algorithm x and y are the first and second viable occurrences in stage i, and z was just promoted from stage $i-1$ to stage i as the third viable occurrence in the stage and it reveals an inconsistency with the periodicity. It is possible that in the $O(n \log m)$ time algorithm x is promoted from stage i to stage $i+1$ before z is promoted to stage i; therefore, in that algorithm, y and z are the only viable occurrences in stage i, and there is no inconsistency. We can easily fix the order in such a case to be the same by deleting x from stage i first, because 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 because the algorithm is still correct with the different order.

(2) The real-time online algorithm has to output the pattern occurrences immediately at their end; thus, 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 $\mathcal{P}$ 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 viable occurrences 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 viable occurrences 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 4.2. *The algorithm just described 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 at most $1/n^{\alpha}$.*

PROOF. The algorithm updates the running fingerprint buffer with the current text symbol in constant time. Each delayed stage action can be properly done because 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 Galil [1981] 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 of storing the fingerprints of the pattern prefixes $\mathcal{P}_i$. The real-time algorithm needs to additionally 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 therefore needs to know the length of the pattern m (or an approximation m' of m such that $\log m' = \Theta(\log m)$) in advance while preprocessing the pattern.

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. Conflict-free fingerprint functions can be obtained by trying out several random seeds in multiple passes over the pattern if the pattern is available for additional re-processing (i.e., in a k-pass streaming pattern preprocessing algorithm) or by trying out several random seeds in parallel. Observe that such low-probability fingerprint-periodicity conflicts would repeat in every occurrence of the pattern in the text because the viable occurrences within an actual occurrence of the patten in the text includes all viable occurrences processed while matching the pattern against itself.

THEOREM 5.1. *Given the fingerprint function and the pattern, if the pattern is fingerprint-periodicity conflict free, then when the streaming algorithm discards viable occurrences in the text due to fingerprint-periodicity conflict, the discarded viable occurrences do not need to be reported as potential occurrences.*

PROOF. Assume that the algorithm encounters a fingerprint-periodicity conflict while processing the text. Such conflicts arise when some viable occurrence is promoted into

the next stage but it does not fit the arithmetic progression and periodicy requirements. Let x and y be the last two viable occurrences in this stage and let z be the newly promoted viable occurrences. Note that there might be other viable occurrence starting at positions before x in the text at the same or higher number stages.

Suppose that the viable occurrence starting at $x' \leq x$ is an actual pattern occurrence. Because the surviving viable occurrences only depend on fingerprints, all viable occurrences that remained within an actual pattern occurrence must be the same as the viable occurrences maintained at the same relative phase while matching the pattern against itself (as a text string) in the pattern preprocessing.

Therefore, if there were no such fingerprint-periodicity conflicts while preprocessing the pattern, there can be no actual pattern occurrence starting at x'. Therefore, if the algorithm encounters such a fingerprint-periodicity conflict (occurrences x, y, and z), it is permissible to discard all the viable occurrences $x' \leq x$ while keeping y and z and all viable occurrences $z' > z$. □

6. CONCLUSION

In addition to their string-matching algorithm, Porat and Porat [2009] also presented approximate string-matching streaming algorithms that rely on their exact streaming string-matching algorithm. In private communications, Porat [2010] mentioned to us that those bounds have since been further improved. By swapping Porat and Porat's exact string-matching black-box component with our new real-time string-matching algorithm, one can improve their approximate string-matching time bounds by an additional $\log m$ factor.

Porat and Porat [2009] also mention numerous applications for streaming string matching. In many cases, one would be interested in searching simultaneously for multiple patterns, also called the *dictionary matching problem*. By running d instances of our new streaming algorithm, the bounds trivially add up to $O(dn)$ time and $O(\sum_{j=1}^{d} \log m_j)$ space following an $O(\sum_{j=1}^{d} m_j)$ time pattern preprocessing. However, if the dictionary patterns have uniform length m, then our $O(n \log m)$ time streaming algorithm can be extended to d patterns within the same $O(n \log m)$ time bounds. In private communications, Porat [2010] mentioned to us that he also had reached somewhat similar conclusions about his work.

There still remains a gap between the lower and upper bounds:

—Ergün et al. [2010] proved that $\Omega(\log m)$ space is required for streaming string matching when $n \geq m^{1+\epsilon}$. We suspect that the same lower bound should hold even when $n = cm$, for a constant c.
—In private communications [Alon 2010] pointed out an $\Omega(d)$ space streaming lower bound argument for dictionary matching with d patterns. It is an interesting question to close the gap between the lower bounds and the $O(d \log m)$ time upper bound as the pattern length m grows.

ACKNOWLEDGMENTS

The authors wish to thank Noga Alon, Roberto Grossi, Gadi Landau, Benny Pinkas, and Ely Porat for useful discussions and comments.

REFERENCES

N. Alon. 2010. Private communications.

F. Ergün, H. Jowhari, and M. Saglam. 2010. Periodicity in streams. In *APPROX-RANDOM*, M. J. Serna, R. Shaltiel, K. Jansen, and J. D. P. Rolim, Eds. Lecture Notes in Computer Science Series, vol. 6302. Springer, 545–559.

N. Fine and H. Wilf. 1965. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society* 16, 109–114.

Z. Galil. 1981. String matching in real time. *Journal of the ACM* 28, 1, 134–149.

Z. Galil. 1985. Optimal parallel algorithms for string matching. *Information and Control* 67, 144–157.

Z. Galil and J. Seiferas. 1983. Time-space-optimal string matching. *Journal of Computer and System Scences* 26, 280–294.

R. Karp and M. Rabin. 1987. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development* 31, 2, 249–260.

D. Knuth, J. Morris, and V. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing* 6, 322–350.

B. Porat and E. Porat. 2009. Exact and approximate pattern matching in the streaming model. In *FOCS*. IEEE Computer Society, 315–323.

E. Porat. 2010. Private communications.

Received November 2011; revised March 2012; accepted March 2012