Real-Time Streaming Multi-Pattern Search for Constant Alphabet*

Shay Golan[1] and Ely Porat[2]

1 **Bar Ilan University, Ramat Gan, Israel**
golansh1@cs.biu.ac.il
2 **Bar Ilan University, Ramat Gan, Israel**
porately@cs.biu.ac.il

Abstract

In the streaming multi-pattern search problem, which is also known as the streaming dictionary matching problem, a set $D = \{P_1, P_2, \ldots, P_d\}$ of d patterns (strings over an alphabet Σ), called the *dictionary*, is given to be preprocessed. Then, a text T arrives one character at a time and the goal is to report, before the next character arrives, the longest pattern in the dictionary that is a current suffix of T. We prove that for a constant size alphabet, there exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that takes constant time per character and uses $\mathcal{O}(d \log m)$ words of space, where m is the length of the longest pattern in the dictionary. In the case where the alphabet size is not constant, we introduce two new randomized Monte-Carlo algorithms with the following complexities:

- $\mathcal{O}(\log\log|\Sigma|)$ time per character in the worst case and $\mathcal{O}(d \log m)$ words of space.
- $\mathcal{O}(\frac{1}{\varepsilon})$ time per character in the worst case and $\mathcal{O}(d|\Sigma|^{\varepsilon} \log \frac{m}{\varepsilon})$ words of space for any $0 < \varepsilon \leq 1$.

These results improve upon the algorithm of Clifford et al. [12] which uses $\mathcal{O}(d \log m)$ words of space and takes $\mathcal{O}(\log\log(m + d))$ time per character.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases multi-pattern, dictionary, streaming pattern matching, fingerprints

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.41

1 Introduction

We consider one of the most fundamental pattern matching problems, the *dictionary matching problem* [12, 16, 5, 6, 30, 20, 7, 21, 17, 18, 4], where a set of patterns $D = \{P_1, P_2, \ldots, P_d\}$, called the *dictionary*, is given along with a string T, called the *text*, such that each pattern P_i is a string of length m_i, and all the strings are over an alphabet Σ. The goal is to find all the occurrences of patterns from D in T. The dictionary matching problem is a natural generalization of the simple pattern matching problem of one pattern, and it has many applications in different areas. For example, in the area of Intrusion Detection and Anti-Viruses systems [36], the goal is to detect viruses in a stream of data by looking for known digital signatures of these viruses. Due to the importance of the problem, significant efforts have been made to speed up algorithms for this problem, for example, by using GPUs [38, 39, 37, 40, 43, 26] or even using a designated hardware [15, 2, 41, 29, 42, 9].

* This work is supported in part by ISF grant 1278/16, and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 683064).

The streaming model. In the *streaming* model [3, 25, 32, 28], we have a stream of data to process in near real-time while using only sublinear space. For pattern matching problems, the pattern is given in advance and the text arrives one character at a time, and the goal is to decide after the arrival of each character, whether the current suffix of the text matches the preprocessed pattern. Since the seminal paper of Porat and Porat [33] which introduced the first algorithm for this problem in the streaming model, there has been a rising interest in solving pattern matching problems in the streaming model [10, 14, 31, 11, 27, 12, 13, 22].

For the dictionary matching problem in the streaming model, D is given in advance, and the text T arrives one character at a time. After the arrival of $T[q]$ the algorithm must report the *longest*[1] suffix of $T[1..q]$, which is a pattern in D. The space usage of the algorithm is limited to sublinear space, hence, one cannot even store the dictionary D explicitly. The efficiency of algorithms in this model is measured by the amount of time required to process a text character and the total space usage of the algorithm. Another closely related model is the *online* model, which is the same as the streaming model without the constraint of using sublinear space.

Previous results and related work. The current most efficient algorithm for the streaming dictionary matching is due to Clifford et al. [12] which uses $\mathcal{O}(d \log m)$ words[2] of space and takes $\mathcal{O}(\log\log(d+m))$ time per character, where $m = \max\{m_i\}$ is the length of the longest pattern. This algorithm assumes that there are no two patterns $P_i, P_j \in D$ such that P_i is a suffix of P_j. Otherwise, the algorithm reports any time *some* pattern that is a current suffix of the text, but not necessarily the longest one. The algorithm is a randomized Monte-Carlo algorithm and is correct with high probability.

In the online model, most of the algorithms are variations of the Aho and Corasick [1] algorithm. This algorithm has two versions, the DFA (deterministic finite automaton) and the state-machine. The DFA version takes $\mathcal{O}(1)$ time per character and uses $\mathcal{O}(M|\Sigma|)$ words of space, where $M = \sum_{i=1}^{d} m_i$ is the sum of the patterns' length. The state-machine version uses only $\mathcal{O}(M)$ words of space, and its amortized running time is also constant. However, for the online model, which measures the running time per character, in the worst case the state-machine version takes $\Omega(m)$ time per character, which is unreasonable. Hence, the algorithm of Kopelowitz et al. [30], improves the state-machine algorithm, to $\mathcal{O}(\log\log|\Sigma|)$ time per character, and it still uses $\mathcal{O}(M)$ words of space and $\mathcal{O}(1)$ amortized time per character. Both Aho and Corasick [1] and Kopelowitz et al. [30] algorithms are deterministic, and we use some of their concepts in our results.

Our results. Our first result is for the case of a constant alphabet and is stated in the following theorem:

▶ **Theorem 1.** *For a constant size alphabet, there exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends constant time per arriving text character and uses* $\mathcal{O}(d \log m)$ *words of space.*

---

[1] This is a common simplification in which one must only report the longest pattern that has arrived (if several patterns end at the same text location), since converting such a solution to one that reports all the patterns is straightforward with additional time which is linear in the number of reported patterns, and this way the focus is on the time cost that is independent from the output size.

[2] We assume the RAM model where each word has size of $\Omega(\log n)$ bits

We mention the open problem of Breslauer and Galil [10] who solve the problem for the case of one pattern. They ask whether $\Omega(d \log m)$ words of space is required for any streaming dictionary matching algorithm. If the answer to this problem is positive, then the algorithm of Theorem 1 has optimal time and space.

For the general case where the alphabet size is arbitrary, we introduce two new algorithms, and each of them suffices as a proof for Theorem 1. The first algorithm is an improvement over the algorithm of Clifford et al. [12] with the same space usage, but with running time of $\mathcal{O}(\log\log|\Sigma|)$ time per character, compared to $\mathcal{O}(\log\log(m+d))$ time per character of [12]. Moreover, our algorithm solves the stronger version of the problem where the algorithm has to report the *longest*[1] pattern in D which is a current suffix of the text.

▶ **Theorem 2.** *There exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends* $\mathcal{O}(\log\log|\Sigma|)$ *time per arriving text character and uses* $\mathcal{O}(d \log m)$ *words of space.*

We point out that even though someone who is familiar with the area would expect an *amortized* $\mathcal{O}(1)$ time per character for the algorithm of Theorem 2, unfortunately this is not the case. In the algorithm for the online model by Kopelowitz et al. [30], the algorithm has a tree of states, and after the arrival of a character the algorithm moves to a state which its depth is larger than the former state's depth by at most one. Thus, the amortized analysis of this algorithm was based on the depth of algorithm's state. In our algorithm, we also have states with depths but we could sometimes jump into a state that is much deeper than the former state, therefore, such an amortized analysis will not hold. An interesting question is whether one can design an algorithm with the same space usage and worst case time per character, but with amortized $\mathcal{O}(1)$ time per character.

Our second algorithm is a real-time algorithm, with a small amount of extra space.

▶ **Theorem 3.** *For any constant* $0 < \varepsilon \leq 1$ *there exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends* $\mathcal{O}(\frac{1}{\varepsilon})$ *time per arriving text character and uses* $\mathcal{O}(d|\Sigma|^{\varepsilon} \log \frac{m}{\varepsilon})$ *words of space.*

1.1 Algorithmic Overview

We prove simultaneously Theorem 2 and a degenerate version of Theorem 3, where $\varepsilon = 1$, as stated in the following lemma:

▶ **Lemma 4.** *There exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends* $\mathcal{O}(1)$ *time per arriving text character and uses* $\mathcal{O}(d|\Sigma| \log m)$ *words of space.*

Then, Theorem 3 is deduced from Lemma 4 by implying the following theorem of Rozen [34].

▶ **Theorem 5.** *Let* $\mathcal{A}$ *be an algorithm for the online dictionary pattern matching problem which uses* $\mathcal{O}(s_{\mathcal{A}}(d, m, |\Sigma|))$ *words of space and takes* $\mathcal{O}(t_{\mathcal{A}}(d, m, |\Sigma|))$ *time per character. Then, for any* $0 < \varepsilon \leq 1$*, there exists an algorithm* $\mathcal{A}_{\varepsilon}$ *for this problem which uses* $\mathcal{O}(s_{\mathcal{A}}(d, \frac{m}{\varepsilon}, |\Sigma|^{\varepsilon}))$ *words of space and takes* $\mathcal{O}(\frac{1}{\varepsilon} t_{\mathcal{A}}(d, \frac{m}{\varepsilon}, |\Sigma|^{\varepsilon}))$ *time per character.*

The algorithms for Theorem 2 and Lemma 4 are very similar and have only small number of differences, therefore we describe them as one algorithm, and demonstrate only the differences. We follow the basic partition of D, presented by Clifford et al. [12], into three types of patterns. The types are short patterns, long patterns with a small period length, and long patterns with a large period length. We introduce an algorithm for each type, $\mathcal{A}_1$, $\mathcal{A}_{2a}$, and $\mathcal{A}_{2b}$, respectively. Theorem 2 and Lemma 4 are obtained by running all three algorithms in parallel.

Each one of the algorithms $\mathcal{A}_1$, $\mathcal{A}_{2a}$, and $\mathcal{A}_{2b}$ is composed of two phases. At the high level, the algorithm considers for each pattern $\log m$ prefixes, called *heads* of the pattern. For a pattern P_i of length m_i, the algorithm considers all the prefixes of length $\ell \in (m_i - 2\log m, m_i - \log m]$. Thus, the total number of heads is at most $d \log m$. The algorithm utilizes the fact that each occurrence of P_i in the text must begin with an occurrence of some head, such that this occurrence ends at a position which is a multiple of $\log m$.[3]

In the first phase, at each text position that is a multiple of $\log m$, the algorithm finds the current longest suffix that is a head of some pattern. The running time of the first phase is as stated in Theorem 2 or Lemma 4, with additional $\mathcal{O}(\log m)$ running time for each text position that is a multiple of $\log m$. This runtime is de-amortized during the arrival of $\log m$ characters between each two such positions. We introduce the first phase of $\mathcal{A}_1$ in Section 4 and the first phase of $\mathcal{A}_{2a}$ and $\mathcal{A}_{2b}$ in Section 5.

In the second phase, after finding the longest suffix that is a head of some pattern, the algorithm reads the text one character at a time using a state machine, which is inspired by the Aho and Corasick [1] algorithm. The initial state is obtained by the longest head that is found in the first phase, and each state transition is done according to the character that arrived. Whenever a pattern in the dictionary is a current suffix of the text, the state of the machine represents this pattern or a longer string which this pattern is its suffix. Hence, the algorithm has the correct pattern to report at any time. The details of the second phase appear in Section 3.

2 Preliminaries

A string S of length $|S| = \ell$ is a sequence of characters $S[1]S[2]\dots S[\ell]$ over an alphabet Σ. A *substring* of S is denoted by $S[x..y] = S[x]S[x+1]\dots S[y]$ for $1 \le x \le y \le \ell$. If $x = 1$, the substring is called a *prefix* of S, and if $y = \ell$, the substring is called a *suffix* of S.

A prefix of S of length $y \ge 1$ is called a *period* of S if and only if $S[i] = S[i+y]$ for all $1 \le i \le \ell - y$. The shortest period of S is called *the principal period* of S, and its length is denoted by ρ_S. If $\rho_S \le \frac{|S|}{2}$ we say that S is *periodic*.

The proof of this lemma and other lemmas will appear in the final version of this paper.

▶ **Lemma 6.** *Let u be a periodic string with principal period length ρ_u. If v is a substring of u of length at least $2\rho_u$ then $\rho_u = \rho_v$.*

The *cyclic shift of* S is $\sigma(S) = S[2..\ell]S[1]$. For any $0 \le i < \ell$ the i^{th} cyclic shift of S is $\sigma^i(S) = S[i+1..\ell]S[1..i]$.

Fingerprints. For a natural number n we denote $[n] = \{1, 2, \dots, n\}$. For the following let $u, v \in \bigcup_{i=0}^{n} \Sigma^i$ be two strings of length at most n. Porat and Porat [33] and Breslauer and Galil [10] proved that for every constant $c > 1$ there exists a *fingerprint function* $\phi : \bigcup_{i=0}^{n} \Sigma^i \to [n^c]$, such that:

1. If $|u| = |v|$ and $u \ne v$ then $\phi(u) \ne \phi(v)$ with high probability (at least $1 - \frac{1}{n^{c-1}}$).
2. *The sliding property:* Let w=uv be the concatenation of u and v. If $|w| \le n$, then given the length and the fingerprints of any two strings from u,v and w, one can compute the fingerprint of the third string in constant time.

[3] For the sake of simplicity we assume that $\log m$ is an integer, if this is not the case we use $\lceil \log m \rceil$ instead.

Our algorithms often uss fingerprints in order to quickly validate if two strings are equal or not. To ease presentation, in the rest of the paper we assume that fingerprints never give false positives. This assumption is covered by the algorithms only failing with small probability.

2.1 Multi-labeled Trees with Lowest Labeled Ancestor queries

Let $\mathcal{L}$ be a set of labels of size $|\mathcal{L}| = \lambda$. A multi-labeled tree T is a rooted tree such that each node $v \in T$ is associated with some $\mathcal{L}_v \subseteq \mathcal{L}$. For each $v \in T$ and $\ell \in \mathcal{L}$, the lowest label ancestor $\mathsf{LLA}(v, \ell)$ is the lowest node u on the path from the root of T to v such that $\ell \in \mathcal{L}_u$, or $\perp$ if such u does not exist. We denote the total size of all the labels among the entire tree by $M = \sum_{v \in T} |\mathcal{L}_v|$. The following theorem is due to Kopelowitz et al. [30].

▶ **Theorem 7** (Deduced from [30, Theorem 3]). *For any multi-labeled tree T with a label set $\mathcal{L} = \{1, 2, \ldots, \lambda\}$, there exists a data structure that supports* LLA *queries in $\mathcal{O}(\log\log\lambda)$ time and uses $\mathcal{O}(n + M)$ words of space where n is the size of the tree and $M = \sum_{v \in T} |\mathcal{L}_v|$.*

3 The Second Phase Algorithm

In this section, we introduce the second phase of algorithms $\mathcal{A}_1$, $\mathcal{A}_{2a}$, and $\mathcal{A}_{2b}$. For each $P_i \in D$ we define the j^{th} head of P_i to be $P_i[1..m_i - j]$. The j^{th} heads set is $\mathsf{Heads}_j(D) = \{P_i[1..m_i - j] \,|\, P_i \in D\,,\, |P_i| \geq j\}$ and for a set of lengths L we define:

$$\mathsf{Heads}_L(D) = \bigcup_{j \in L} \mathsf{Heads}_j(D) = \{P_i[1..m_i - j] \,|\, P_i \in D\,,\, j \in L\,,\, |P_i| \geq j\}$$

We assume that an algorithm $\mathcal{A}$ for the first phase is given such that at each text position q that is a multiple of $\log m$, $\mathcal{A}$ finds the longest string in $\mathsf{Heads}_{[\log m, 2\log m)}(D)$ that is a suffix of $T[1..q]$. The time per character and space usage of $\mathcal{A}$ matches Theorem 2 or Lemma 4, with additional $\mathcal{O}(\log m)$ time per text position that is a multiple of $\log m$.

Our algorithm uses concepts from the Aho and Corasick algorithm [1] and especially from its online version of Kopelowitz et al. [30]. The main idea in our implementation is that instead of creating the complete Aho and Corasick state machine, we create only states that correspond to strings in $\mathsf{Heads}_{[0,2\log m)}(D)$, which are all the $2\log m$ longest prefixes of each pattern in the dictionary. Thus, the number of states is $\mathcal{O}(d \log m)$. To overcome the missing states, the algorithm uses the pattern prefixes reported by $\mathcal{A}$ to jump into the correct state soon enough, before it has to report on a pattern occurrence.

The algorithm creates a state v_S for each $S \in \mathsf{Heads}_{[0,2\log m)}(D)$ and one additional state v_ε for the empty string. In addition, the algorithm creates a perfect hash table [19, 23, 24, 35] that stores a pointer from the fingerprint of any $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D)$ to the state v_S. Another perfect hash table stores for the fingerprint of each $S \in \mathsf{Heads}_{[0,2\log m)}(D)$ the index of the longest pattern in D which is a suffix of S, if such a pattern exists.

Intuitively, the goal is that whenever the machine's state is v_S and the character that arrived is ω, the machine transits into the state $v_{S'}$ where S' is the longest suffix of $S\omega$ among all the states' strings. Since v_ε exists, such a transition is always well defined. For the algorithm of Lemma 4 each state stores explicitly all the $|\Sigma|$ transitions that correspond to any possible character, as in DFA. However, for Theorem 2 this goal is apparently impossible without a factor of $|\Sigma|$ for the space usage. Therefore, we are satisfied with a slightly weaker property, which is sufficient for the goal of reporting all patterns' occurrences. In the following paragraphs we introduce the details of the algorithm for Theorem 2.

State machine for Theorem 2. For each state v_S that represents the string S and for each $\omega \in \Sigma$, if there exists a state corresponding to the string $S' = S\omega$, the algorithm has a *goto link* from S to S' with the label ω. All the goto links of v_S are maintained in a perfect hash table due to their labels. In addition, v_S has a *failure link* to v_{S^*} where S^* is the longest proper suffix of S that has a state in the machine. Since the empty string has a state, v_ε, the failure link is defined for every state, except for v_ε itself.

The failure tree. We define the *failure tree* of the state machine, T_{fail}, as the tree induced by the states of the machine and the failure links. Since each state has exactly one failure link, except for v_ε, T_{fail} is well defined. We consider T_{fail} as a multi-labeled tree, with $\mathcal{L} = \Sigma$ as the set of labels, and for each state $v_S \in T_{\mathsf{fail}}$ and $\omega \in \Sigma$ we have that $\omega \in \mathcal{L}_{v_S}$ if and only if v_S has a goto link with the character ω. The algorithm creates the data structure of Theorem 7, which supports LLA queries in $\mathcal{O}(\log\log|\Sigma|)$ time on T_{fail}.

Performing a transition. When the machine's state is v_S and the character ω arrives, the algorithm performs the following transition. Firstly, if v_S has a goto link with label ω, the algorithm uses this link and moves into $v_{S\omega}$. If such a link does not exist, the algorithm performs the $\mathsf{LLA}(v_S, \omega)$ query. Let $v_{S'} \in T$ be the result of the query, then $v_{S'}$ has a goto link with the label ω, and the algorithm uses this link to move into $v_{S'\omega}$. If $\mathsf{LLA}(v_S, \omega) = \perp$, the algorithm moves to state v_ε. This ends the special part of the algorithm for Theorem 2.

Text processing. The second phase algorithm runs $\mathcal{A}$ to process each text character that arrives. On each position q which is a multiple of $\log m$ the algorithm creates a new process, which is alive until the time $T[q + 2\log m]$ arrives, and then the process is terminated by the algorithm. Hence, at any time the algorithm runs two processes.

Focus on the process that starts when $T[q]$ arrives for q which is a multiple of $\log m$. While the first $\frac{\log m}{2}$ characters $(T[q], \ldots, T[q + \frac{\log m}{2} - 1])$ arrive, the algorithm executes $\mathcal{A}$ for $\mathcal{O}(\log m)$ time to retrieve $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D)$ which is the longest suffix of $T[1..q]$, and keeps a buffer of the arriving characters. This execution takes $\mathcal{O}(1)$ time per character by standard de-amortization. Then, when the subsequent $\frac{\log m}{2}$ characters arrive the algorithm uses the buffer, and performs all the $\log m$ transitions beginning at v_S. Thus, by performing two transitions per arriving character, using the buffer, when $T[q+\log m]$ arrives, the machine already performed the transitions corresponding to the first $\log m$ characters.

At the following $\log m$ characters, the algorithm continues to perform transitions according to the text characters. Whenever the machine is in a state $v_{S'}$, the algorithm reports the longest suffix of S' that is a pattern in D, as the current longest suffix of the text that is a pattern in D, using the preprocessed hash table.

Due to the following lemma, the machine's state corresponds to a sufficiently long suffix of the text at any time. In particular, while processing the last $\log m$ characters of each process, if some P_i is a suffix of the text, then P_i is also a suffix of the machine's state string.

▶ **Lemma 8.** *Consider the process that starts when $T[k\log m]$ arrives. Let v_S be the state of the machine after processing $T[k\log m + i]$ for $0 \le i < 2\log m$. Then, the longest suffix of $T[1..k\log m + i]$ in $\mathsf{Heads}_{[\max\{0,\log m - i\}, 2\log m - i)}(D)$ is a suffix of S.*

Hence, since at any time there is a process which reads the arriving character as part of its last $\log m$ characters and reports matches, we deduce the following corollary.

▶ **Corollary 9.** *When $T[q]$ arrives, the algorithm reports the longest pattern in D that is a suffix of $T[1..q]$.*

Complexities. The number of states in the machine is $\mathcal{O}(|\mathsf{Heads}_{[0,2\log m)}(D)|) = \mathcal{O}(d\log m)$. For the algorithm of Lemma 4, each state has $|\Sigma|$ links, one for each $\omega \in \Sigma$. Therefore, the space usage of the second phase is $\mathcal{O}(d|\Sigma|\log m)$. The second phase for Lemma 4 takes $\mathcal{O}(1)$ time per character, since each transition is performed in constant time.

For the algorithm of Theorem 2, each state v_S has at most one goto link into v_S and one failure link from v_S. Therefore, there are $\mathcal{O}(d\log m)$ links. Since T_{fail} has $\mathcal{O}(d\log m)$ nodes, and the total size of all the nodes' labels sets is exactly the number of goto links, then the LLA data structure uses $\mathcal{O}(d\log m)$ words of space. Therefore, the total space usage of the algorithm is $\mathcal{O}(d\log m)$ words. The processing of each character requires a LLA query, thus, the second phase of Theorem 2 takes $\mathcal{O}(\log\log|\Sigma|)$ time per character.

4 Short Patterns

For very short strings $P_i \in D$ of length at most $2\log m$, we use the algorithm of Aho and Corasick [1]. More specifically, for the algorithm of Theorem 2, we use the version of Kopelowitz et al. [30], which takes $\mathcal{O}(\log\log|\Sigma|)$ time per character in the worst-case and uses $\mathcal{O}(d\log m)$ words of space. For the algorithm of Lemma 4, we use the DFA version of [1, Section 6], which takes $\mathcal{O}(1)$ time per character and uses $\mathcal{O}(d|\Sigma|\log m)$ words of space.

In this section, we introduce the first phase of $\mathcal{A}_1$, which deals with patterns of $D_1 = \{P_i \in D \mid 2\log m < m_i \leq 8d\log m\}$. Intuitively, at each position that is a multiple of $\log m$ the algorithm performs kind of a binary search for the longest text suffix which is a string in $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$, similarly to the *fat binary search* of Belazzougui et al. [8]. We define the *text fingerprint* of position q as $\phi(T[1..q])$. The algorithm maintains a sliding window of the last $8d\log m$ text fingerprints. Maintaining this window takes $\mathcal{O}(1)$ time per character using the sliding property of ϕ. Using this sliding window, for any $0 \leq \ell < 8d\log m$, one can compute the fingerprint of $T[q-\ell+1..q]$ in constant time. Hence, if the algorithm had all the fingerprints of all the suffixes of strings from $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$, the algorithm can easily perform the binary search where for each length it would make one query on a perfect hash table that maintains the suffixes' fingerprints. However, there exist too many such suffixes to maintain in a perfect hash table.

Let $i_{max} = \lceil \log_2 \min\{m, 8d\log m\}\rceil$ be the number of bits required to represent the lengths of patterns in D_1. The algorithm performs the binary search on the interval $[0, 2^{i_{max}}]$, such that at each iteration the length of range considered by the algorithm is a power of 2. For each $P \in \mathsf{Heads}_{[\log m,2\log m)}(D_1)$ the algorithm maintains the fingerprints of all the suffixes whose lengths may be queried by the binary search. These lengths are exactly the lengths whose binary representation is the same as the binary representation of $|P|$, except for some suffix of the representation that is replaced by zeros. Let $\Delta_{|P|} = \{|P| - (|P| \bmod 2^i) \mid 0 \leq i \leq i_{max}\}$ be the set of suffixes lengths for the string P. We define $\mathsf{Suffixes}_1 = \bigcup_{P\in\mathsf{Heads}_{[\log m,2\log m)}(D_1)}\{P[|P|-\ell+1..|P|] \mid \ell \in \Delta_{|P|}\}$ to be the set of all suffixes that may be queried by the binary search. Notice that $|\Delta_{|P|}| \leq i_{max} \leq \lceil\log m\rceil$ and therefore the total size of $\mathsf{Suffixes}_1$ is $\mathcal{O}(d\log^2 m)$. Given a perfect hash table that maintains the fingerprints of all the strings in $\mathsf{Suffixes}_1$, the algorithm is able to find the longest string in $\mathsf{Suffixes}_1$ that is a current suffix of the text by a binary search. At each iteration, the algorithm computes the fingerprint of some suffix of $T[1..q]$ and queries the hash table with this fingerprint to validate that this suffix is in $\mathsf{Suffixes}_1$. By maintaining with each $S \in \mathsf{Suffixes}_1$, the longest suffix of S from $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$, the algorithm is able to report in $\mathcal{O}(\log m)$ time the longest string from $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$ that is suffix of the text. However, storing such a perfect hash table takes $\mathcal{O}(d\log^2 m)$ words of space, which is too much. Thus, we have to reduce the space usage to $\mathcal{O}(d\log m)$.

Suffixes tree. In order to reduce the number of strings, we consider for each $S \in \mathsf{Suffixes}_1$ the string $p(S)$ which must precede S in the binary search as the *parent string* of S. Formally, if S is a suffix of some $P \in \mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ of length $|S| = \ell \in \Delta_{|P|}$ and $\ell' = \max\{\ell' \in \Delta_{|P|} \mid \ell' < \ell\}$ is the length preceding ℓ in $\Delta_{|P|}$ (if $\ell = 0$, which is the minimum value in $\Delta_{|P|}$ let $\ell' = 0$). Then, we define $p(S) = P[|P| - \ell' + 1..|P|]$ to be the suffix of S of length ℓ'. It is straightforward that $p(S) \in \mathsf{Suffixes}_1$. We define the *suffixes tree*, T_{suf}, as the tree induced by the strings and the parent string relation. Since each string has exactly one parent, except for the empty string, T_{suf} is well defined. Notice that a binary search that finds $S \in \mathsf{Suffixes}_1$ as an intermediate result, must consider all the strings on the path from the root to the node of S during its execution. Moreover, for each string S of length ℓ that is an intermediate result of the binary search, in the iteration after finding S, the binary search focuses on the range $[\ell, \ell + 2^{i(S)}]$ where $i(S) = \max\{i \mid \ell \bmod 2^{i-1} = 0\}$.

Compress T_{suf}. The algorithm traverses the tree from the root to the leaves. For any string S which has only one child, the algorithm shrinks the path from S to its first descendant S' that has at least two children, or S' is in $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$. The shrinking is done by setting $\mathsf{child}(S) = S'$ and removing all the strings between. Let $\mathsf{Suffixes}'_1$ be the set of remaining strings in the tree after the compression. The algorithm maintains a perfect hash table that maps any $S \in \mathsf{Suffixes}'_1$ into $i(S)$, and if S has only one child, S is associated also with the length $|\mathsf{child}(S)|$. In addition, S is associated with the index of the longest string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ that is a suffix of S as well.

Query processing. For each text position which is a multiple of $\log m$, the algorithm finds the current longest suffix from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ as follows. The algorithm initializes a length $\ell = 0$ and an exponent $i = i_{max} + 1$. At each iteration, it must be that the suffix of the text of length ℓ is in $\mathsf{Suffixes}'_1$, let denote this string as T_ℓ. On iterations where T_ℓ has multiple children (which can be retrieved from the hash table), the algorithm decrements i by one, computes the fingerprint of the text suffix of length $\ell + 2^i$ and queries the hash table with this fingerprint. If this fingerprint is maintained in the table then the length ℓ is updated to $\ell + 2^i$. On iterations where T_ℓ has only one child, the algorithm computes the fingerprint of the text of length $|\mathsf{child}(T_\ell)|$ and queries the hash table with this fingerprint. If this fingerprint is maintained in the table then the length ℓ is updated. Otherwise, the search is terminated. When $i < 0$, the search is also terminated.

The following lemma states that the longest suffix of $T[1..q]$ from $\mathsf{Suffixes}'_1$ is found by the binary search. Since $\mathsf{Heads}_{[\log m, 2\log m)}(D_1) \subseteq \mathsf{Suffixes}'_1$, it is guaranteed that if S is the current longest suffix of the text from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ then the algorithm finds a string S' such that S is a suffix of S'. Hence, since the algorithm reports the longest suffix of S' that is a string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$, this string must be S.

▶ **Lemma 10.** *When $T[q]$ arrives, for q which is a multiple of $\log m$, the first phase of $\mathcal{A}_1$ finds the longest suffix of $T[1..q]$ that is a string in $\mathsf{Suffixes}'_1$.*

Complexities. For every text character, the first phase of the algorithm just updates the sliding window of fingerprints, in constant time. For a text position that is a multiple of $\log m$, the binary search is performed in $\mathcal{O}(i_{max}) = \mathcal{O}(\log m)$ time. The algorithm maintains a sliding window of $\mathcal{O}(d \log m)$ text fingerprints. In addition, it stores a hash table that maintains a constant number of words per each string in $\mathsf{Suffixes}'_1$. By simple analysis, we have that $|\mathsf{Suffixes}'_1| = \mathcal{O}(d \log m)$, so, the total space usage of the first phase of $\mathcal{A}_1$ is $\mathcal{O}(d \log m)$ words of space.

5 Long Patterns

In this section, we treat the patterns whose length is at least $8d \log m$. The algorithm distinguishes between patterns with a small period length and those with a large period length. For each pattern P_i with length $m_i > 8d \log m$, we define Q_i to be the prefix of P_i of length $|Q_i| = m_i - (2d+2)\log m$. In Section 5.1, we introduce the first phase of algorithm $\mathcal{A}_{2a}$ for $D_{2a} = \{P_i \in D \,|\, m_i > 8d \log m \text{ and } \rho_{Q_i} \leq d \log m\}$. In Section 5.2, we introduce the first phase of algorithm $\mathcal{A}_{2b}$ for $D_{2b} = \{P_i \in D \,|\, m_i > 8d \log m \text{ and } \rho_{Q_i} > d \log m\}$.

5.1 Long Patterns with Short Periods

In this section, we introduce the first phase of $\mathcal{A}_{2a}$ which considers patterns P_i of length at least $8d \log m$, with $\rho_{Q_i} < d \log m$. Intuitively, we utilize the periodicity of the patterns prefixes and search for the same periodicity in the text in a sufficiently long substring. At each position, if a string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ ended in this position, there exists an occurrence of Q_i which ends at the $(2d+2)\log m$ preceding positions. In particular, since $\rho_{Q_i} < d \log m$, it must be that the text contains a long substring that has a period length ρ_{Q_i} that continues (at least) until the last $(2d+2)\log m$ positions. At any text position that is a multiple of $\log m$, by computing the fingerprint of the last $6d \log m$ characters, the algorithm determines a set of optional strings from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ whose suffix of length $6d \log m$ is a current suffix of the text. Notice that for each pair of such strings, one string must be the suffix of the other. This is because for each $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ there exists a Q_i, which is a prefix of S, and the suffix of S of length $6d \log m$ must contain at least two periods of Q_i, thus, this periodicity must continue until the suffix of S of length $6d \log m$. Therefore, to identify the longest text suffix that is a string in $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$, the algorithm finds the longest optional string that appears in the text due to the length of the periodic subtext. The main challenge is in maintaining the periodicity of the text in a manner that is easy to update (in constant time per character) and query. To tackle this challenge we utilize the combinatorial relationships between different Q_is.

For each $P_i \in D_{2a}$, we denote $\mathsf{prefix}(P_i) = P_i[1..2d \log m]$. Due to Lemma 6 we have that $\rho_{\mathsf{prefix}(P_i)} = \rho_{Q_i}$. So, if P_i occurs at position c of the text, then in particular, $\mathsf{prefix}(P_i)$ occurs at c, and by the periodicity of Q_i, $\mathsf{prefix}(P_i)$ occurs also at any position $c_k = c + k \cdot \rho_{\mathsf{prefix}(P_i)}$ for any positive integer k such that $c_k + |\mathsf{prefix}(P_i)| \leq c + |Q_i|$. To identify occurrences of prefixes of P_i from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ the algorithm searches for a sufficiently long arithmetic progression of $\mathsf{prefix}(P_i)$ occurrences in the text. Since all the $\mathsf{prefix}(\cdot)$ strings that the algorithm searches for are of the same length $2d \log m$, the algorithm is able to search them with a constant time per character by using the sliding window of the last $8d \log m$ text fingerprints and a perfect hash table of all the prefixes of patterns in D_{2a}.

Let $\mathsf{Prefixes}_{2a} = \{\mathsf{prefix}(P_i) \,|\, P_i \in D_{2a}\}$ be the set of all prefixes of length $2d \log m$, and let $\mathsf{p}_i, \mathsf{p}_j \in \mathsf{Prefixes}_{2a}$ be two strings from this set. We distinguish between two cases: in the first case, the prefixes p_i and p_j *agree* with each other, which means that between two close occurrences of one of them, there must exist an occurrence of the other. In the second case, p_i and p_j *disagree*, and whenever there exist two close occurrences of one of them, there is no occurrence of the other.

Formally, for each $\mathsf{p} \in \mathsf{Prefixes}_{2a}$, let $\alpha(\mathsf{p}) = \min\{\sigma^s(P[1..\rho_{\mathsf{p}}]) \,|\, s \in \{0, 1, \ldots, \rho_{\mathsf{p}} - 1\}\}$ be the *identify period* (also known as the Lyndon representation) of p, where $\sigma(\cdot)$ is the cyclic shift function (see Section 2), and the minimum is taken according to lexicographic order. The following lemma formalizes the possible relations between prefixes of patterns from D_{2a}.

▶ **Lemma 11.** *Let* $\mathsf{p}_i, \mathsf{p}_j \in \mathsf{Prefixes}_{2a}$*, and let* S *be the string of length* $2d\log m + \rho_{\mathsf{p}_i}$ *such that the prefix and suffix of* S *are both equal* p_i*. Then,* S *contains an occurrence of* p_j *if and only if* $\alpha(\mathsf{p}_i) = \alpha(\mathsf{p}_j)$*.*

Detect periodic substrings. In the preprocessing phase, we cluster the strings of $\mathsf{Prefixes}_{2a}$ according to their identify period. Let $\mathsf{Prefixes}^u_{2a} = \{\mathsf{p}_i \mid \mathsf{p}_i \in \mathsf{Prefixes}_{2a} \text{ and } \alpha(\mathsf{p}_i) = u\}$ be the cluster of prefixes with identify period u. We associate with each prefix $\mathsf{p}_i \in \mathsf{Prefixes}^u_{2a}$ a shift value $0 \leq s(\mathsf{p}_i) < \rho_{\mathsf{p}_i}$, such that the prefix of p_i of length $|u|$ is $\sigma^{s(\mathsf{p}_i)}(u)$. Let $\Psi = \{s(\mathsf{p}_i) \mid \mathsf{p}_i \in \mathsf{Prefixes}^u_{2a}\}$ be the set of all shift values of strings in $\mathsf{Prefixes}^u_{2a}$ and let $s_1 < s_2 < \cdots < s_h$ be the elements of Ψ ordered by increasing value. With each string $\mathsf{p}_i \in \mathsf{Prefixes}^u_{2a}$ such that $s(\mathsf{p}_i) = s_j$, the algorithm maintains the length $\delta(\mathsf{p}_i) = s_j - s_{j-1}$ (for $j = 0$ we have $\delta(\mathsf{p}_i) = |u| - s_h + s_1$), and the string $\mathsf{p}' \in \mathsf{Prefixes}^u_{2a}$ with shift value $s(\mathsf{p}') = s_{i-1}$ (for $i = 0$, $s(\mathsf{p}') = s_h$).

For each cluster $\mathsf{Prefixes}^u_{2a}$, we say that a string X is *u-periodic* if and only if $|X| \geq 2d\log m$, the prefix and suffix of length $2d\log m$ of X are strings in $\mathsf{Prefixes}^u_{2a}$, and the principal period length of X is $|u|$ (i.e., $\rho_X = |u|$).

In order to detect periodic substrings of the text, the algorithm maintains a sliding window of the last $8d\log m$ positions, where at each text position q, which is the end of some string $\mathsf{p}_i \in \mathsf{Prefixes}_{2a}$, the algorithm maintains (1) the id number of p_i and (2) the length of the *maximal* suffix of $T[1..q]$ that is $\alpha(\mathsf{p}_i)$-periodic string. Whenever a new character arrives, the algorithm computes the fingerprint of the current suffix of length $2d\log m$ and checks if it is a fingerprint of some $\mathsf{p}_i \in \mathsf{Prefixes}_{2a}$. If there exists such p_i, let $u = \alpha(\mathsf{p}_i)$ and let p' be the string preceded p_i in the cluster $\mathsf{Prefixes}^u_{2a}$ according to the cyclic shift. To compute the total length of the maximal u-periodic suffix of $T[1..q]$, the algorithm checks whether an occurrence of p' ended at $T[q - \delta(\mathsf{p}_i)]$. If such an occurrence exists, the length of the current maximal suffix is the sum of the length of the u-periodic suffix of $T[1..q - \delta(\mathsf{p}_i)]$ and $\delta(\mathsf{p}_i)$. If p' does not end at $T[q - \delta(\mathsf{p}_i)]$, then the length of the u-periodic sequence up to position q is exactly $2d\log m$.

Heads detection. For each $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$, we denote by $\mathsf{suffix}(S)$ the suffix of S of length $6d\log m$. The algorithm stores the fingerprints of all the strings in $\mathsf{Suffixes}_{2a} = \{\mathsf{suffix}(S) \mid S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})\}$ in a perfect hash table (notice that $\mathsf{Suffixes}_1$ and $\mathsf{Suffixes}_{2a}$ are two sets of heads suffixes, but their definitions are quite different). For each $\mathsf{s} \in \mathsf{Suffixes}_{2a}$, there exists a string $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ such that $\mathsf{s} = \mathsf{suffix}(S)$. By definition, we have that S is a prefix of some $P_i \in D_{2a}$ and Q_i is a prefix of S. Hence, due to the periodicity of Q_i, it must be that the prefix of s of length $3d\log m$ has a principal period length ρ_{Q_i}. Let v be the prefix of s of length $2d\log m$, and let $u = \alpha(v)$. By the periodicity, it must be that all the strings in $\mathsf{Prefixes}^u_{2a}$ appears in s. We denote by $\delta'(\mathsf{s})$ the distance between the last occurrence of some string from $\mathsf{Prefixes}^u_{2a}$ and the end of s. Formally, $\delta'(\mathsf{s}) = |\mathsf{s}| - \max\{j \mid \mathsf{s}[j - 2d\log m + 1..j] \in \mathsf{Prefixes}^u_{2a}\}$. Since there exists a string from $\mathsf{Prefixes}^u_{2a}$ that appears in s, it is obvious that $\delta'(\mathsf{s})$ is less than $6d\log m$. The following lemma proves that in order to detect occurrences of strings from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$, it suffices to detect $\mathsf{suffixes}$ and periodic substrings of the text.

▶ **Lemma 12.** *Let* $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ *we have that* $S = T[q - |S| + 1..q]$ *if and only if* $T[q - 6d\log m + 1..q] = \mathsf{suffix}(S)$ *and the length of maximal periodic suffix of* $T[1..q - \delta'(\mathsf{suffix}(S))]$ *is at least* $|S| - \delta'(\mathsf{suffix}(S))$*.*

Whenever the algorithm finds an occurrence of $\mathsf{s} \in \mathsf{Suffixes}_{2a}$ ended at position q, it might be the end of all the strings in $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ which their suffix is s. Let $\mathsf{Optional}_{\mathsf{s}} = \{S \mid S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a}) \text{ and } \mathsf{suffix}(S) = \mathsf{s}\}$ be the set of heads which are

optional occurrences when s occurs. It is straightforward that $|\mathsf{Optional}_{\mathsf{s}}| \leq m$ due to the periodicity of sufficiently long prefixes of D_{2a}. The algorithm stores all the indices of these strings indexed according to their length in a balanced binary tree. Due to Lemma 12, in order to detect the longest string from $\mathsf{Optional}_{\mathsf{s}}$ that is a suffix of the text, the algorithm has to find the longest string whose length is at most the sum of the length of the longest periodic suffix of $T[1..q - \delta'(\mathsf{s})]$ and $\delta'(\mathsf{s})$.

At each text position q that is a multiple of $\log m$, the algorithm computes the fingerprint of the last $6d \log m$ characters using the sliding window of text fingerprints. If this suffix is some $\mathsf{s} \in \mathsf{Suffixes}_{2a}$, the algorithm uses the sliding window of periodic suffixes to retrieve the length ℓ of the longest periodic suffix of $T[1..q - \delta'(\mathsf{s})]$. Then, the algorithm finds the predecessor of $\ell + \delta'(\mathsf{s})$ in the binary tree and reports the string associated with this length.

Complexities. The algorithm maintains the fingerprints of strings from $\mathsf{Prefixes}_{2a}$ and $\mathsf{Suffixes}_{2a}$, and since each of them is of size $\mathcal{O}(d \log m)$, we have $\mathcal{O}(d \log m)$ fingerprints and each of them is maintained within constant space. Moreover, the algorithm maintains sliding windows of fingerprints and periodic sequences information, each of them of length $\mathcal{O}(d \log m)$ and at each position the sliding windows use $\mathcal{O}(1)$ words of space. Therefore the first phase of $\mathcal{A}_{2a}$ uses $\mathcal{O}(d \log m)$ words of space. The first phase takes $\mathcal{O}(1)$ time per character for computing the fingerprint of the current $2d \log m$ suffix and an additional $\mathcal{O}(\log m)$ time per each text position that is a multiple of $\log m$, for the predecessor query.

5.2 Long Patterns with Long Periods

In this section, we introduce the first phase of $\mathcal{A}_{2b}$, which considers patterns P_i of length at least $8d \log m$, with $\rho_{Q_i} > d \log m$. An overview of the algorithm is as follows. First, the algorithm finds all the occurrences of strings Q_i, with a delay of at most $d \log m$ characters. Each occurrence of Q_i is a possible occurrence of P_i, so the algorithm computes the position q that is a multiple of $\log m$ and is in the range of $[\log m, 2 \log m)$ positions preceding the end of the possible occurrence of P_i. Then, the algorithm computes the expected text fingerprint in this position if this occurrence of Q_i is indeed an occurrence of P_i. The expected text fingerprint is maintained in a designated data structure. When $T[q]$ arrives, the algorithm checks if the text fingerprint matches the expected fingerprint, and if so, it reports the appropriate prefix of P_i from $\mathsf{Heads}_{[\log m, 2 \log m)}(D_{2b})$.

Finding Q_i with a delay. The algorithm finds all the occurrences of any Q_i in the text. The algorithm uses a similar technique to algorithm $\mathcal{A}_{2b}$ of Clifford et al. [12], which creates $\mathcal{O}(\log m)$ *levels* for each pattern in the dictionary. Each level maintains occurrences of a prefix of the pattern of length which is a power of two. The algorithm finds for each pattern occurrences of the shortest prefix of the pattern whose principal period length is greater than $d \log m$, by applying algorithms $\mathcal{A}_1$ and $\mathcal{A}_{2a}$. Thus, all the longer prefixes have at least $d \log m$ characters between any two occurrences, and therefore by a round-robin fashion their levels are treated in $\mathcal{O}(1)$ time per character. The algorithm has the guarantee that each occurrence of any Q_i in the text is found by the algorithm with a delay of at most $d \log m$ text characters. The complexities of this part are according to Theorem 2 or Lemma 4 and the complete details will appear in the final version of this paper.

Extend Q_i to prefix from $\mathsf{Heads}_{[\log m, 2 \log m)}(D_{2b})$. After finding each Q_i with a delay of at most $d \log m$ text characters, the goal is to find at each position q that is a multiple of $\log m$, the longest suffix of the text that is a string from $\mathsf{Heads}_{[\log m, 2 \log m)}(D_{2b})$. Let c

be a text index where an occurrence of Q_i begins. Thus, c is a possible occurrence of P_i,[4] and the algorithm must validate it. The algorithm computes q_c that is the text index in the range $q_c \in (c + m_i - 1 - 2\log m, c + m_i - 1 - \log m]$ which is a multiple of $\log m$. Since the length of the range is $\log m$, there is exactly one such position. Let S be the prefix of P_i of length $\ell = q_c - c + 1$, by definition $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$. The algorithm computes the expected fingerprint of the text in position q_c from $\phi(T[1..c-1])$ and $\phi(S)$. This fingerprint, together with the index of S is maintained in a data structure associated with position q_c.

The algorithm maintains a sliding window of the subsequent $(2d+1)$ positions that are multiples of $\log m$. Since $m_i - 2\log m < |S| \leq m_i - \log m$ and $|Q_i| = m_i - (2d+2)\log m$, the distance (in text characters) between the end of the Q_i's occurrence and q_c is in the range $(2d\log m, (2d+1)\log m]$. Hence, q_c is in the sliding window. For each position q in the sliding window, the algorithm maintains an AVL tree, which maintains expected text fingerprints. Each expected fingerprint is maintained with an id number of the corresponding string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$. When a new occurrence of some Q_i is found, the algorithm inserts into the AVL tree the expected fingerprint it computes, associated with the id number of the corresponding $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$. If this fingerprint already exists in the tree, the algorithm updates the corresponding string to be the longest between the existing string and the new string.

Since $\rho_{Q_i} > d\log m$, at any sequence of $2d\log m$ characters, there exist at most 2 occurrences of Q_i. Thus, the total number of values inserted into the trees in a sequence of $2d\log m$ characters is at most $\mathcal{O}(d)$. In addition, for each position q in the sliding window the number of elements in the AVL tree is at most d, hence, the insertion takes $\mathcal{O}(\log d)$ time. So, the total time of insertions in a sequence of $2d\log m$ characters is $\mathcal{O}(d\log d)$. Since $D_{2b} \neq \emptyset$ we have some strings of length at least $8d\log m$, and therefore $m > d$ and thus $\mathcal{O}(d\log d) = \mathcal{O}(d\log m)$. Using the round-robin fashion, all the insertions to the trees are de-amortized to $\mathcal{O}(1)$ time per character. The round-robin fashion may create a delay of at most $d\log m$ text characters. Recall that any occurrence of Q_i is found with a delay of at most $d\log m$ characters and that the distance between the end of the Q_i's occurrence and q_c is at least $2d\log m$. Thus, there exists at least $d\log m$ characters between the recognizing of Q_i's occurrence and q_c. Hence, when $T[q]$ arrives, its expected fingerprints tree contains all the expected fingerprints corresponding to occurrences of all Q_is which their q_c is q.

When $T[q]$ arrives, for q that is a multiple of $\log m$, the algorithm searches for the current text fingerprint $\phi(T[1..q])$ in the AVL tree of position q. The time required for this search is $\mathcal{O}(\log d) = \mathcal{O}(\log m)$. In the following lemma, we prove that the algorithm indeed finds the longest suffix of $T[1..q]$ from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$ for every q that is a multiple of $\log m$.

▶ **Lemma 13.** *When $T[q]$ arrives, for q which is a multiple of $\log m$, the first phase of $\mathcal{A}_{2b}$ finds the longest suffix of $T[1..q]$ that is a string in $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$.*

Complexities. Since all the usage of round-robin fashion described above is on ranges of size $\mathcal{O}(d\log m)$, the de-amortization uses $\mathcal{O}(d\log m)$ words of space. Hence, summing all the parts, the time and space of the first phase of $\mathcal{A}_{2b}$ are as stated in Theorem 2 or Lemma 4 with $\mathcal{O}(\log m)$ additional time for any position which is a multiple of $\log m$.

[4] For the sake of simplicity, we assume that for any two different patterns $P_i, P_j \in D_{2b}$, we have $Q_i \neq Q_j$. Otherwise, we treat each occurrence of Q_i multiple times, each time as the prefix of another $P_i \in D_{2b}$.

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
2. Mansoor Alicherry, Muthusrinivasan Muthuprasanna, and Vijay Kumar. High speed pattern matching for network IDS/IPS. In *Proceedings of the 14th IEEE International Conference on Network Protocols, ICNP*, pages 187–196, 2006. doi:10.1109/ICNP.2006.320212.
3. Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.
4. Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *27th International Symposium on Algorithms and Computation, ISAAC 2016*, pages 12:1–12:12, 2016. doi:10.4230/LIPIcs.ISAAC.2016.12.
5. Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Dictionary matching with one gap. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM*, pages 11–20, 2014.
6. Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Dictionary matching with a few gaps. *Theor. Comput. Sci.*, 589:34–46, 2015.
7. Tanver Athar, carl Barton, Widmer bland, Jia Gao, Costas S. Illopoulos, Chang Liu, and Solon P. Pissis. Fast circular dictionary-matching algorithm. *Mathematical Structures in Computer Science*, pages 1–14, 2015.
8. Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009*, pages 785–794, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496856.
9. Anat Bremler-Barr, David Hay, and Yaron Koral. Compactdfa: Scalable pattern matching using longest prefix match solutions. *IEEE/ACM Trans. Netw.*, 22(2):415–428, 2014. doi:10.1109/TNET.2013.2253119.
10. Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.
11. Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. *Theor. Comput. Sci.*, 483:2–9, 2013. doi:10.1016/j.tcs.2012.11.040.
12. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *Proceedings of Annual European Symposium on Algorithms, ESA*, pages 361–372, 2015.
13. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. The k-mismatch problem revisited. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 2039–2052, 2016. doi:10.1137/1.9781611974331.ch142.
14. Funda Ergün, Hossein Jowhari, and Mert Saglam. Periodicity in streams. In *Proceedings of Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 13th International Workshop, APPROX 2010, and 14th International Workshop, RANDOM 2010*, pages 545–559, 2010. doi:10.1007/978-3-642-15369-3_41.
15. Yu Fang, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proceedings of 12th IEEE International Conference on Network Protocols (ICNP 2004)*, pages 174–183, 2004. doi:10.1109/ICNP.2004.1348108.
16. Guy Feigenblat, Ely Porat, and Ariel Shiftan. An improved query time for succinct dynamic dictionary matching. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM*, pages 120–129, 2014.

17 Guy Feigenblat, Ely Porat, and Ariel Shiftan. Linear time succinct indexable dictionary construction with applications. In *Data Compression Conference , DCC*, pages 13–23, 2016.

18 Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proceedings of Algorithms - ESA 2015 - 23rd Annual European Symposium*, pages 533–544, 2015. doi:10.1007/978-3-662-48350-3_45.

19 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

20 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. Space-efficient dictionaries for parameterized and order-preserving pattern matching. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM*, pages 2:1–2:12, 2016.

21 Arnab Ganguly, Wing-Kai Hon, and Rahul Shah. A framework for dynamic parameterized dictionary matching. In *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT*, pages 10:1–10:14, 2016.

22 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Streaming Pattern Matching with d Wildcards. In *24th Annual European Symposium on Algorithms (ESA)*, pages 44:1–44:16, 2016.

23 Torben Hagerup. Sorting and searching on the word RAM. In *STACS 98, 15th Annual Symposium on Theoretical Aspects of Computer Science, Paris, France, February 25-27, 1998, Proceedings*, pages 366–398, 1998. doi:10.1007/BFb0028575.

24 Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001. doi:10.1006/jagm.2001.1171.

25 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan. *External Memory Algorithms*, chapter Computing on data streams, pages 107–118. American Mathematical Society, 1999.

26 Cheng-Liang Hsieh, Lucas Vespa, and Ning Weng. A high-throughput DPI engine on GPU via algorithm/implementation co-optimization. *J. Parallel Distrib. Comput.*, 88:46–56, 2016. doi:10.1016/j.jpdc.2015.11.001.

27 Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *proceedings of 30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013*, pages 400–411, 2013. doi:10.4230/LIPIcs.STACS.2013.400.

28 Daniel M. Kane, Jelani Nelson, Ely Porat, and David P. Woodruff. Fast moment estimation in data streams in optimal space. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011*, pages 745–754, 2011. doi:10.1145/1993636.1993735.

29 Junghak Kim and Song-in Choi. High speed pattern matching for deep packet inspection. In *Communications and Information Technology, 2009. ISCIT 2009. 9th International Symposium on*, pages 1310–1315. IEEE, 2009.

30 Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct Online Dictionary Matching with Improved Worst-Case Guarantees. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2016.6.

31 Lap-Kei Lee, Moshe Lewenstein, and Qin Zhang. Parikh matching in the streaming model. In *Proceedings of String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012*, pages 336–341, 2012. doi:10.1007/978-3-642-34109-0_35.

32 S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005. doi:10.1561/0400000002.

33 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Proceedings of 50th Annual IEEE Symposium on Foundations of Computer*

Science, FOCS 2009, October 25-27, 2009, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

34 Yaron Rozen. On the online dictionary matching problem. Master's thesis, Bar-Ilan University, Ramat Gan, Israel, September 2016. Under the supervision of Prof. Ely Porat.

35 Milan Ruzic. Constructing efficient dictionaries in close to sorting time. In *Proceedings of Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, pages 84–95, 2008. doi:10.1007/978-3-540-70575-8_8.

36 Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings IEEE INFOCOM 2004, The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, 2004. URL: http://www.ieee-infocom.org/2004/Papers/54_5.PDF.

37 Antonino Tumeo, Oreste Villa, and Daniel G. Chavarría-Miranda. Aho-corasick string matching on shared and distributed-memory parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 23(3):436–443, 2012. doi:10.1109/TPDS.2011.181.

38 Antonino Tumeo, Oreste Villa, and Donatella Sciuto. Efficient pattern matching on gpus for intrusion detection systems. In *Proceedings of the 7th Conference on Computing Frontiers, 2010*, pages 87–88, 2010. doi:10.1145/1787275.1787296.

39 Lucas Vespa and Ning Weng. GPEP: graphics processing enhanced pattern-matching for high-performance deep packet inspection. In *2011 IEEE International Conference on Internet of Things (iThings) & 4th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 74–81, 2011. doi:10.1109/iThings/CPSCom.2011.36.

40 Lucas Vespa and Ning Weng. Swm: Simplified wu-manber for gpu-based deep packet inspection. In *Proceedings of the International Conference on Security and Management (SAM)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

41 Yaron Weinsberg, Shimrit Tzur-David, Danny Dolev, and Tal Anker. High performance string matching algorithm for a network intrusion prevention system (nips). In *High Performance Switching and Routing, 2006 Workshop on*, pages 7–pp. IEEE, 2006.

42 SangKyun Yun. An efficient tcam-based implementation of multipattern matching using covered state encoding. *IEEE Transactions on Computers*, 61(2):213–221, 2012.

43 Xinyan Zha and Sartaj Sahni. Gpu-to-gpu and host-to-host multipattern string matching on a GPU. *IEEE Trans. Computers*, 62(6):1156–1169, 2013. doi:10.1109/TC.2012.61.