Parameterized Matching in the Streaming Model

Markus Jalsenius[1], Benny Porat[2], and Benjamin Sach[3]

1 **Department of Computer Science, University of Bristol, U.K.**
2 **Department of Computer Science, Bar-Ilan University, Israel.**
3 **Department of Computer Science, University of Warwick, U.K.**

Abstract

We study the problem of parameterized matching in a stream where we want to output matches between a pattern of length m and the last m symbols of the stream before the next symbol arrives. Parameterized matching is a natural generalisation of exact matching where an arbitrary one-to-one relabelling of pattern symbols is allowed. We show how this problem can be solved in constant time per arriving stream symbol and sublinear, near optimal space with high probability. Our results are surprising and important: it has been shown that almost no streaming pattern matching problems can be solved (not even randomised) in less than $\Theta(m)$ space, with exact matching as the only known problem to have a sublinear, near optimal space solution. Here we demonstrate that a similar sublinear, near optimal space solution is achievable for an even more challenging problem.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Pattern matching, streaming algorithms, randomized algorithms

Digital Object Identifier 10.4230/LIPIcs.STACS.2013.400

1 Introduction

We consider the problem of pattern matching in a stream where we want to output matches between a pattern of length m and the last m symbols of the stream. Each answer must be reported before the next symbol arrives. The problem we consider in this paper is known as *parameterized matching* and is a natural generalisation of exact matching where an arbitrary one-to-one relabelling of the pattern symbols is allowed (one per alignment). For example, if the pattern is `abbca` then there there is a parameterized match with `bddcb` as we can apply the relabelling `a`→`b`, `b`→`d`, `c`→`c`. There is however no parameterized match with `bddbb`. We show how this streaming pattern matching problem can be solved in near constant time per arriving stream symbol and sublinear, near optimal, space with high probability. The space used is reduced even further when only a small subset of the symbols are allowed to be relabelled. As discussed in the next section, our results demonstrate a serious push forward in understanding what pattern matching algorithms can be solved in sublinear space.

1.1 Background

Streaming algorithms is a well studied area and specifically finding patterns in a stream is a fundamental problem that has received increasing attention over the past few years. It was shown in [8] that many offline algorithms can be made online (streaming) and deamortised with a $\log m$ factor overhead in the time complexity per arriving symbol in the stream, where m is the length of the pattern. There have also been improvements for specific pattern matching problems but they all have one property in common: space usage is $\Theta(m)$ words. It is not difficult to show that we in fact *need* as much as $\Theta(m)$ space to do pattern

© Markus Jalsenius, Benny Porat, and Benjamin Sach;
licensed under Creative Commons License BY-ND
30th Symposium on Theoretical Aspects of Computer Science (STACS'13).
Editors: Natacha Portier and Thomas Wilke; pp. 400–411
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE

matching, unless errors are allowed. The field of pattern matching in a stream took a significant step forwards in 2009 when it was shown to be possible to solve exact matching using only $O(\log m)$ words of space and $O(\log m)$ time per new stream symbol [15]. This method, which is based on fingerprints, correctly finds all matches with high probability. The initial approach was subsequently somewhat simplified [10] and then finally improved to run in constant time [7] within the same space requirements.

Being able to do exact matching in sublinear space raised the question of what other streaming pattern matching problems can be solved in small space. In 2011 this question was answered for a large set of such problems [9]. The result was rather gloomy: almost no streaming pattern matching problems can be solved in sublinear space, not even using randomised algorithms. An $\Omega(m)$ space lower bound was given for L_1, L_2, L_∞, Hamming, edit distance and pattern matching with wildcards as well as for any algorithm that computes the cross-correlation/convolution. So what other pattern matching problems could possibly be solved in small space? It seems that the only hope to find any is by imposing various restrictions on the problem definition. This was indeed done in [15] where a solution to k-mismatch (exact matching where up to k mismatches are allowed) was given which uses $O(k^2\text{poly}(\log m))$ time per arriving stream symbol and $O(k^3\text{poly}(\log m))$ words of space. The solution involves multiple instances of the exact matching algorithm run in parallel. Note that the space bound approaches $\Theta(m)$ as k increases, so the algorithm is only interesting for sufficiently small k. Further, the space bound is very far from the known $\Omega(k)$ lower bound. We also note that it is straightforward to show that exact matching with k wildcards in the pattern can be solved with the k-mismatch algorithm. To our knowledge, no other streaming pattern matching have been solved in sublinear space so far.

In this paper we present the first push forward since exact matching by giving a sublinear, near optimal space and near constant time algorithm for parameterized matching in a stream. This natural problem turns out to be significantly more complicated to solve than exact matching and our results provide the first demonstration that small space and time bounds are achievable for a more challenging problem. Note that our space bound, as opposed to k-mismatch, is essentially optimal like for exact matching. One could easily argue that our results are surprising, and yet again the question of what other problems are solvable in sublinear space calls for an answer. In particular, given that restrictions to the problem have to be made, what restrictions should one make to break the $\Omega(m)$ space barrier.

1.2 Problem definition and related work

A pattern P of length m is said to *parameterized match*, or *p-match* for short, an m length string S if there is an injective (one-to-one) function f such that $S[j] = f(P[j])$ for all $j \in \{0, \ldots, m-1\}$. In our streaming setting, the pattern is known in advance and the symbols of the stream T arrive one at a time. We use the letter i to denote the index of the latest symbol in the stream. Our task is to output whether there is a p-match between P and $T[(i-m+1),\ i]$ before $T[i+1]$ arrives. The mapping f may be distinct for each i.

One may view this matching problem as that of finding matches in a stream encrypted using a substitution cipher. In offline settings, p-matching has its origin in finding duplication and plagiarism in software code although has since found numerous other applications. Since the first introduction of the problem, a great deal of work has gone into its study in both theoretical and practical settings (see e.g. [3, 1, 4, 5, 6, 12]). Notably, in an offline setting, the exact p-matching problem can be solved in near linear time using a variant [1] of the classic linear time exact matching algorithm KMP [14].

When the sublinear space algorithm for exact matching was given in [15], properties of the periods of strings formed a crucial part of their analysis. However, when considering p-matching the period of a string is a much less straightforward concept than it is for exact matching. For example, it is no longer true that consecutive matches must either be separated by the period of the pattern or be at least $m/2$ symbols apart. This property, which holds for exact but not p-matching, allows for an efficient encoding of the positions of the matches. This was crucial to reducing the space requirements of the previous streaming algorithms. Unfortunately, p-matches can occur at arbitrary positions in the stream, requiring new insights. This is not the only challenge that we face.

A natural way to match two strings under parameterization is to consider their *predecessor strings*. For a string S, the predecessor string, denoted $\text{pred}(S)$, is a string of length $|S|$ such that $\text{pred}(S)[j]$ is the distance, counted in numbers of symbols, to the previous occurrence of the symbol $S[j]$ in S. In other words, $\text{pred}(S)[j] = d$, where d is the smallest positive value for which $S[j] = S[j-d]$. Whenever no such d exists, we set $\text{pred}(S)[j] = 0$. As an example, if $S =$ `aababcca` then $\text{pred}(S) =$ `01022014`. We can perform p-matching offline by only considering predecessor strings using the fundamental fact [3] that two equal length strings S and S' p-match iff $\text{pred}(S) = \text{pred}(S')$. A plausible approach for our streaming problem would now be to translate the problem of p-matching in a stream to that of exact matching. This could be achieved by converting both pattern and stream into their corresponding predecessor strings and maintaining fingerprints of a sliding window of the translated input. However, consider the effect on the predecessor string, and hence its fingerprint, of sliding a window in the stream along by one. The leftmost symbol x, say, will move out of the window and so the predecessor value of the new leftmost occurrence of x in the new window will need to be set to `0` and the corresponding fingerprint updated. We cannot afford to store the positions of all characters in a $\Theta(m)$ length window.

We will show a matching algorithm that solves these problems and others we encounter en route using minimal space and in near constant time per arriving symbol. A number of technical innovations are required, including new uses of fingerprinting, a new compressed encoding of the positions of potential matches, a separate deterministic algorithm designed for prefixes of the pattern with small parameterized period as well as the deamortisation of the entire matching process. Section 2 gives a more detailed overview of these main hurdles.

1.3 Our new results

Our main result is a fast and space efficient algorithm for the streaming p-matching problem. It applies to *dense* alphabets where both the pattern and streaming text alphabets are $\Sigma = \{0, \ldots, |\Sigma| - 1\}$. Theorem 1 below is proved over the subsequent sections of this paper. Some supporting proofs are left for the full paper due to space constraints, including the proof of Theorem 2, which is based on communication complexity arguments.

▶ **Theorem 1.** *Let the pattern have length m, the text have length n and both have alphabets $\Sigma = \{0, \ldots, |\Sigma| - 1\}$. There is a randomised algorithm for streaming p-matching that takes $O(1)$ worst-case time per character and uses $O(|\Sigma| \log m)$ words of space. The probability that the output is correct at all text alignments is at least $1 - 1/n^c$ for any constant c.*

▶ **Theorem 2.** *There is a randomised space lower bound of $\Omega(|\Sigma|)$ bits for the streaming p-matching problem, where Σ is the pattern alphabet.*

Parameterized matching is often specified under the assumption that only some symbols are variable (allowed to be relabelled). The mapping f we used in Section 1.2 has to reflect this constraint. More precisely, let the pattern alphabet be partitioned into fixed symbols

Σ_{fixed} and variable symbols Π. For $\sigma \in \Sigma_{\text{fixed}}$, we require that $f(\sigma) = \sigma$. The result from Theorem 1 can be extended to handle *general* alphabets with arbitrary fixed symbols. The idea is to apply a suitable reduction that was given in [1] (Lemma 2.2) together with the streaming exact matching algorithm of Breslauer and Galil [7], as well as applying a "filter" on the text stream using a dynamic dictionary (for instance that of [2]). The dictionary is used to map text symbols to the variable pattern symbols in Π. The proof is omitted.

▶ **Theorem 3.** *Suppose Π is the set of pattern symbols that can be relabelled under p-matching. All other pattern symbols are fixed. Without any constraints on the text alphabet, there is a randomised algorithm for streaming p-matching that takes $O(\sqrt{\log|\Pi|/\log\log|\Pi|})$ worst-case time per character and uses $O(|\Pi|\log m)$ words of space, where m is the length of the pattern. The probability that the algorithm outputs correctly at all alignments of an n length text is at least $1-1/n^c$, where c is any constant.*

As part of the proof of Theorem 1 we had to develop an algorithm that efficiently solves streaming p-matching for patterns with small *parameterized period*. The parameterized period (*p-period*) of the pattern P, denoted ρ, is the smallest positive integer such that $P[0,\ (m-1-\rho)]$ p-matches $P[\rho,\ m-1]$. That is, ρ is the shortest distance that P must be slid by to p-match itself. Our algorithm is deterministic and is interesting in its own right (see Section 4). We also provide an almost matching space lower bound (proof omitted).

▶ **Theorem 4.** *Suppose the pattern and text alphabets are both $\Sigma = \{0, \ldots, |\Sigma|-1\}$ and the pattern has p-period ρ. There is a deterministic algorithm for streaming p-matching that takes $O(1)$ worst-case time per character and uses $O(|\Sigma|+\rho)$ words of space. Further, there is a deterministic space lower bound of $\Omega(|\Sigma|+\rho)$ bits.*

1.4 Fingerprints

We will make extensive use of Rabin-Karp style fingerprints of strings which are defined as follows. Let S be a string over the alphabet Σ. Let $p > |\Sigma|$ be a prime and choose $r \in \mathbb{Z}_p$ uniformly at random. The fingerprint $\phi(S)$ is given by $\phi(S) \overset{\text{def}}{=} \sum_{k=0}^{|S|-1} S[k]r^k \bmod p$. A critical property of the fingerprint function ϕ is that the probability of achieving a false positive, $\Pr(\phi(S) = \phi(S') \wedge S \neq S')$, is at most $|S|/(p-1)$ (see [13, 15] for proofs). Let n denote the total length of the stream. Our randomised algorithm will make $o(n^2)$ (in fact near linear) fingerprint comparisons in total. Therefore, by the applying the union bound, for any constant c, we can choose p to be of size $\Theta(n^{c+3})$ so that with probability at least $1-1/n^c$ there will be no false positive matches.

As we assume the RAM model with word size $\Theta(\log n)$, a fingerprint fits in a constant number of words. We assume that all fingerprint arithmetic is performed within $\mathbb{Z}_p$. In particular we will take advantage of two fingerprint operations.

$\ominus$ *Splitting:* Given $\phi\big(S[0,\ a]\big)$, $\phi\big(S[0,\ b]\big)$ (where $b > a$) and the value of $r^{-a} \bmod p$, we can compute $\phi\big(S[a+1,\ b]\big) = \phi\big(S[0,\ b]\big) \ominus \phi\big(S[0,\ a]\big)$ in $O(1)$ time.

$\circledcirc$ *Zeroing:* Let S, S' be two equal length strings such that S' is identical to S except for in positions $z \in Z \subseteq [0, s-1]$ at which $S'[z] = 0$. We write $\phi\big(S\big) \circledcirc Z$ to denote $\phi\big(S'\big)$. Given $\phi\big(S\big)$ and $(S[z],\ r^z \bmod p)$ for all $z \in Z$, computing $\phi\big(S\big) \circledcirc Z$ takes $O(|Z|)$ time.

2 Overview, key properties and notation

The overall idea of our algorithm in Theorem 1 follows that of previous work on streaming exact matching in small space, however for p-matching the situation is much more complex

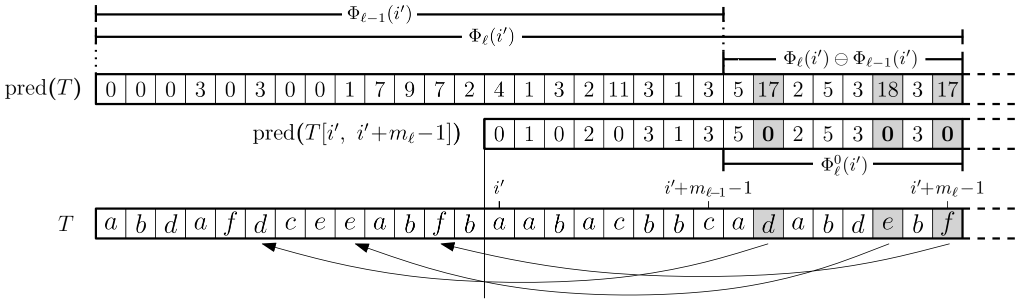

Figure 1 The key fingerprints used by the randomised algorithm. Characters contributing differently to $\Phi^0_\ell(i')$ and $\Phi_\ell(i') \ominus \Phi_{\ell-1}(i')$ are highlighted.

and calls for not only more involved details and methods but also a deep fundamental understanding of the nature of p-matching. We will now describe the overall idea, introduce some important notation and at the end of this section we will highlight key facts about p-matching that are crucial for our solution.

The main algorithm will try to match the streaming text with various prefixes of the pattern P. Let Σ_P denote the pattern alphabet. We define $\delta = |\Sigma_\text{P}| \log m$ and let P_0 denote the shortest prefix of P that has p-period greater than 3δ (recall the definition of p-period given above Theorem 4). We define s prefixes P_ℓ of increasing length so that $|P_\ell| = 2^\ell |P_0|$ for $\ell \in \{1, \ldots, s-1\}$, where $s \leqslant \lceil \log m \rceil$ is the largest value such that $|P_{s-1}| \leqslant m/2$. The final prefix P_s has length $m - 4\delta$. For all ℓ, we define $m_\ell = |P_\ell|$, hence $m_\ell = 2m_{\ell-1}$.

In order to determine if there is a p-match between the text and a pattern prefix, we will compare the fingerprints of their predecessor strings (recall that two strings p-match iff their predecessor strings are the same). We will need two related (but typically distinct) fingerprint definitions to achieve this. Figure 1 will be helpful when reading the following definitions which are discussed in an example below. For any index i' and $\ell \in \{0, \ldots, s\}$,

$$\Phi_\ell(i') \stackrel{\text{def}}{=} \phi\big(\text{pred}(T[0,\ (i'+m_\ell-1)])\big)\,,$$
$$\Phi^0_\ell(i') \stackrel{\text{def}}{=} \phi\big(\text{pred}(T[i',\ (i'+m_\ell-1)])[m_{\ell-1},\ m_\ell-1]\big)\,.$$

For each $\ell \in \{1, \ldots, s\}$ the main algorithm runs a process whose responsibility for finding p-matches between the text and P_ℓ (P_0 is handled separately as will be discussed later). The process responsible for P_ℓ will ask the process responsible for $P_{\ell-1}$ if it has found any p-matches, and if so it will try to extend the matches to P_ℓ. As an example, suppose that the process for $P_{\ell-1}$ finds a match at position i' of the text (refer to Figure 1). The process will then store this match along with the fingerprint $\Phi_{\ell-1}(i')$ which has been built up as new symbols arrive. The process for P_ℓ will be handed this information when the symbol at position $i'+m_\ell-1$ arrives. The task is now to work out if i' is also a matching position with P_ℓ. With the fingerprint $\Phi_\ell(i')$ available (built up as new symbols arrive), the process for P_ℓ can use fingerprint arithmetics to determine if i' is a matching position. This is one instance where the situation becomes more tricky than one might first think.

As position i' is a p-match with $P_{\ell-1}$ it suffices to compare the second half of the predecessor string of P_ℓ with the second half of the predecessor string of $T[i',\ i'+m_\ell-1]$. Fingerprints are used for this comparison. It is crucial to understand that $\Phi_\ell(i') \ominus \Phi_{\ell-1}(i')$ cannot be used directly here; some predecessor values of the text might point very far back, namely to some position *before* index i'. In Figure 1 we have shaded the three symbols for which this is true and have drawn arrows indicating their predecessors. Thus, in order to

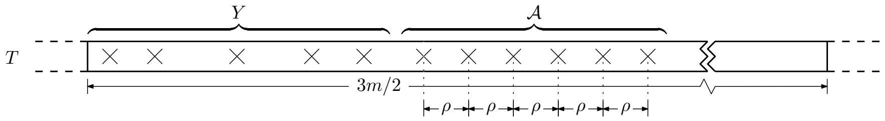

Figure 2 Partitioning of positions (×) at which P p-matches in a $3m/2$ length substring of T.

correctly do the fingerprint comparison we need to set those positions to zero (we want the fingerprint of the predecessor string of the text substring starting at position i', not the beginning of T). The fingerprint we defined as $\Phi_\ell^0(i')$ above is the fingerprint we want to compare to the fingerprint of the second half of the predecessor string of P_ℓ. Using fingerprint operations, we have from the definitions that $\Phi_\ell^0(i') = \big(\Phi_\ell(i') \ominus \Phi_{\ell-1}(i')\big) \circledcirc \Delta_\ell(i')$, where $\Delta_\ell(i')$ is the set of positions that have to be set to zero. For a substring of T of length $\Theta(m_{\ell-1})$ consider the subset of positions which occur in $\Delta_\ell(i')$ for at least one value of i'. Any such position has a predecessor value greater than $m_{\ell-1}$. By summing over all distinct symbols we have that the size of this subset is only $O(|\Sigma_\mathrm{P}|)$. Thus, we can maintain in small space every position in a suitable length window that will *ever* have to be set to zero.

Let us go back to the example where the process for $P_{\ell-1}$ had found a p-match at position i'. The process stores i' along with the fingerprint $\Phi_{\ell-1}(i')$. This information is not needed by the process for P_ℓ until $m_{\ell-1}$ text symbols later. During the arrival of these symbols, the process for $P_{\ell-1}$ might detect more p-matches, in fact many more matches. Their positions and corresponding fingerprints have to be stored until needed by the process for P_ℓ. We now have a space issue: how do we store this information in small space? To appreciate this question, first consider exact matching. Here matches are known to be either an exact period length apart or very far apart. The matching positions can therefore be represented by an arithmetic progression. Further, the fingerprints associated with the matches in an arithmetic progression can easily be stored succinctly as one can work out each one of the fingerprints from the first one. For p-matching the situation is much more complex: matches can occur more chaotically and, as we have seen above, fingerprints must be updated dynamically to reflect that symbols could be mapped differently in two distinct alignments. Handling these difficulties in small space (and small time complexity) is a main hurdle and is one point at which our work differ significantly from all previous work on streaming matching in small space. We cope with this space issue in the next section.

2.1 The structure of parameterized matches

First recall that an *arithmetic progression* is a sequence of numbers such that the (common) difference between any two successive numbers is constant. We can specify an arithmetic progression by its start number, the common difference and the length of the sequence. In the next lemma we will see that the positions at which a string P of length m p-matches a longer string of length $3m/2$ can be stored in small memory: either a matching position belongs to an arithmetic progression or it is one of relatively few positions that can be listed explicitly in $O(|\Sigma_\mathrm{P}|)$ space. The proof (consult Figure 2) is deferred to Section 5.

▶ **Lemma 5.** *Let X be the set of positions at which P p-matches within an $3m/2$ length substring of T. The set X can be partitioned into two sets Y and $\mathcal{A}$ such that $|Y| \leqslant 6|\Sigma_\mathrm{P}|$, $\max(Y) < \min(\mathcal{A})$ and $\mathcal{A}$ is an arithmetic progression with common difference ρ, where ρ is the p-period of P.*

The lemma is incredibly important for the algorithm as it allows us to store all partial matches (that need to be kept in memory before being discarded) in a total of $O(|\Sigma_\mathrm{P}|\log m)$

space across all processes. The question of how to store their associated fingerprints remains, but is nicely resolved with the corollary below that follows immediately from the proof of Lemma 5. We can afford to store fingerprints explicitly for the positions that are identified to belong to the set Y from Lemma 5, and for the matching positions in the arithmetic progression $\mathcal{A}$ we can, as for exact matching, work out every fingerprint given the first one.

▶ **Corollary 6.** *For pattern P, text T and arithmetic progression $\mathcal{A}$ as specified in Lemma 5,* $\mathrm{pred}(T)[(i+m-\rho),\ (i+m-1)]$ *is the same for all $i \in \mathcal{A}$.*

2.2 Deamortisation

So far we have described the overall approach but it is of course a major concern how to carry out computations in constant time per arriving symbol. In order to *deamortise* the algorithm, we run a separate process responsible for the pattern prefix P_0 that uses the deterministic algorithm of Section 4 (i.e. Theorem 4). As P_0 has p-period greater than 3δ, the p-matches it outputs are at least this far apart. This enables the other processes to operate with a small delay: process P_ℓ expects process $P_{\ell-1}$ to hand over matches and fingerprints with a small delay, and it will itself hand over matches and fingerprints to $P_{\ell+1}$ with a small delay. One of the reasons for the delays is that processes operate in a round-robin scheme – one process per arriving symbol. The process that is responsible for P_s (which has length $m-4\delta$) returns matches with a delay of up to 3δ arriving symbols. Hence there is a gap of length δ in which we can work out if the whole of P matches. To do this we have another process that runs in parallel with all other processes and explicitly checks if any match with P_s can be extended with the remaining 4δ symbols by directly comparing their predecessor values with the last 4δ predecessor values of the pattern. This job is spread out over δ arriving symbols, hence matches with P are outputted in constant time.

3 The main algorithm

We are now in a position to describe the full algorithm of Theorem 1. Recall that the algorithm will find p-matches with each of the pattern prefixes $P_0, \ldots, P_s$ defined in the previous section. If a shorter prefix fails to match at a given position then there is no need to check matches for longer prefixes. Our algorithm runs three main processes concurrently which we label A, B and C. The term process had a slightly different meaning in the previous section, but hopefully this will cause no confusion. Each process takes $O(1)$ time per arriving symbol. Recall that both the pattern and text alphabets are $\Sigma_\mathrm{P} = \{0, \ldots, |\Sigma_\mathrm{P}| - 1\}$. **Process A** finds p-matches with prefix P_0 which are inserted as they occur into a *match queue* M_0. **Process B** finds p-matches for prefixes $P_1, \ldots, P_s$ which are inserted into the match queues $M_1, \ldots, M_s$, respectively. The p-matches are inserted with a delay of up to 3δ symbol arrivals after they occur. **Process C** finds p-matches with the whole pattern P which are outputted in constant time as they occur as described in Section 2.2.

It is crucial for the space usage that the match queues $M_0, M_1, \ldots, M_s$ will be stored in a compressed fashion. The delay in detecting p-matches with P_ℓ in Process B is a consequence of deamortising the work required to find a prefix match, which we spread out over $\Theta(\delta)$ arriving symbols. We can afford to spread out the work in this way because the p-period of $P_{\ell-1}$ is at least δ so any p-matches are at least this far apart.

Throughout this section we assume that $m > 14\delta$ so that $m_\ell - m_{\ell-1} \geqslant 3\delta$ for $\ell \in \{1, \ldots, s\}$. If $m \leqslant 14\delta$, or the p-period of P is 3δ or less, we use the deterministic algorithm presented in Section 4 to solve the problem within the required bounds.

3.1 Process A (finding matches with P_0)

From the definition of P_0 we have that if we remove the final character (giving the string $P[0,\ m_0-2]$) then its p-period is at most 3δ. The p-period of P_0 itself could be much larger. As part of process A we run the deterministic pattern matching algorithm from Section 4 (see Theorem 4) on $P[0,\ m_0-2]$. It returns p-matches in constant time and uses $O(|\Sigma_{\mathrm{P}}|+3\delta)=O(|\Sigma_{\mathrm{P}}|\log m)$ space.

In order to establish matches with the whole of P_0 we handle the final character separately. If the deterministic subroutine reports a match that ends in $T[i-1]$, when $T[i]$ arrives we have a p-match with P_0 if and only if $\mathrm{pred}(T)[i]=\mathrm{pred}(P_0)[m_0-1]$ (or $\mathrm{pred}(T)[i]\geqslant m_0$ if $\mathrm{pred}(P_0)[m_0-1]=0$). As the alphabet is of the form $\Sigma_{\mathrm{P}}=\{0,\dots|\Sigma_{\mathrm{P}}|-1\}$, we can compute the value of $\mathrm{pred}(T)[i]$ in $O(1)$ time by maintaining an array A of length $|\Sigma_{\mathrm{P}}|$ such that for all $\sigma\in\Sigma_{\mathrm{P}}$, $A[\sigma]$ gives the index of the most recent occurrence of symbol σ.

Whenever Process A finds a match with P_0 at position i' of the text, the pair $(i',\Phi_0(i'))$ is added to a (FIFO) queue M_0, which is queried by Process B when handling prefix P_1.

3.2 Process B (finding matches with $P_1,\dots,P_s$)

We split the discussion of the execution of Process B into s *levels*, $1,\dots,s$. For each level ℓ the fingerprint $\Phi^0_\ell(i')$ is computed for each position i' at which $P_{\ell-1}$ p-matches. Then, as discussed in Section 2, if $\Phi^0_\ell(i')=\phi(\mathrm{pred}(P_\ell)[m_{\ell-1},\ (m_\ell-1)])$, there is also a match with P_ℓ at i'. The algorithm will in this case add the pair $(i',\Phi_\ell(i'))$ to the queue M_ℓ which is subject to queries by level $\ell+1$. To this end we compute $\Phi_\ell(i')\ominus\Phi_{\ell-1}(i')$ and $\Delta_\ell(i')$, where $\Delta_\ell(i')$ contains all the positions which should be zeroed in order to obtain $\Phi^0_\ell(i')$. In the example of Figure 1, $\Delta_\ell(i')=\{1,5,7\}$ (the `d`, `e` and `f`, respectively).

In order for process B to spend only constant time per arriving symbol, all its work must be scheduled carefully. The preparation of the $\Delta_\ell(i')$ values takes place as a subprocess we name B1. Computing $\Phi_\ell(i')\ominus\Phi_{\ell-1}(i')$ and establishing matches takes place in another subprocess named B2. The two subprocesses are run in sequence for each arriving symbol.

Subprocess B1 (prepare zeroing) We use a queue D_ℓ associated with each level l which contains the most recent $O(|\Sigma_{\mathrm{P}}|)$ positions with predecessor the values greater than $m_{\ell-1}$. We will see below that $\Delta_\ell(i')$ is a subset of the positions in D_ℓ (adjusted to the offset i').

Unfortunately, in the worst case, for an arriving symbol $T[i]$, i could belong to all of the D_ℓ queues. Since we can only afford constant time per arriving symbol, we cannot insert i into more than a constant number of queues. The solution is to buffer arriving symbols. When some $T[i]$ arrives we first check whether $\mathrm{pred}(T)[i]>m_0$. If so, the pair $(i,\mathrm{pred}(T)[i])$ is added to a buffer $\mathcal{B}$ to be dealt with later. Together with the pair we also store the value $r^i \bmod p$ which will be needed to perform the required zeroing operations.

In addition to adding a new element to the buffer $\mathcal{B}$, the Subprocess B1 will also process elements from $\mathcal{B}$. If is is currently not in the state of processing an element, it will now start doing so by removing an element from $\mathcal{B}$ (unless $\mathcal{B}$ is empty). Call this element $(j,\mathrm{pred}(T)[j])$. Over the next s arriving symbols the Subprocess B1 will do the following. For each of the s levels ℓ, if $\mathrm{pred}(T)[j]>m_{\ell-1}$, add $(j,\mathrm{pred}(T)[j])$ to the queue D_ℓ. If D_ℓ contains more than $12|\Sigma_{\mathrm{P}}|$ elements, discard the oldest.

Subprocess B2 (establish matches) This subprocess schedules the work across the levels in a round-robin fashion by only considering level $\ell=1+(i \bmod s)$ when the symbol $T[i]$ arrives. Potential matches may not be reported by this subprocess until up to 3δ arriving symbols after they occur. As $P_{\ell-1}$ has p-period at least 3δ, the processing of potential matches does not overlap.

The Subprocess B2 for level ℓ is always in one of two states: either it is *checking* whether a matching position i' for $P_{\ell-1}$ is also a match with P_ℓ, or it is *idle*. If idle, level ℓ looks into queue $M_{\ell-1}$ which holds matches with $P_{\ell-1}$. If $M_{\ell-1}$ is non-empty, level ℓ removes an element from $M_{\ell-1}$, call this element $(i', \Phi_{\ell-1}(i'))$, and enters the checking state. Whenever $i > i' + m_\ell + \delta$, level ℓ will start checking if i' is also a matching position with P_ℓ. It does so by first computing the fingerprint $\Phi_\ell(i') \ominus \Phi_{\ell-1}(i')$, which by definition equals $\big(\Phi_\ell(i') - \Phi_{\ell-1}(i')\big)r^{-i'-m_{\ell-1}} \bmod p$. We can ensure the fingerprint $\Phi_\ell(i')$ is always available when needed by maintaining a circular buffer of the most recent $\Theta(\delta)$ fingerprints of the text. Similarly we can obtain $r^{-i'-m_{\ell-1}} \bmod p$ in $O(1)$ time by keeping a buffer of the most recent $\Theta(\delta)$ values of $r^{-i} \bmod p$ along with $r^{-m_\ell} \bmod p$ for all ℓ.

Over the next at most $|\Sigma_\mathrm{P}|$ arriving symbols for which Subprocess B2 is considering level ℓ (i.e. those with $\ell = 1 + (i \bmod s)$), $\Phi^0_\ell(i')$ will be computed from $\Phi_\ell(i') \ominus \Phi_{\ell-1}(i')$ by stepping through the elements of the queue D_ℓ. For any element $(j, \mathrm{pred}(T)[j]) \in D_\ell$, we have that $(j - i' - m_{\ell-1}) \in \Delta_\ell(i')$ if and only if $\mathrm{pred}(T)[j] > j - i'$. Further, as Subprocess B1 stored $r^j \bmod p$ with the element in D_ℓ and $r^{i'} \bmod p$ is obtained through the circular buffer as above, we can perform the zeroing in $O(1)$ time.

Having computed $\Phi^0_\ell(i')$, we then compare it to $\phi(\mathrm{pred}(P_\ell)[m_{\ell-1},\ (m_\ell - 1)])$. If they are equal, we have a p-match with P_ℓ at position i' of the text, and the pair $(i', \Phi_\ell(i'))$ is added to the queue M_ℓ. This occurs before $T[i' + m_\ell + 3\delta]$ arrives.

3.3 Correctness, time and space analysis

The time and space complexity almost follow immediately from the description of our algorithm, but correctness requires further attention. In particular one has to show that buffers do not overflow, elements in queues are dealt with before being discarded and every possible match will be found (disregarding the probabilistic error in the fingerprint comparisons).

▶ **Lemma 7.** *The algorithm described above proves Theorem 1.*

Proof. Coupled with the discussion in Section 2, the time and space complexity almost follow immediately from the description. It only remains to show that, at any time, $|\mathcal{B}| \leqslant |\Sigma_\mathrm{P}|$. First observe that any symbol $\sigma \in \Sigma_T$ is only inserted into $\mathcal{B}$ when $\mathrm{pred}(T)[i] > m_0 > \delta$ which can only happen at most once in every $\delta = |\Sigma_\mathrm{P}| \log m$ arriving symbols. Further we remove one element every $s \leqslant \lceil \log m \rceil$ arrivals and in particular remove the σ occurrence after at most $|\mathcal{B}|\lceil \log m \rceil$ arrivals. As $\mathcal{B}$ is initially empty, by induction it follows that no symbol occurs more than once in $\mathcal{B}$.

For correctness, it remains to show that we correctly obtain the positions of $\Phi^0_\ell(i')$ from D_ℓ. It follows from the description that all positions of $\Phi^0_\ell(i')$ correspond to elements inserted into D_ℓ at some point. However we need to prove that these elements are present in D_ℓ while $\Phi^0_\ell(i')$ is calculated. Any element inserted into $\mathcal{B}$ during $T[i',\ (i' + m_\ell - 1)]$ has cleared the buffer by the end of interval B (which has length δ) by the argument above. Therefore any relevant element has been inserted into D_ℓ by the start of interval C, during which we calculate $\Phi^0_\ell(i')$. Any element inserted into D_ℓ is at least $m_{\ell-1}$ characters from its predecessor. Therefore, summing over all symbols in the alphabet, there are at most $4|\Sigma_P|$ positions in $T[i',\ (i' + 2m_\ell - 1)]$ which are inserted into D_ℓ. As D_ℓ is a FIFO queue of size $12|\Sigma|$, the relevant elements are still present after interval C. As commented above, potential matches in M_ℓ are separated by more than 3δ arrivals because $P_{\ell-1}$ has p-period more than 3δ. They are processed within 3δ arrivals so M_ℓ does not overflow. ◀

4 The deterministic matching algorithm

We now describe the deterministic algorithm that solves Theorem 4. Its running time is $O(1)$ time per character and it uses $O(|\Sigma_{\mathrm{P}}| + \rho)$ words of space, where ρ is the parameterized period of P. We require that both the pattern and text alphabets are $\Sigma_{\mathrm{P}} = \{0, \ldots, |\Sigma_{\mathrm{P}}| - 1\}$.

We first briefly summarise the overall approach of [1] which our algorithm follows. It resembles the classic KMP algorithm. When $T[i]$ arrives, the overall goal is to calculate the largest r such that $P[0,\ r-1]$ p-matches $T[(i-r+1),\ i]$. A p-match occurs iff $r = m$. When a new text character $T[i+1]$ arrives the algorithm compares $\mathrm{pred}(P)[r]$ to $\mathrm{pred}(T)[i+1]$ in $O(1)$ time to determine whether $P[0,\ r]$ p-matches $T[(i-r+1),\ i+1]$. More precisely, the algorithm checks whether either $\mathrm{pred}(P)[r] = \mathrm{pred}(T)[i+1]$, or $\mathrm{pred}(P)[r] = 0 \wedge \mathrm{pred}(T)[i+1] > r$. The second case covers the possibility that the previous occurrence in the text was outside the window. If there is a match, we set $r \leftarrow r+1$ and $i \leftarrow i+1$ and continue with the next text character. If not, we shift the pattern prefix $P[0,\ r-1]$ along by its p-period, denoted ρ_{r-1}, so that it is aligned with $T[(i-r+\rho_{r-1}+1),\ i]$. This is the next candidate for a p-match. In the original algorithm, the p-periods of all prefixes are stored in an array of length m called a prefix table.

Our main hurdle here is to store both a prefix table suitable for p-matching as well as an encoding of the pattern in only $O(|\Sigma_{\mathrm{P}}| + \rho)$ space, while still allowing efficient access to both. It is well-known that any string P can be stored in space proportional to its exact period. In Lemma 9, which follows from Lemma 8, we show an analogous result for $\mathrm{pred}(P)$.

▶ **Lemma 8.** *For any $j \in [\rho]$ there is a constant k_j such that $\mathrm{pred}(P)[j+k\rho]$ is 0 for $k < k_j$, and c_j for $k \geqslant k_j$, where $c_j > 0$ is a constant that depends on j.*

▶ **Lemma 9.** *The predecessor string $\mathrm{pred}(P)$ can be stored in $O(\rho)$ space, where ρ is the p-period of P. Further, for any $j \in [m]$ we can recover $\mathrm{pred}(P)[j]$ in $O(1)$ time.*

We now explain how to store the parameterized prefix table in only $O(\rho)$ space, in contrast to $\Theta(m)$ space which a standard prefix table would require. The p-period ρ_r of $P[0,\ r]$ is, as a function of r, non-decreasing in r. This property enables us to run-length encode the prefix table and store it as a doubly linked list with at most ρ elements, hence using only $O(\rho)$ space. Each element corresponds to an interval of prefix lengths with the same p-period, and the elements are linked together in increasing order (of the common p-period). This does not allow $O(1)$ time random access to the p-period of any prefix, but for our purposes it will suffice to perform sequential access. To accelerate computation we also store a second linked list of the indices of the first occurrences of each symbol in P in ascending order, i.e. every j such that $\mathrm{pred}(P)[j] = 0$. This uses $O(|\Sigma_{\mathrm{P}}|)$ space.

There is a crucial second advantage to compressing the prefix table which is that it allows us to upper bound the number of prefixes of P we need to inspect when a mismatch occurs. When a mismatch occurs in our algorithm, we repeatedly shift the pattern until a p-match between a text suffix and pattern prefix occurs. Naively it seems that we might have to check many prefixes within the same run. However, as a consequence of Lemma 8 we are assured that if some prefix does not p-match, every prefix in the same run with $\mathrm{pred}(P)[j] \neq 0$ will also mismatch (except possibly the longest). Therefore we can skip inspecting these prefixes. This can be seen by observing (using Lemma 8) that for j such that $\rho_j = \rho_{j+1}$, we have $\mathrm{pred}(P)[j - \rho_j] \in \{0,\ \mathrm{pred}(P)[j]\}$. By keeping pointers into both linked lists, it is straightforward to find the next prefix to check in $O(1)$ time. Whenever we perform a pattern shift we move at least one of the pointers to the left. Therefore the total number of pattern shifts inspected while processing $T[i]$ is at most $O(|\Sigma_{\mathrm{P}}| + \rho)$. As each pointer only moves to the right by at most one when each $T[i]$ arrives, an amortised time complexity of $O(1)$ per character follows. The space usage is $O(|\Sigma_{\mathrm{P}}| + \rho)$, dominated by the linked lists.

We now briefly discuss how to deamortise our solution by applying Galil's KMP deamortisation argument [11]. The main idea is to restrict the algorithm to shift the pattern at most twice when each text character arrives, giving a constant time algorithm. If we have not finished processing $T[i]$ by this point we accept $T[i+1]$ but place it on the end of a buffer, output 'no match' and continue processing $T[i]$. The key property is that the number of text arrivals until the next p-match occurs is at least the length of the buffer. As we shift the pattern up to twice during each arrival we always clear the buffer before (or as) the next p-match occurs. Further, the size of the buffer is always $O(|\Sigma_{\mathrm{P}}| + \rho)$. This follows from the observation above that the number of pattern shifts required to process a single text character is $O(|\Sigma_{\mathrm{P}}| + \rho)$. This concludes the algorithm of Theorem 4.

5 The proof of Lemma 5

In this section we prove the important Lemma 5. Let i_{left} denote an arbitrary position in T where P p-matches. Let X be the set of positions at which P p-matches within $T[i_{\text{left}},\ (i_{\text{left}} + 3m/2 - 1)]$. We now prove that there exist disjoint sets Y and $\mathcal{A}$ with the properties set out in the statement of the lemma.

Let α be the smallest integer such that all distinct symbols in P occur in the prefix $P[0,\ \alpha]$. We first show that ρ, the p-period of P is at least $\alpha/|\Sigma|$. From the minimality of α, we have that $P[\alpha]$ is the leftmost occurrence of some symbol. By the definition of the p-period, we have that $P[0,\ (m-1-\rho)]$ p-matches $P[\rho,\ m-1]$. Under this shift, $P[\alpha]$ (in $P[\rho,\ m-1]$) is aligned with $P[\alpha-\rho]$ (in $P[0,\ (m-1-\rho)]$) . Assume that $P[\alpha-\rho]$ is not a leftmost occurrence and let j be the position of the previous occurrence of $P[j] = P[\alpha-\rho]$. As a p-match occurs, we have that $P[j] = P[j+\alpha] \neq P[\alpha]$, contradiction. By repeating this argument we find distinct symbols at positions $\alpha - k\rho$ for all $k > 0$, implying that $\rho \geqslant \alpha/|\Sigma|$.

We first deal with two simple cases: $\rho > m/8$ or $\alpha \geqslant m/4$ (which implies that $\rho \geqslant m/(4|\Sigma|)$). In these two cases the number of p-matches is easily upper bounded by $6|\Sigma|$, so all positions can be stored in the set Y. We therefore continue under the assumption that $\alpha < m/4$ and $\rho < m/8$. As $\rho \geqslant \alpha/|\Sigma|$, there are at most $(\alpha+1)/(\alpha/|\Sigma|) \leqslant 2|\Sigma|$ positions from the range $[i_{\text{left}},\ i_{\text{left}}+\alpha]$ at which P can p-match T. We can store these positions in the set Y. Next we will show that the positions from the range $[(i_{\text{left}}+\alpha+1),\ (i_{\text{left}}+3m/2-1)]$ at which P p-matches T can be represented by the arithmetic progression $\mathcal{A}$.

First we show that ρ is an *exact period* (not p-period) of $\text{pred}(P)[\alpha+1,\ m-1]$ (but not necessarily the shortest period). Consider arbitrary positions $P[j]$ and $P[j-\rho]$ where $\alpha < j < m-\rho$. By the definition of the p-period, we have that $P[\rho,\ m-1]$ p-matches $P[0,\ (m-1-\rho)]$ and hence that $\text{pred}(P[\rho,\ m-1]) = \text{pred}(P[0,\ (m-1-\rho)])$. In particular, $\text{pred}(P[\rho,\ m-1])[j] = \text{pred}(P[0,\ (m-1-\rho)])[j] = \text{pred}(P)[j]$, where the second equality follows because we take the predecessor string of a prefix of P. Also observe that $\text{pred}(P[\rho,\ m-1])[j]$ either equals 0 or $\text{pred}(P)[j-\rho]$ by definition. Further, $\text{pred}(P[0,\ (m-1-\rho)])[j] = \text{pred}(P)[j] \neq 0$ as $j > \alpha$ and all leftmost occurrences are before α. This implies that $\text{pred}(P[\rho,\ m-1])[j] \neq 0$, hence, $\text{pred}(P)[j-\rho] = \text{pred}(P[\rho,\ m-1])[j] = \text{pred}(P[0,\ (m-1-\rho)])[j] = \text{pred}(P)[j]$.

Recall that P p-matches $T[i_{\text{left}},\ i_{\text{left}}+m-1]$ so $\text{pred}(P) = \text{pred}(T[i_{\text{left}},\ i_{\text{left}}+m-1])]$ and hence ρ is an exact period of $\text{pred}(T[i_{\text{left}},\ i_{\text{left}}+m-1])[\alpha+1,\ m-1]$. Let $j \in \{\alpha+1, \ldots, m-2\}$ and observe that by definition, $\text{pred}(T[i_{\text{left}},\ i_{\text{left}}+m-1])[j] \in \{0, \text{pred}(T)[i_{\text{left}}+j]\}$. However, $\text{pred}(T[i_{\text{left}},\ (i_{\text{left}}+m-1)])[j] = \text{pred}(P)[j] > 0$ because $j > \alpha$ and all leftmost occurrences are in $P[0,\ \alpha]$. This implies that $\text{pred}(T[i_{\text{left}},\ (i_{\text{left}}+m-$

$1)])[j] = \text{pred}(T)[i_{\text{left}} + j]$. As j was arbitrary, we have that $\text{pred}(T)[(i_{\text{left}} + \alpha + 1),\ (i_{\text{left}} + m - 1)] = \text{pred}(T[i_{\text{left}},\ (i_{\text{left}} + m - 1)])[\alpha + 1,\ m - 1]$ and hence ρ is an exact period of $\text{pred}(T)[(i_{\text{left}} + \alpha + 1),\ (i_{\text{left}} + m - 1)]$.

Let i_{right} be the rightmost position in $T[i_{\text{left}},\ i_{\text{left}} + 3m/2 - 1]$ where P p-matches. By the same argument as for i_{left}, we have that ρ is an exact period of $\text{pred}(T)[(i_{\text{right}} + \alpha + 1),\ (i_{\text{right}} + m - 1)]$. Thus, both $\text{pred}(T)[(i_{\text{left}} + \alpha + 1),\ (i_{\text{left}} + m - 1)]$ and $\text{pred}(T)[(i_{\text{right}} + \alpha + 1),\ (i_{\text{right}} + m - 1)]$ has an exact period of ρ. As these two strings overlap by at least ρ characters, we have that ρ is also an exact period of $\text{pred}(T)[i_{\text{left}} + \alpha + 1,\ i_{\text{right}} + m - 1]$.

Let $i \in \{(i_{\text{left}} + \alpha + 1), \ldots, i_{\text{right}} - 1\}$ be arbitrary such that P p-matches $T[i,\ (i + m - 1)]$. We now prove that if $i + \rho < i_{\text{right}}$ then P p-matches $T[i + \rho,\ (i + \rho + m - 1)]$. As p-matches must be at least ρ characters apart this is sufficient to conclude that all remaining matches form an arithmetic progression with common difference ρ. As ρ is an exact period of $\text{pred}(T)[(i_{\text{left}} + \alpha + 1),\ (i_{\text{right}} + m - 1)]$, we have that $\text{pred}(T)[i,\ (i + m - 1)] = \text{pred}(T)[i + \rho,\ (i + \rho + m - 1)]$. By definition, this implies that $\text{pred}(T[i,\ (i + m - 1)]) = \text{pred}(T[i + \rho,\ (i + \rho + m - 1)])$ and hence a p-match also occurs at $i + \rho$.

References

1 Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *IPL*, 49(3):111 – 115, 1994.

2 Arne A. Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *STOC '00*, pages 335–342, 2000.

3 B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *STOC '93*, pages 71–80, 1993.

4 Brenda S. Baker. Parameterized pattern matching by boyer-moore-type algorithms. In *SODA '95*, pages 541–550, 1995.

5 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28 – 42, 1996.

6 Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. on Comp.*, 26(5):1343–1362, 1997.

7 Dany Breslauer and Zvi Galil. Real-time streaming string-matching. In *CPM '11*, pages 162–172, 2011.

8 Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. In *CPM '08*, pages 143–151, 2008.

9 Raphaël Clifford, Markus Jalsenius, Ely Porat, and Benjamin Sach. Space lower bounds for online pattern matching. In *CPM '11*, pages 184–196, 2011.

10 F. Ergun, H. Jowhari, and M. Sağlam. Periodicity in streams. In *RANDOM '10*, pages 545–559, 2010.

11 Zvi Galil. String matching in real time. *Journal of the ACM*, 28(1):134–149, 1981.

12 Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. *ACM Trans. Algorithms*, 3(3), 2007.

13 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res Dev*, 31(2):249 –260, 1987.

14 D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM J. on Comp.*, 6:323–350, 1977.

15 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *FOCS '09*, pages 315–323, 2009.