Algorithmica (2019) 81:1988–2015
https://doi.org/10.1007/s00453-018-0521-7

# Streaming Pattern Matching with $d$ Wildcards

**Shay Golan**[1] · **Tsvi Kopelowitz**[1] · **Ely Porat**[1]

Received: 25 April 2017 / Accepted: 21 September 2018 / Published online: 13 October 2018

**Abstract**
In the pattern matching with $d$ wildcards problem one is given a text $T$ of length $n$ and a pattern $P$ of length $m$ that contains $d$ wildcard characters, each denoted by a special symbol '?'. A wildcard character matches any other character. The goal is to establish for each $m$-length substring of $T$ whether it matches $P$. In the streaming model variant of the pattern matching with $d$ wildcards problem the text $T$ arrives one character at a time and the goal is to report, before the next character arrives, if the last $m$ characters match $P$ while using only $o(m)$ words of space. In this paper we introduce two new algorithms for the $d$ wildcard pattern matching problem in the streaming model. The first is a randomized Monte Carlo algorithm that is parameterized by a constant $0 \leq \delta \leq 1$. This algorithm uses $\tilde{O}(d^{1-\delta})$ amortized time per character and $\tilde{O}(d^{1+\delta})$ words of space. The second algorithm, which is used as a black box in the first algorithm, is a randomized Monte Carlo algorithm which uses $O(d + \log m)$ worst-case time per character and $O(d \log m)$ words of space.

**Keywords** Pattern matching · Streaming algorithms · Fingerprints · String combinatorics

Part of this work took place while the second author was at University of Michigan. This work is supported in part by NSF Grants CCF-1217338, CNS-1318294, and CCF-1514383, 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).

✉ Shay Golan
golansh1@cs.biu.ac.il

Tsvi Kopelowitz
kopelot@gmail.com

Ely Porat
porately@cs.biu.ac.il

[1] Bar Ilan University, Ramat Gan, Israel

## 1 Introduction

We investigate the *pattern matching with d wildcards problem* (PMDW) in the *streaming model*. Let $\Sigma$ be an alphabet and let '?' $\notin \Sigma$ be a special character called the *wildcard character* which matches any character in $\Sigma$. The PMDW problem is defined as follows. Given a *text* string $T = t_0 t_1 \dots t_{n-1}$ over $\Sigma$ and a *pattern* string $P = p_0 p_1 \dots p_{m-1}$ over alphabet $\Sigma \cup \{?\}$ such that $P$ contains exactly $d$ wildcard characters, report all of the occurrences of $P$ in $T$. This definition of a match is one of the most well studied problems in pattern matching [9,19,22,27,29,36]. Applications that use the definition of PMDW are raised in several areas such as Firewall rules and computational biology.

*The streaming model.* The advances in technology over the last decade and the massive amount of data passing through the internet has intrigued and challenged computer scientists, as the old models of computation used before this era are now less relevant or too slow. To this end, new computational models have been suggested to allow computer scientists to tackle these technological advances. One prime example of such a model is the *streaming* model [1,26,30,35]. Pattern matching problems in the streaming model are allowed to preprocess $P$ into a data structure that uses space that is sublinear in $m$ (notice that space usage during the preprocessing phase itself is not restricted). Then, the text $T$ is given online, one character at a time, and the goal is to report, for every integer $\alpha \ge m-1$, whether $t_{\alpha-m+1} \dots t_\alpha$ matches $P$. This reporting must take place before $t_{\alpha+1}$ arrives. Throughout this paper we let $\alpha$ denote the index of the last text character that has arrived.

Following the breakthrough result of Porat and Porat [37], recently there has been a rising interest in solving pattern matching problems in the streaming model [6,7,13,14, 20,28,34]. However, this is the first paper to directly consider the important wildcard variant.

*Related work.* Notice that one way for solving PMDW (not necessarily in the streaming model), is to treat '?' as a regular character, and then run an algorithm that finds all occurrences of $P$ (that does not contain any wildcards) in $T$ with up to $k = d$ mismatches. This is known as the $k$-mismatch problem [2,11,12,14,16,17,25,33,38]. The most-efficient algorithm for the streaming $k$-mismatch problem is due to Clifford, Kociumaka and Porat [16]. The algorithm of [16] implies a solution for PMDW in the streaming model that uses $O(d \operatorname{polylog} m)$ words of space and $O(\sqrt{d} \operatorname{polylog} m)$ time per character. Notice that this algorithm focuses on solving the more general $k$-mismatch problem. In the following corollary we state the implied result for PMDW.

**Corollary 1** *There exists a randomized Monte Carlo algorithm for the PMDW problem in the streaming model that succeeds with probability* $1 - 1/poly(n)$, *uses* $O(d \operatorname{polylog} m)$ *words of space and spends* $O(\sqrt{d} \operatorname{polylog} m)$ *time per arriving text character.*

We mention that while our work is in the streaming model, in the closely related online model (see [15,18]), which is the same as the streaming model without the constraint of using sublinear space, Clifford et al. [10] presented an algorithm, known as the *black box* algorithm, which solves several pattern matching problems. When applied to PMDW, the black box algorithm uses $O(m)$ words of space and $O(\log^2 m)$ time per arriving text character. In the offline model the most efficient algorithms for PMDW take $O(n \log m)$ time and were introduced by Cole and Hariharan [19] and by Clifford and Clifford [9].

One may also consider the problem of allowing wildcards in the text, which is not covered by the definition of the PMDW problem given here. However, the reduction to pattern matching with up to $k$ mismatches works also for the wildcards in the text case, and so the bounds of Corollary 1 also apply to wildcards in the text.

### 1.1 New Results

We introduce two new algorithms for the PMDW problem, as stated in the following theorems (the $\tilde{O}$ notation hides factors logarithmic in $m$). Notice that, for any $\delta > 1/2$ Theorem 3 improves the time per character compared to Corollary 1, while increasing the space usage. The proof of Theorem 3 uses the algorithm of Theorem 2.

**Theorem 2** *There exists a randomized Monte Carlo algorithm for the PMDW problem in the streaming model that succeeds with probability* $1 - 1/poly(n)$, *uses* $O(d \log m)$ *words of space and spends* $O(d + \log m)$ *time per arriving text character.*

**Theorem 3** *For any constant* $0 \le \delta \le 1$ *there exists a randomized Monte Carlo algorithm for the PMDW problem in the streaming model that succeeds with probability* $1 - 1/poly(n)$, *uses* $\tilde{O}(d^{1+\delta})$ *words of space and spends* $\tilde{O}(d^{1-\delta})$ *amortized time per arriving text character.*

### 1.2 Algorithmic Overview

Our algorithms make use of the notion of a *candidate*, which is a location in the last $m$ indices of the current text that is currently considered as a possible occurrence of $P$. As more characters arrive, it becomes clear if this candidate is an actual occurrence or not. In general, an index continues to be a candidate until the algorithm encounters a proof that the candidate is not a valid occurrence (or until the candidate is reported as a match). The algorithm of Theorem 2 works by obtaining such proofs efficiently.

*Overview of algorithm for Theorem* 2. For the streaming pattern matching problem without wildcards, the algorithms of Porat and Porat [37] and Breslauer and Galil [6] have three major components.[1] The first component is a partitioning of the interval $[0, m-1]$ into *pattern intervals* of exponentially increasing lengths. Each pattern interval $[i, j]$ corresponds to a *text interval* $[\alpha - j + 1, \alpha - i + 1]$, where $\alpha$ is the index of the last text character that arrived.[2] Notice that when a new text character arrives,

[1] The algorithms of Porat and Porat [37] and Breslauer and Galil [6] are not presented in this way. However, we find that this new way of presenting our algorithm (and theirs) does a better job of explaining what is going on.

[2] The first pattern interval starts at 0, and so the last text interval ends at location $\alpha + 1$, which is a location of a text character that has yet to arrive. To understand why this convention is appropriate, notice that initially every text location should be considered as a candidate, but in order to save space we only address such candidates a moment before their corresponding character arrives since this is the first time the algorithm can obtain proof that the candidate is not a match.

the text intervals are shifted by one location. The second component maintains all of the candidates in a given text interval. This implementation leverages periodicity properties of strings in order to guarantee that the candidates in a given text interval form an arithmetic progression, and thus can be maintained with constant space. The third component is a fingerprint mechanism for testing if a candidate is still valid. Whenever the border of a text interval passes through a candidate, that candidate is tested.

The main challenge in applying the above framework for patterns with wildcards comes from the lack of a good notion of periodicity which can guarantee that the candidates in a text interval form an arithmetic progression. To tackle this challenge, we design a new method for partitioning the pattern into intervals, which, combined with new fundamental combinatorial properties, leads to an efficient way for maintaining the candidates in small space. In particular, we prove that with our new partitioning there are at most $O(d \log m)$ candidates that are not part of any arithmetic progression for any text interval. Remarkably, the proof bounding the number of such candidates uses a more global perspective of the pattern, as opposed to the techniques used in non-wildcard results.

*Overview of algorithm for Theorem* 3. The algorithm of Theorem 3 uses the algorithm of Theorem 2 (with a minor adaptation) combined with a new combinatorial perspective on *periodicity* that applies to strings with wildcards. The notion of periodicity in strings (without wildcards) and its usefulness are well studied [6,21,23,24,32,37]. However, extending the usefulness of periodicity to strings with wildcards runs into difficulties, since the notions are either too inclusive or too exclusive (see [3–5,8,40]). Thus, we introduce a new definition of periodicity, called the *wildcard-period length* that captures, for a given pattern with wildcards, the smallest possible average distance between occurrences of the pattern in any text. See Definition 7. For a string $S$ with wildcards, we denote the wildcard-period length of $S$ by $\pi_S$.

Let $P^*$ be the longest prefix of $P$ such that $\pi_{P^*} \leq d^\delta$. The algorithm of Theorem 3 has two main components, depending on whether $P^* = P$ or not. In the case where $P^* = P$, the algorithm takes advantage of the wildcard-period length of $P$ being small, which, together with techniques from number theory and new combinatorial properties of strings with wildcards, allows to spend only $\tilde{O}(1)$ time per character and to use only $\tilde{O}(d^{1+\delta})$ words of space. This is summarized in Theorem 21. Of particular interest is Lemma 20 which combines number theory with combinatorial string properties in a new way. We expect these ideas to be useful in other applications.

If $P^* \neq P$, then we use the algorithm of Theorem 21 to locate occurrences of $P^*$, and by maximality of $P^*$, occurrences of prefixes of $P$ that are longer than $P^*$ must appear far apart (on average). These occurrences are given as input to a minor adaptation of the algorithm of Theorem 2 in the form of candidates. Utilizing the large average distance between candidates, we obtain an $\tilde{O}(d^{1-\delta})$ amortized time cost per character.

## 2 Preliminaries

### 2.1 Periods

We assume without loss of generality that the alphabet is $\Sigma = \{1, 2, \dots, n\}$. For a string $S = s_0 s_1 \dots s_{\ell-1}$ over $\Sigma$ and integer $1 \le k \le \ell$ , the substring $s_0 s_1 \dots s_{k-1}$ is called a *prefix* of $S$ and $s_{\ell-k} \dots s_{\ell-1}$ is called a *suffix* of $S$.

A prefix of $S$ of length $i \ge 1$ is a *period* of $S$ if and only if $s_j = s_{j+i}$ for every $0 \le j \le \ell - i - 1$. The shortest period of $S$ is called *the principal period* of $S$, and its length is denoted by $\rho_S$. If $\rho_S \le \frac{|S|}{2}$ we say that $S$ is *periodic*.

The following lemma is due to Breslauer and Galil [6].

**Lemma 4** ([6, Lemma 3.1]) *Let $u$ and $v$ be strings such that $u$ contains at least three locations of occurrences of $v$. Let $t_1 < t_2 < \dots < t_h$ be the locations of all occurrences of $v$ in $u$. Assume that for $i = 1, \dots, h-2$, we have $t_{i+2} - t_i \le |v|$. Then, the sequence $(t_1, t_2, \dots, t_h)$ forms an arithmetic progression with difference $\rho_v$.*

The following lemma follows from Lemma 4.

**Lemma 5** *Let $v$ be a string of length $\ell$ and let $u$ be a string of length at most $2\ell$. If $u$ contains at least three occurrences of $v$ then the distance between any two occurrences of $v$ in $u$ is a multiple of $\rho_v$ and $v$ is a periodic string.*

***Proof*** Let $0 \le c_1 < c_2 < c_3 \le |u| - 1$ be three occurrences of $v$ in $u$. Thus, $c_3 \le (|u| - 1) - (|v| - 1) \le 2\ell - \ell = \ell$, and so $c_3 - c_1 \le \ell$. Therefore, by Lemma 4, all the occurrences of $v$ in $u$ form an arithmetic progression with common difference $\rho_v$. In particular, the distance between any two occurrences of $v$ in $u$ is a multiple of $\rho_v$. Hence, $\rho_v + \rho_v \le (c_3 - c_2) + (c_2 - c_1) = c_3 - c_1 \le \ell = |v|$ and $\rho_v \le \frac{|v|}{2}$. Thus, by definition, $v$ is a periodic string. □

The following lemma will be useful in the paper.

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

***Proof*** Since $v$ is a substring of $u$, we have by definition that $\rho_u$ is a period length of $v$, and thus $\rho_v \le \rho_u$ by the minimality of $\rho_v$. It only remains to prove that $\rho_u \le \rho_v$, which we do by showing that $\rho_v$ is a period length of $u$. We denote $u = u_0 u_1 \dots u_{|u|-1}$. Let $0 \le i < |u| - \rho_v$ be an index in $u$, we have to prove that $u_i = u_{i+\rho_v}$. Let $a$ be an index such that $v$ occurs in $u$ in position $a$, thus $u_a u_{a+1} \dots u_{a+2\rho_u-1}$ is a substring of both $u$ and $v$. Since $\rho_u$ is a period length of $u$, $u_i = u_{i+z\cdot\rho_u}$ for any $z \in \mathbb{Z}$ if $0 \le i + z \cdot \rho_u < |u|$. In particular, for $z = \left\lceil \frac{a-i}{\rho_u} \right\rceil$ we have that $u_i = u_{i+\left\lceil \frac{a-i}{\rho_u} \right\rceil \rho_u}$. Let $b = i + z \cdot \rho_u$. Notice that $a \le b < a + \rho_u$ and $a \le b + \rho_v < a + 2\rho_u$. Therefore, $b$ and $b + \rho_v$ are both indices of characters in $v$, and thus $u_b = u_{b+\rho_v}$. Hence, we have that $u_i = u_{i+z\cdot\rho_u} = u_{i+z\cdot\rho_u+\rho_v} = u_{i+\rho_v}$, where the last equality is based again on the fact that $\rho_u$ is a period length of $u$. □

*Periods and wildcards* For a string $u$ with no wildcards, there is an inverse relationship between the maximum number of occurrences of $u$ in a text of a given length and the principal period length of $u$. Next, we define the *wildcard-period length* of a string over $\Sigma \cup \{?\}$ which captures a similar type of relationship for strings *with* wildcards. The usefulness of this definition for our needs is discussed in more detail in Sect. 6. Let $occ(S', S)$ be the number of occurrences of a string $S$ in a string $S'$.

**Definition 7** For a string $S$ over $\Sigma \cup \{?\}$, the wildcard-period length of $S$ is

$$\pi_S = \min_{S' \in \Sigma^{2|S|-1}} \left\{ \left\lceil \frac{|S|}{occ(S', S)} \right\rceil \right\}.$$

### 2.2 Fingerprints

For the following let $u, v \in \bigcup_{i=0}^{n} \Sigma^i$ be two strings of size at most $n$. Porat and Porat [37] and Breslauer and Galil [6] extended the fingerprint method of Karp and Rabin [31], and 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 $u \neq v$ then $\phi(u) \neq \phi(v)$ 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 $|w| \leq n$ 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 A Generic Algorithm

We start with a generic algorithm (pseudo-code is given in Fig. 1) for solving pattern matching problems in the streaming model. With proper implementations of the algorithm's components, the algorithm solves the PMDW problem. The generic algorithm makes use of the notion of a *candidate*. Initially every text index $c$ is considered as a candidate for a pattern occurrence from the moment $t_{c-1}$ arrives. An index continues

**Fig. 1** Generic algorithm. The purpose of the initialization is to consider location 0 as a candidate before any candidate has arrived

```
INIT()
1  $Q_{I_0}$.Enqueue(0)

PROCESS-CHARACTER($t_\alpha$)
1  for $h = 0$ to $k$
2      $c = Q_{I_h}$.Dequeue()
3      if $c$ exists and $c$ is valid
4          if $h = k$
5              report $c$ as a match
6          else $Q_{I_{h+1}}$.Enqueue($c$)
7  $Q_{I_0}$.Enqueue($\alpha + 1$)
```

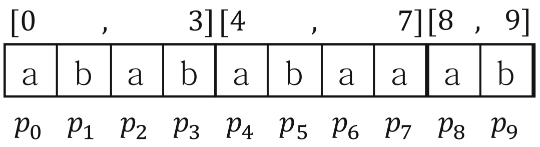


**Fig. 2** Example of a pattern and its arbitrarily chosen pattern intervals. The pattern length is 10 and the pattern intervals are [0, 3], [4, 7] and [8, 9]

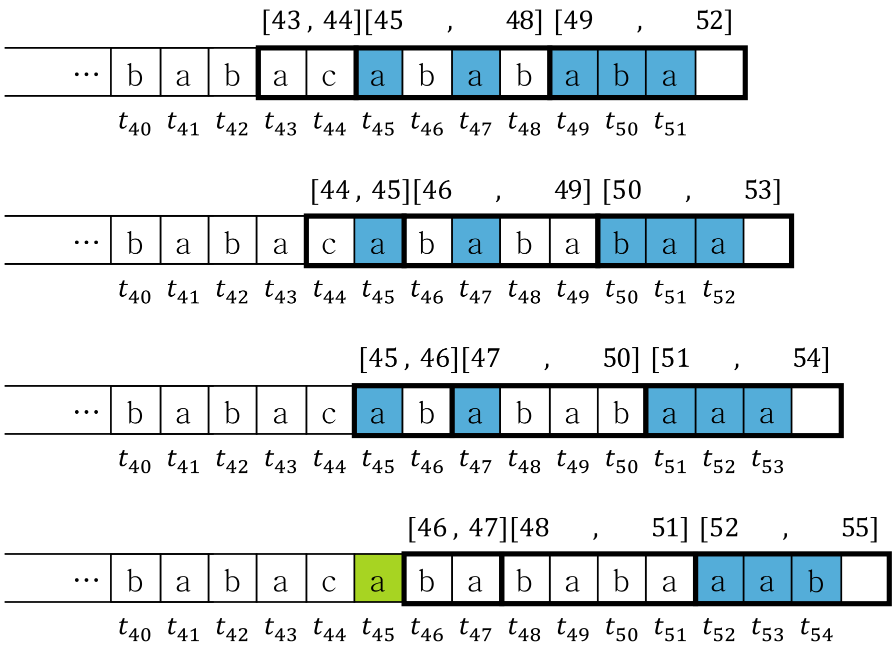

[43, 44] [45 , 48] [49 , 52]
··· b a b a c a b a b a b a
$t_{40}$ $t_{41}$ $t_{42}$ $t_{43}$ $t_{44}$ $t_{45}$ $t_{46}$ $t_{47}$ $t_{48}$ $t_{49}$ $t_{50}$ $t_{51}$
[44, 45] [46 , 49] [50 , 53]
··· b a b a c a b a b a b a a
$t_{40}$ $t_{41}$ $t_{42}$ $t_{43}$ $t_{44}$ $t_{45}$ $t_{46}$ $t_{47}$ $t_{48}$ $t_{49}$ $t_{50}$ $t_{51}$ $t_{52}$
[45, 46] [47 , 50] [51 , 54]
··· b a b a c a b a b a b a a a
$t_{40}$ $t_{41}$ $t_{42}$ $t_{43}$ $t_{44}$ $t_{45}$ $t_{46}$ $t_{47}$ $t_{48}$ $t_{49}$ $t_{50}$ $t_{51}$ $t_{52}$ $t_{53}$
[46, 47] [48 , 51] [52 , 55]
··· b a b a c a b a b a b a a a b
$t_{40}$ $t_{41}$ $t_{42}$ $t_{43}$ $t_{44}$ $t_{45}$ $t_{46}$ $t_{47}$ $t_{48}$ $t_{49}$ $t_{50}$ $t_{51}$ $t_{52}$ $t_{53}$ $t_{54}$

**Fig. 3** Example of an execution of the generic algorithm with the pattern of Fig. 2. In each row a new text character arrives. The bold borders illustrate the text intervals. Each blue cell is a position of a candidate and the green cell corresponds to a match. When $t_{52}$ arrives, the candidate $c_1 = 45$ is tested, since it exits a text interval. The candidate $c_1$ remains alive because *abababaa* is a prefix of the pattern. Notice that at this time the candidate $c_2 = 47$ is not a valid occurrence of the pattern, but, the algorithm does not remove $c_2$ until $c_2$ reaches the end of the text interval. When $t_{54}$ arrives, the candidates $c_1 = 45$ and $c_2 = 47$ are tested, as they have reached the end of their text intervals. At this time, $c_2$ is removed since the text *ababaaab* is not a prefix of the pattern. The candidate $c_1$ remains alive and is reported as a match, since $c_1$ reached the end of the last text interval (Color figure online)

to be a candidate until the algorithm encounters a proof that the candidate is not a valid occurrence (or until it is reported as a match). A candidate is *alive* until such proof is given.

The generic algorithm is composed of three conceptual parts that affect the complexities of the algorithm. An example of an execution of the generic algorithm appears in Figs. 2 and 3:

- *Pattern and text intervals.* The first part is an ordered list $\mathcal{I} = (I_0, \ldots, I_k)$ of intervals. The disjoint union of the intervals of $\mathcal{I}$ is exactly $[0, m-1]$ and the intervals are ordered such that $I = [i, j]$ precedes $I' = [i', j']$ if and only if $j < i'$. Each interval $I \in \mathcal{I}$ is called a *pattern interval*. For each pattern interval $I = [i, j] \in \mathcal{I}$ we define a corresponding *text interval*, $\texttt{text\_interval}(I, \alpha) = [\alpha - j + 1, \alpha - i + 1]$. When character $t_\alpha$ arrives, a text location $c \in \texttt{text\_interval}(I, \alpha)$ is a candidate if and only if $t_c \cdots t_{c+i-1}$ matches $p_0 \cdots p_{i-1}$. The *candidate set* $\mathcal{C}(I, \alpha)$ is the set of text positions in $\texttt{text\_interval}(I, \alpha)$ which are candidates right after the arrival of $t_\alpha$.

- *Candidate queues.* The second conceptual part of the generic algorithm is an implementation of a *candidate-queue* data structure. For any interval $I \in \mathcal{I}$, the algorithm maintains a candidate queue $Q_I$. At any time $\alpha$, which is the time right after $t_\alpha$ arrives, but before $t_{\alpha+1}$ arrives, $Q_I$ stores a (possibly implicit) representation of $\mathcal{C}(I, \alpha)$. Thus, the operations of the data structure are *time-dependent*. Candidate-queues support the following operations.

  **Definition 8** A candidate-queue for an interval $[i, j] = I \in \mathcal{I}$ supports the following operations at time, where $t_\alpha$ is the last text character that arrived.

  1. $\texttt{Enqueue}()$: add $c = \alpha - i + 1$ to the candidate-queue.
  2. $\texttt{Dequeue}()$: remove and return a candidate $c = \alpha - j$, if such a candidate exists.

  Since there is a bijection between pattern intervals and text intervals we say that a candidate-queue that is associated with pattern interval $I$ is also associated with the corresponding text interval $\texttt{text\_interval}(I, \alpha)$.

- *Assassinating candidates.* The third conceptual part addresses the following. When a new text character arrives, all the text intervals move one position ahead, and some candidates leave some text intervals and their corresponding candidate sets. The third conceptual part is a mechanism for testing if a candidate is valid after that candidate leaves a candidate set. This mechanism is used in order to determine if the candidate should enter the candidate-queue of the next text interval, or be reported as a match if there are no more text intervals.

The implementation of each of the three components controls the complexities of the algorithm. Minimizing the number of intervals reduces the number of candidates leaving text intervals at a given time. Efficient implementations of the candidate-queue operations and testing if a candidate is valid control both the space usage and the amount of time spent on each candidate that leaves an interval. Notice that the implementations of these components may depend on each other, which is also the case in our solution.

*A naïve implementation.* The following naïve implementation of the generic algorithm is helpful for gaining intuition as to how the algorithm works. Let $\mathcal{I}_{naïve} = ([0, 0], [1, 1], \ldots, [m-1, m-1])$. The implementation of candidate queue $Q_I$ explicitly stores the set $\mathcal{C}(I, \alpha)$ at time $\alpha$. Notice that $\mathcal{C}(I, \alpha)$ contains at most one candidate. The task of verifying that a candidate $c$ is valid in between text intervals is a straightforward comparison of $p_{\alpha-c}$ with $t_\alpha$. Each such comparison costs $O(1)$ time. The runtime of the algorithm is $\Theta(m)$ per character in the worst-case, and the space usage is also $\Theta(m)$.[3] We refer to this algorithm as the *naïve algorithm*.

[3] For example, if the pattern is $aa \ldots a = a^m$ and the text is $a^n$, then each candidate $c$ is alive as long as the characters $t_c, \ldots t_{c+m-1}$ arrive. Therefore, after the arrival of the first $m - 1$ characters, any additional arriving character is compared with $m$ pattern characters.

*Using fingerprints* If there are no wildcards in $P$, then one can use the following fingerprint based algorithm that verifies the validity of a candidate $c$ only once all the characters $t_c, t_{c+1}, \dots, t_{c+m-1}$ have arrived. This algorithm is closely related to the Karp and Rabin [31] algorithm. The algorithm uses a partitioning of $[0, m-1]$ into only one interval containing all of $[0, m-1]$.

The algorithm maintains the *text fingerprint* which is the fingerprint of the text from its beginning up to the last arriving character. For each text index $c$, just before $t_c$ arrives the algorithm creates a candidate for the index $c$ and stores the text fingerprint $\phi(t_0 t_1 \dots t_{c-1})$ as satellite information of the candidate $c$. Then, $c$ (together with its satellite information) is added to the candidate-queue via the `Enqueue()` operation. When the character $t_{c+m-1}$ arrives, the text fingerprint is $\phi(t_0 \dots t_{c+m-1})$. At this time, the algorithm uses the `Dequeue()` operation to extract $c$ together with $\phi(t_0 t_1 \dots t_{c-1})$ from the candidate-queue. Then, the algorithm tests if $c$ is valid by computing $\phi(t_c \dots t_{c+m-1})$ from the current text fingerprint $\phi(t_0 t_1 \dots t_{c+m-1})$ and the fingerprint $\phi(t_0 t_1 \dots t_{c-1})$ (using the sliding property of the fingerprint function), and then testing if $\phi(t_c \dots t_{c+m-1})$ equals $\phi(p_0 \dots p_{m-1})$. The fingerprint algorithm spends only constant time per text character, but, like the naïve algorithm, uses $\Theta(m)$ words of space to store the candidate-queue.

## 3.1 Fingerprints with Wildcards

Using fingerprints together with wildcards seems to be a difficult task, since for any string $S$ with $x$ wildcards there are $|\Sigma|^x$ different strings over $\Sigma$ that match the string $S$. Each one of these different strings may have a different fingerprint and therefore there are $\Theta(|\Sigma|^x)$ fingerprints to store, which is not feasible. In order to still use fingerprints for solving PMDW we use a special partitioning of $[0, m-1]$, which is described in Sect. 4. The partitioning in Sect. 4 is based on the following preliminary partitioning.

*The preliminary partitioning.* We use a representation of $P$ as $P = P_0?P_1?\dots?P_d$ where each subpattern $P_i$ contains only characters from $\Sigma$ (and may also be an empty string). Let $W = (w_1, w_2, \dots, w_d)$ be the indices of wildcards in $P$ such that for all $1 \le i < d$ we have $w_i < w_{i+1}$. The interval $[0, m-1]$ is partitioned into pattern intervals as follows:

$$\mathcal{J} = ([0, w_1 - 1], [w_1, w_1], [w_1 + 1, w_2 - 1], \dots, [w_d, w_d], [w_d + 1, m - 1]).$$

Since some of the pattern intervals in this partitioning could be empty, we discard such intervals. The pattern intervals of the form $[w_i, w_i]$ are called *wildcard interval*s and the other pattern intervals are called *regular intervals*. Notice that for a text index $c$, the substring $t_c \dots t_{c+m-1}$ matches $P$ if and only if for each regular interval $[i, j]$, $t_{c+i} \dots t_{c+j} = p_i \dots p_j$.

*A preliminary algorithm.* Given the preliminary partition $\mathcal{J}$, one could use the following algorithm for testing the validity of a candidate $c$ whenever it leaves a text interval. During the initialization of the algorithm we precompute and store the fingerprints for all of the subpatterns corresponding to regular intervals. Each time a candidate $c$ is added to a candidate-queue for interval $[i, j] \in \mathcal{J}$ via the `Enqueue()` operation, the algorithm stores the current text fingerprint $\phi(t_0 \ldots t_{c+i-1})$ together with the candidate $c$. When the character $t_{c+j}$ arrives, the text fingerprint is $\phi(t_0 \ldots t_{c+j})$. At this time, the algorithm uses the `Dequeue()` operation to extract $c$ together with $\phi(t_0 t_1 \ldots t_{c+i-1})$ from the candidate-queue of interval $[i, j]$. If $[i, j]$ is a regular interval, then the algorithm tests if $c$ is valid, and removes (assassinates) $c$ if it is not. This validity test is executed by applying the sliding property of the fingerprint function to compute $\phi(t_{c+i} \ldots t_{c+j})$ from the current text fingerprint $\phi(t_0 t_1 \ldots t_{c+j})$ and the fingerprint $\phi(t_0 t_1 \ldots t_{c+i-1})$, and then testing if $\phi(t_{c+i} \ldots t_{c+j})$ is the same as $\phi(p_i \ldots p_j)$. If $[i, j]$ is a wildcard interval then $c$ stays alive without any testing.

A naïve implementation of the candidate queues provides an algorithm that costs $O(d)$ time per character, but uses $\Theta(m)$ words of space. To overcome this space usage we employ a more complicated partitioning, which, together with a modification of the requirements from the candidate-queues, allows us to design a data structure that uses much less space. However, this space efficiency comes at the expense of a slight increase in the time per character.

## 4 The Partitioning

The key idea of the new partitioning is to use the partitioning of Sect. 3.1 as a preliminary partitioning, and then perform a secondary partitioning of the regular pattern intervals, thereby creating even more regular intervals. As mentioned, the intervals are partitioned in a special way which allows us to implement candidate-queues in a compact manner (see Sect. 5).

The following definition is useful in the next lemma.

**Definition 9** For an ordered set of intervals $\mathcal{I} = (I_0, I_1, \ldots I_k)$ and for any integer $0 \le x \le k$, let $\mu_{\mathcal{I}}(I_x) = \max_{0 \le y \le x} \{|I_y|\}$ be the length of the longest interval in the sequence $I_0, \ldots, I_x$. When $\mathcal{I}$ is clear from context we simply write $\mu(I_x) = \mu_{\mathcal{I}}(I_x)$.

The following lemma provides a partitioning which is used to improve the preliminary partitioning algorithm. The properties of the partitioning that are described in the statement of the lemma are essential for our new algorithm. The most essential property is Property 3, since it guarantees that for each pattern interval $I = [i, j]$, there exists a substring of $P$ prior to $p_i$ and with no wildcards whose length is $|I|$. If this substring is not periodic, then for any $\alpha$, $\mathcal{C}(I, \alpha)$ does not contain more than two candidates. If this substring is periodic, then we show how to utilize the periodicity of the string in order to efficiently maintain all the candidates in $\mathcal{C}(I, \alpha)$ for any $\alpha$ (see Sect. 5). In the proof of the lemma we introduce a specific partitioning which has all of the stated properties.

**Lemma 10** *Given a pattern $P$ of length $m$ with $d$ wildcards, there exists a partitioning of the interval $[0, m-1]$ into subintervals $\mathcal{I} = (I_0, I_1 \ldots, I_k)$ which has the following properties:*

1. *If $I = [i, j] \in \mathcal{I}$ is a pattern interval then $p_i \ldots p_j$ either corresponds to exactly one wildcard from $P$ (and so $j = i$) or it is a substring that does not contain any wildcards.*

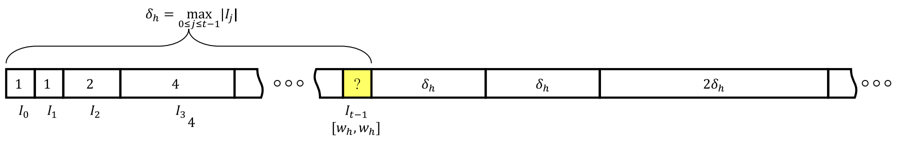


**Fig. 4** The general case: for each $J_h \in \mathcal{J}$ we first create two intervals of length $\delta_h$ and then we iteratively create pattern intervals where the length of each pattern interval is double the length of the previous pattern interval

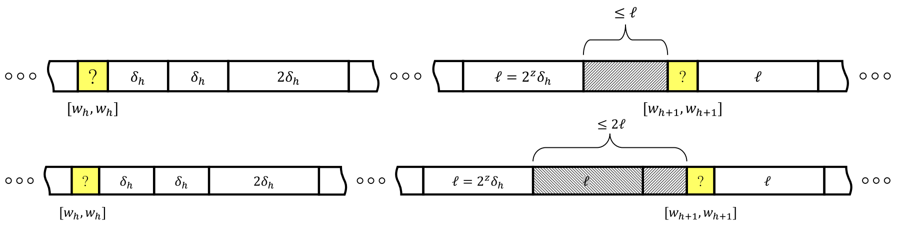
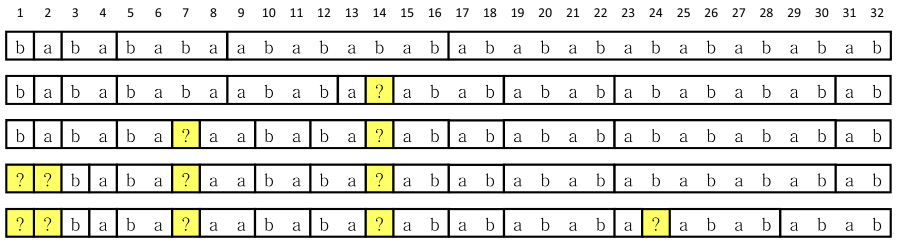


**Fig. 5** Once there is no more room left in $J_h$, if the remaining interval is of length at most $\ell$ (the top case), then we create one pattern interval for all the remaining interval. Otherwise (the bottom case) we create two pattern intervals, the first pattern interval of length $\ell$ and the second pattern interval using the remaining part of $J_h$



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
b a b a b a b a a b a b a b a b a b a b a b a b a b a b a b a b
b a b a b a b a a b a b a ? a b a b a b a b a b a b a b a b a b
b a b a b a ? a a b a b a ? a b a b a b a b a b a b a b a b a b
? ? b a b a ? a a b a b a ? a b a b a b a b a b a b a b a b a b
? ? b a b a ? a a b a b a ? a b a b a b a b a ? a b a b a b a b

**Fig. 6** Example of patterns and their intervals in the secondary partitioning. Each bold rectangle corresponds to an interval in the partition

2. $k = O(d + \log m)$.
3. *For each regular pattern interval $I = [i, j] \in \mathcal{I}$ with $\mu_{\mathcal{I}}(I) > 1$, the length i prefix of P contains a consecutive sequence of $\mu_{\mathcal{I}}(I) \ge |I|$ non-wildcard characters.*
4. $|\{\mu_{\mathcal{I}}(I_0), \mu_{\mathcal{I}}(I_1), \ldots, \mu_{\mathcal{I}}(I_k)\}| = O(\log m)$.

***Proof*** We introduce a secondary partitioning of the preliminary partitioning described in Sect. 3.1, and prove that the secondary partitioning has all the required properties; see Figs. 4, 5 and 6. Recall that we use a representation of $P$ as $P = P_0?P_1?\ldots?P_d$. Let $J_h$ be the preliminary pattern interval corresponding to $P_h$. The secondary partitioning is executed on the pattern intervals $\mathcal{J} = (J_0, J_1, \ldots, J_d)$, where the partitioning of $J_h$ is dependent on the partitioning of $J_0, \ldots, J_{h-1}$. Thus, for $h > 0$, the secondary partitioning of $J_h$ takes place only after the secondary partitioning of $J_{h-1}$.

When partitioning pattern interval $J_h = [i, j]$, let $\delta_h$ be the length of the longest pattern interval in the secondary partitioning of $[0, i-1]$. For the first pattern interval let $\delta_0 = 1$. If $j \le i + \delta_h - 1$ then the only pattern interval is all of $J_h$. If $j \le i + 2 \cdot \delta_h - 1$

then we create the pattern intervals $[i, i+\delta_h-1]$ and $[i+\delta_h, j]$. Otherwise, we first create the pattern intervals $[i, i+\delta_h-1]$ and $[i+\delta_h, i+2\cdot\delta_h-1]$,[4] and for as long as there is enough room in the remaining preliminary pattern interval $J_h$ (between the position right after the end of the last secondary pattern interval that was just created and $j$) we iteratively create pattern intervals where the length of each pattern interval is double the length of the previous pattern interval. Once there is no more room left in $J_h$, let $\ell$ be the length of the last pattern interval we created. If the remaining part of $J_h$ is of length at most $\ell$, then we create one pattern interval for all the remaining part of $J_h$. Otherwise, if the remaining part of $J_h$ is $[s, j]$ we create two pattern intervals, the first pattern interval is $[s, s+\ell-1]$ (of length $\ell$) and the second pattern interval using the remaining part of $J_h$, $[s+\ell, j]$; see Fig. 5.

The secondary partitioning implies all of the desired properties:

**Property 1** Since the secondary partitioning is a subpartitioning of the preliminary partitioning and the preliminary partitioning already had this property, then the secondary partitioning has this property as well.

**Property 2** For a subpattern $P_h$, the length of every pattern interval created from $J_h$ during the secondary partitioning, except for the first two pattern intervals and possibly also the last two pattern intervals, is at least twice the length of the longest pattern interval preceding it. So the total number of such pattern intervals is $O(\log m)$. The number of other regular pattern intervals is at most $4(d+1)$. Additionally, there are $d$ wildcard pattern intervals. So the total number of pattern intervals is at most $4(d+1)+d+O(\log m)=O(d+\log m)$.

**Property 3** If there is a regular pattern interval $I'=[i', j']$ such that $j'<i$ and $|I'|=\mu_{\mathcal{I}}(I)$, then the subpattern associated with $I'$ meets the requirement.

If there is no such pattern interval, then by the definition of $\mu_{\mathcal{I}}(I)$ it must be that $\mu_{\mathcal{I}}(I)=|I|$. Moreover, it must be the case that the length of $I$ is twice the length of the pattern interval preceding $I$, and $I$ is contained in a preliminary pattern interval $J_h$ for some $h$. Let the length of the first pattern interval created in $J_h$ be denoted by $\delta_h$. Let $I_{h,1}, I_{h,2}, \ldots I_{h,r}$ be the first $r$ pattern intervals created in $J_h$ such that $I_{h,r}=I$. The length of any pattern interval $I_{h,r'}$ for $1<r'\le r$ is $2^{r'-2}\delta_h$ (since $|I_{h,1}|=|I_{h,2}|=\delta_h$, and for $2<r'\le r$ we have $|I_{h,r'}|=2|I_{h,r-1}|$), and in particular the length of $I$ is $2^{r-2}\delta_h$. Recall that $I=[i, j]$. The length of the prefix of $P_h$ up to the index $i$ is the sum of the lengths of all the pattern intervals $I_{h,r'}$ for $r'<r$. These lengths sum up to $(1+\sum_{r'=2}^{r-1}2^{r'-2})\delta_h=2^{r-2}\delta_h=|I|$. So the prefix of $P_h$ fulfills the requirement.

**Property 4** We prove a stronger claim: for each $0\le h\le k$, $\mu_{\mathcal{I}}(I_h)$ is a power of 2.

This is true by induction. The first pattern interval is of length 1, and therefore $\mu_{\mathcal{I}}(I_0)=1=2^0$. If $|I_h|\le\mu_{\mathcal{I}}(I_{h-1})$ then $\mu_{\mathcal{I}}(I_h)=\mu_{\mathcal{I}}(I_{h-1})$ which is a power of 2 by the induction hypothesis. Otherwise, if $|I_h|>\mu_{\mathcal{I}}(I_{h-1})$ then by the secondary partitioning algorithm $|I_h|=2\mu_{\mathcal{I}}(I_{h-1})$, and $\mu_{\mathcal{I}}(I_h)=2\mu_{\mathcal{I}}(I_{h-1})$. Hence $\mu_{\mathcal{I}}(I_h)$ is also a power of 2.

The largest pattern interval is at most of length $m$, and therefore there are at most $\lceil\log m\rceil$ different values in $\{\mu_{\mathcal{I}}(I_0), \mu_{\mathcal{I}}(I_1), \ldots, \mu_{\mathcal{I}}(I_k)\}$. □

[4] The choice of having the first two intervals to be of the same length $\delta_h$ is in order to guarantee the third property in the lemma, as shown below.

## 5 The Candidate-Fingerprint-Queue

The algorithm of Theorem 2 is obtained via an implementation of the candidate-queues that uses $O(d \log m)$ words of space, at the expense of having $O(d + \log m)$ intervals in the partitioning. Such space usage implies that we do not store all candidates explicitly. This is obtained by utilizing properties of periodicity in strings. Since candidates are not stored explicitly, we cannot store explicit information per candidate, and in particular we cannot explicitly store fingerprints. On the other hand, we are still interested in using fingerprints in order to perform assassinations.

To tackle this, we strengthen our requirements from the candidate-queue data structure to return not just the candidate but also the fingerprint information that is needed to perform the test of whether the candidate is still valid. For our purposes, this data structure cannot explicitly maintain all the fingerprints information. Thus, we extend the definition of a candidate-queue to a *candidate-fingerprint-queue* as follows.

**Definition 11** A candidate-fingerprint-queue $Q_I$ for an interval $[i, j] = I \in \mathcal{I}$ supports the following operations, where $t_\alpha$ is the last text character that arrived.

1. `Enqueue`$(\phi(t_0 \dots t_{\alpha-i}))$: add $c = \alpha - i + 1$ to the candidate-queue.
2. `Dequeue`(): remove and return a candidate $c = \alpha - j$, if such a candidate exists, together with $\phi(t_0 \dots t_{c-1})$ and $\phi(t_0 \dots t_{c+i-1})$.

In order to reduce clutter of presentation, in the rest of this section we refer to the candidate-fingerprint-queue simply as the *queue*.

### 5.1 Implementation

Our implementation of the queue assumes that we use a partitioning $\mathcal{I}$ which has the properties stated in Lemma 10. Let $I = [i, j]$ be a pattern interval in $\mathcal{I}$ and let $c$ be a candidate from $\mathcal{C}(I, \alpha)$. The *entrance prefix* of $c$ is the substring $t_c \dots t_{c+i-1}$, and the *entrance fingerprint* is $\phi(t_c \dots t_{c+i-1})$. By definition, since $c \in \mathcal{C}(I, \alpha)$, the entrance prefix of $c$ matches $p_0 \dots p_{i-1}$ (which may contain wildcards). Recall that a candidate $c$ is inserted into $Q_I$ together with $\phi(t_0 \dots t_{c-1})$, which we call the *candidate fingerprint* of $c$.

*Satellite information.* The implementation associates each candidate $c$ with *satellite information* (SI), which includes the candidate fingerprint and the entrance fingerprint of the candidate. The SI of a candidate combined with the sliding property of fingerprints are crucial for the implementation of the queue. When $c$ is added to $Q_I$, for some $I = [i, j]$, we compute the entrance fingerprint of $c$ from the candidate fingerprint and from $\phi(t_0 \dots t_{c+i-1})$ which is the text fingerprint at that time. When $c$ is removed from $Q_I$, we compute $\phi(t_0 \dots t_{c+i-1})$ in constant time from the SI of $c$. See Fig. 7.

*Arithmetic progressions and entrance prefixes.* Let $I = [i, j] \in \mathcal{I}$, recall that each candidate $c$ in $Q_I$ has an entrance prefix that matches the prefix of $P$ of length $i$. Assume that this prefix of $P$ does not contain any wildcards. In such a case, the entrance prefix of $c$ must be exactly the prefix of $P$. In particular, the entrance prefixes of all the candidates in $\mathcal{C}(I, \alpha)$ must be the same, and if there are more than three candidates,

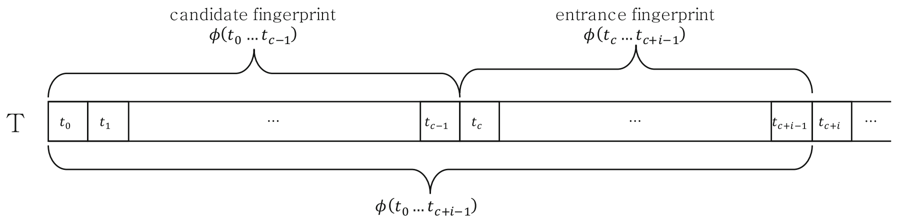


**Fig. 7** The satellite information of a candidate $c$ in a text interval $\texttt{text\_interval}(I, \alpha)$ for $I = [i, j]$ includes the candidate fingerprint $\phi(t_0 \ldots t_c - 1)$ and the entrance fingerprint $\phi(t_c \ldots t_{c+i-1})$. The fingerprint of $\phi(t_0 \ldots t_{c+i-1})$ can be computed in constant time, using the sliding fingerprint property

then one can use the techniques that appear in [6,37] and store all the candidates as an arithmetic progression, using only $O(1)$ words of space. However, since the prefix of $P$ of length $i$ might contain wildcards it is possible for two candidates in $\mathcal{C}(I, \alpha)$ to have different entrance prefixes. In particular all of the mismatches between the entrance prefixes of the two candidates must be at positions corresponding to wildcards in the length $i$ prefix of $P$.

In order to implement the queue using a small amount of space, we distinguish between two types of candidates. This distinction is based on the following definition.

**Definition 12** Suppose $\mathcal{I}$ is a partitioning that satisfies the properties of Lemma 10. For a pattern interval $I = [i, j] \in \mathcal{I}$ let $U_I$ be the set of strings (with no wildcards) that match the prefix of $P$ of length $i$ whose principal period length is at most $\frac{\mu_{\mathcal{I}}(I)}{2}$. If $|U_I| = 1$ then the only string $u \in S_I$ is denoted by $u_I$.

In Lemma 13 we prove that $|U_I| \leq 1$. Corollary 15 (which is a consequence of Lemma 14) states that if at some time $\alpha$ there exist at least three candidates in $\mathcal{C}(I, \alpha)$ with the same entrance prefix $u$, then it must be that $|U_I| = 1$ and $u = u_I$. Thus, the first type of candidates in $\mathcal{C}(I, \alpha)$ are those that have $u_I$ as their entrance prefix, and in Lemma 16 we prove that all such candidates must form an arithmetic progression. Thus, all the first type candidates and their SI can be stored implicitly using $O(1)$ words of space (see Lemma 17). The second type of candidates are the rest of the candidates, and these candidates are stored explicitly together with their SI. We prove in Lemma 18 that the total number of such candidates is $O(d \log m)$, thereby obtaining our claimed space usage.

**Lemma 13** *Suppose $\mathcal{I}$ is a partitioning that satisfies the properties of Lemma 10. Let $I = [i, j] \in \mathcal{I}$ be a pattern interval. Then $|U_I| \leq 1$.*

***Proof*** Assume by contradication that $|U_I| \geq 2$, and let $u_1 \neq u_2$ be two different strings in $U_I$.

By Property 3 of Lemma 10 there exists a string $v$ of length $\mu_{\mathcal{I}}(I) \geq |I| = j - i + 1$ containing only non-wildcard characters that is a substring of the length $i$ prefix of $P$. Since it is possible that $v$ has several occurrences in $p_0 \ldots p_{i-1}$, let $r$ be a location of an arbitrary occurrence of $v$. Since $u_1$ (and $u_2$) matches $p_0 \ldots p_{i-1}$, then $v$ is a substring of $u_1$ (and $u_2$) at location $r$. Thus, $\rho_v \leq \rho_{u_1} \leq \frac{\mu_{\mathcal{I}}(I)}{2} = \frac{|v|}{2}$, and so $v$ must be periodic and $|v| \geq 2\rho_v$. By definition of $U_I$, we have $\rho_{u_1} \leq \frac{\mu_{\mathcal{I}}(I)}{2} = \frac{|v|}{2}$ and $\rho_{u_2} \leq \frac{\mu_{\mathcal{I}}(I)}{2} = \frac{|v|}{2}$. Therefore, by Lemma 6, $\rho_{u_1} = \rho_v = \rho_{u_2}$.

Let $k$ be an index of a mismatch between $u_1$ and $u_2$. In particular, let the $k$'th character of $u_1$ be $x_1$, and the $k$'th character of $u_2$ be $x_2 \neq x_1$. Let $\gamma$ be an integer (possibly negative) such that the $k + \gamma \cdot \rho_v$ location in $u_1$ is within the occurrence of $v$ in $u_1$ (and so also within the occurrence of $v$ in $u_2$). Notice that such $\gamma$ must exist since $|v| \geq 2\rho_v$ (for example, setting $\gamma = \lceil \frac{r-k}{\rho_v} \rceil$). Since $\rho_{u_1} = \rho_v = \rho_{u_2}$, the character at location $k + \gamma \cdot \rho_v$ in $u$ must be $x_1$, while the character at location $k + \gamma \cdot \rho_v$ in $u_2$ must be $x_2$. But $u_1$ and $u_2$ match at all of the locations corresponding to $v$. Thus we have obtained a contradiction. □

**Lemma 14** *Suppose $\mathcal{I}$ is a partitioning that satisfies the properties of Lemma 10. For a pattern interval $I = [i, j] \in \mathcal{I}$:*

1. *If $|U_I| = 0$ then for any text $T$ and time $\alpha \geq 0$, there can never be three candidates in $\mathcal{C}(I, \alpha)$ with the same entrance prefix.*
2. *If $|U_I| = 1$ then for any text $T$ and time $\alpha \geq 0$, if there exist three candidates in $\mathcal{C}(I, \alpha)$ that all share the same entrance prefix, then this entrance prefix must be $u_I$.*

***Proof*** **Case 1** Assume by contradiction that there exists a text $T$ such that at time $\alpha$ there are three candidates $c_1, c_2, c_3 \in \mathcal{C}(I, \alpha)$ that share the same entrance prefix $u$, but $|U_I| = 0$. Assume without loss of generality that $c_1 < c_2 < c_3$. Since $c_1$, $c_2$, and $c_3$ are all occurrences of $u$, then $c_3 - c_2$ and $c_2 - c_1$ are period lengths of $u$. Thus, $\rho_u \leq \min\{c_2 - c_1, c_3 - c_2\} \leq \frac{c_3 - c_1}{2} \leq \frac{(\alpha - i + 1) - (\alpha - j + 1)}{2} \leq \frac{j-i}{2} < \frac{|I|}{2} \leq \frac{\mu_{\mathcal{I}}(I)}{2}$. Hence, $u \in U_I$ by definition, contradicting the assumption.
**Case 2** Suppose that there exists a text $T$ such that at time $\alpha$ there are three candidates $c_1, c_2, c_3 \in \mathcal{C}(I, \alpha)$ that share the same entrance prefix $u$. As in case 1, $\rho_u < \frac{\mu_{\mathcal{I}}(I)}{2}$. Hence, $u \in U_I$ and therefore, by Lemma 13, $u = u_I$ as required. □

**Corollary 15** *Suppose $\mathcal{I}$ is a partitioning that satisfies the properties of Lemma 10 and let $I = [i, j] \in \mathcal{I}$ be a pattern interval. For a text $T$ and time $\alpha$, if there exist three candidates in $\mathcal{C}(I, \alpha)$ with the same entrance prefix $u$ then $|U_I| = 1$ and $u = u_I$.*

**Lemma 16** *Suppose $\mathcal{I}$ is a partitioning that satisfies the properties by Lemma 10. For a pattern interval $I = [i, j] \in \mathcal{I}$ and time $\alpha \geq 0$ if there are $h \geq 3$ candidates $c_1 < c_2 < \cdots < c_h$ in $\mathcal{C}(I, \alpha)$ that have $u_I$ as their entrance prefix, then the sequence $c_1, c_2, \ldots, c_h$ forms an arithmetic progression whose difference is $\rho_{u_I}$.*

***Proof*** The distance between any two candidates in $\mathcal{C}(I, \alpha)$ is at most $|I|$, and $|I| \leq i$ by Property 3 of Lemma 10. Hence, by Lemma 4, all of the occurrences of $u_I$ in $T$ that begin in `text_interval`$(I, \alpha)$ form an arithmetic progression with difference $\rho_{u_I}$. Each of these occurrences matches the $i$ length prefix of $P$, and therefore is a candidate in $\mathcal{C}(I, \alpha)$. Hence, all the candidates of $\mathcal{C}(I, \alpha)$ with $u_I$ as their entrance prefix form an arithmetic progression with difference of $\rho_{u_I}$. □

*Implementation details.* For any pattern interval $I = [i, j]$ and time $\alpha$ we split the set of candidates $\mathcal{C}(I, \alpha)$ into two disjoint sets. The set $\mathcal{C}_{ap}(I, \alpha) = \{c \in$

$\mathcal{C}(I, \alpha) \mid t_c \ldots t_{c+i-1} = u_I\}$ contains all the candidates whose entrance prefix is $u_I$ (*ap* represents *arithmetic progression*; see proof of Lemma 17), and the set $\overline{\mathcal{C}_{ap}}(I, \alpha) = \mathcal{C}(I, \alpha) \setminus \mathcal{C}_{ap}(I, \alpha)$ contains all the other candidates of $\mathcal{C}(I, \alpha)$. Since the membership of $c$ in either $\mathcal{C}_{ap}(I, \alpha)$ or $\overline{\mathcal{C}_{ap}}(I, \alpha)$ depends only on the entrance prefix of $c$, the algorithm determines the membership by using the entrance fingerprint of $c$. We use a linked list $\mathcal{L}_{Q_I}$ to store all of the candidates of $\overline{\mathcal{C}_{ap}}(I, \alpha)$ together with their SI. Adding and removing a candidate that belongs in $\mathcal{L}_{Q_I}$ together with its SI is straightforward. The candidates of $\mathcal{C}_{ap}(I, \alpha)$ are maintained using a separate data structure that leverages Corollary 15 and Lemma 16. Thus, during a `Dequeue()` operation, the queue verifies if the candidate to be returned is in $\mathcal{L}_{Q_I}$ or in the separate data structure for the $\mathcal{C}_{ap}(I, \alpha)$ candidates. Finally, for each pattern interval $I$ the data structure stores the fingerprint of $u_I$, the fingerprint of the principal period of $u_I$ and the length of the principal period of $u_I$.

**Lemma 17** *There exists an implementation of candidate-fingerprint-queues such that the queue $Q_I$ at time $\alpha > 0$ maintains all the candidates of $\mathcal{C}_{ap}(I, \alpha)$ and their SI using $O(1)$ words of space.*

***Proof*** If $|\mathcal{C}_{ap}(I, \alpha)| \le 2$ then $Q_I$ stores the candidates of $\mathcal{C}_{ap}(I, \alpha)$ explicitly in $O(1)$ words of space. Otherwise, by Lemma 16, all the candidates of $\mathcal{C}_{ap}(I, \alpha)$ form an arithmetic progression. An arithmetic progression of arbitrary length can be represented using $O(1)$ words of space. However, $Q_I$ also needs access to the SI for the candidates in this progression. To do this, $Q_I$ explicitly stores the first candidate ($\min \mathcal{C}_{ap}(I, \alpha)$) together with its SI, the common difference of the progression ($\rho_{u_I}$), the length of the current progression, and the fingerprint of the principal period of $u_I$. When a new candidate $c$ with entrance fingerprint $\phi(u_I)$ enters $Q_I$, $c$ becomes the largest element in $\mathcal{C}_{ap}(I, \alpha)$, and so we first increment the length of the arithmetic progression, and if $c$ is currently the only candidate in the arithmetic progression, then $Q_I$ stores $c$ and its SI (since then $c$ is the first candidate in the progression). When a `Dequeue()` operation needs to remove the first candidate $c$ in the progression, then $Q_I$ removes $c$, which is stored explicitly together with its SI, decrements the length of the progression, and if there are remaining candidates in the progression then $Q_I$ computes the information for the new first remaining candidate in order to store its information explicitly. To do this, $Q_I$ first computes the location of the new first candidate from $\rho_{u_I}$ and the location of $c$. The SI of the new first candidate is computed in constant time (via the sliding property) from the fingerprint of the principal period of $u_I$ and the candidate fingerprint of $c$. $\square$

*Space usage.* The space usage of all of the queues has three components. The first component is the lists $\mathcal{L}_{Q_I}$, which maintains the candidates of $\overline{\mathcal{C}_{ap}}(I, \alpha)$ for all the intervals $I$. The second component is the data structures for storing the candidates with entrance prefix $u_I$ (the candidates of $\mathcal{C}_{ap}(I, \alpha)$) in each $I \in \mathcal{I}$. Since, by Lemma 17, for each $I \in \mathcal{I}$ all the candidates with entrance prefix $u_I$ are maintained using $O(1)$ words, all such candidates use $O(|\mathcal{I}|) = O(d + \log m)$ words of space. The third component is storing for each pattern interval $I$ the fingerprint of $u_I$, the fingerprint of the principal period of $u_I$ and the length of the principal period of $u_I$, which takes a total of $O(d + \log m)$ words of space. In the following lemma we prove that the total space usage of all of the lists $\mathcal{L}_{Q_I}$ is $O(d \log m)$.

**Lemma 18** $\sum_{I\in\mathcal{I}} \left|\overline{\mathcal{C}_{ap}}(I,\alpha)\right| = O(d\log m)$.

***Proof*** By Lemma 10, we know that $|\{\mu(I_0),\ldots,\mu(I_k)\}| = O(\log m)$. For each $\ell \in \{\mu(I_0),\ldots,\mu(I_k)\}$ let $\mathcal{I}_\ell \subseteq \mathcal{I}$ be the sequence of all pattern intervals $I \in \mathcal{I}$ such that $\mu(I) = \ell$. We show that $\sum_{I\in\mathcal{I}_\ell} |\overline{\mathcal{C}_{ap}}(I,\alpha)| = O(|\mathcal{I}_\ell| + d)$. Notice that $\sum_{\ell\in\{\mu(I_0),\ldots,\mu(I_k)\}} |\mathcal{I}_\ell| = |\mathcal{I}|$. Combining with Property 2 of Lemma 10 which states that $|\mathcal{I}| = O(d + \log m)$ we have that:

$$\begin{aligned}
\sum_{I\in\mathcal{I}} \left|\overline{\mathcal{C}_{ap}}(I,\alpha)\right| &= \sum_{\ell\in\{\mu(I_0),\ldots,\mu(I_k)\}} \sum_{I\in\mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}}(I,\alpha)\right| \\
&= \sum_{\ell\in\{\mu(I_0),\ldots,\mu(I_k)\}} O(|\mathcal{I}_\ell| + d) \\
&= \sum_{\ell\in\{\mu(I_0),\ldots,\mu(I_k)\}} O(|\mathcal{I}_\ell|) + \sum_{\ell\in\{\mu(I_0),\ldots,\mu(I_k)\}} O(d) \\
&= O(d + \log m) + O(d\log m) \\
&= O(d\log m)
\end{aligned}$$

We focus on intervals for which $\left|\overline{\mathcal{C}_{ap}}(I,\alpha)\right| \geq 3$, since if $\left|\overline{\mathcal{C}_{ap}}(I,\alpha)\right| \leq 2$ the bound is straightforward.

Let $[i^*, j^*]$ be the leftmost interval in $\mathcal{I}_\ell$. By definition of $\mathcal{I}_\ell$, we have $j^* - i^* + 1 = \ell$, and so by Property 3 of Lemma 10, there exists a string $v$ of length $\ell$ containing only non-wildcard characters that is a substring of the length $i^*$ prefix of $P$. Let $r$ be an arbitrary location of $v$ in $p_0 \ldots p_{i^*-1}$ (since $v$ could appear several times in the prefix). For any $[i', j'] = I' \in \mathcal{I}_\ell$ the entrance prefix (which does not contain wildcards) of each candidate in $\mathcal{C}(I',\alpha)$ matches the $i'$ length prefix of $P$ (which can contain wildcards), and in particular, the location which is $r$ locations to the left of any candidate in $\mathcal{C}(I',\alpha)$ is a location of an occurrence of $v$ in the text.[5]

Since we focus on intervals $I \in \mathcal{I}_\ell$ for which $|\overline{\mathcal{C}_{ap}}(I,\alpha)| \geq 3$, then there exist three occurrences of $v$ in the text in positions corresponding to a shift of $r$ characters from locations of $I$'s candidates. These occurrences are within a substring of the text of length at most $2|v|$, since all three candidates are in $\mathcal{C}(I,\alpha)$ and so the distance between the first and the last candidates is at most $|I| - 1 \leq \ell - 1 = |v| - 1$. Thus, by Lemma 5, $v$ must be periodic, and the distance between any two candidates in $\mathcal{C}(I,\alpha)$ must be a multiple of $\rho_v$.

Let $\hat{c} = \max\left[\bigcup_{I\in\mathcal{I}_\ell} \mathcal{C}(I,\alpha)\right]$ be the rightmost (largest index) candidate in the intervals corresponding to pattern intervals in $\mathcal{I}_\ell$. Since $\hat{c}$ is a candidate in some $\mathcal{C}(I',\alpha)$

[5] Notice that this occurrence is well defined since $i' \geq i^* \geq r + |v|$.

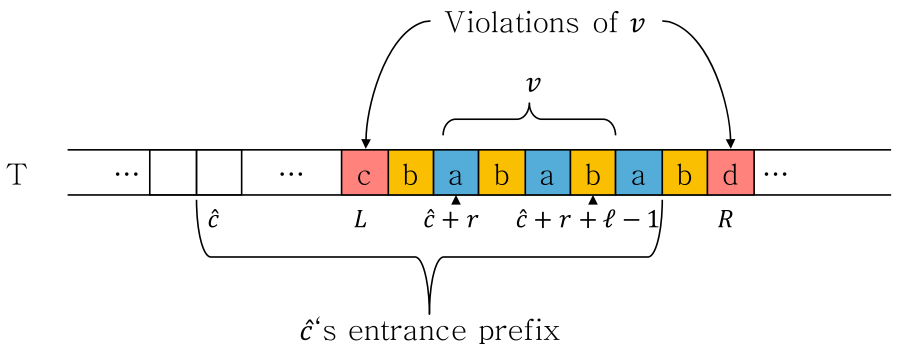


**Fig. 8** Positions $L$ and $R$ are the violations of the periodic substring that contains $v$. Notice that it is possible that $L \leq \hat{c}$, and similarly it is possible that $R$ is in the entrance interval of $\hat{c}$

for $I' \in \mathcal{I}_\ell$, then there is an occurrence of $v$ at location $\hat{c}+r$. Thus, $t_{\hat{c}+r} \dots t_{\hat{c}+r+\ell-1} = v$. We extend this occurrence of $v$ to the left and to the right in $T$ for as long as the length of the period does not increase (it certainly cannot decrease). Let the resulting substring be $t_{L+1} \dots t_{R-1}$. See Fig. 8. If $L \geq 0$ then the index $L$ is called the *left violation* of $v$. Similarly, if $R \leq \alpha$ then the index $R$ is called the *right violation* of $v$. Notice that the period of $v$ extends all the way to the beginning of the text if and only if $L = -1$, in which case there is no left violation. Similarly, the period of $v$ extends all the way to the current end of the text if and only if $R = \alpha + 1$, in which case there is no right violation. Finally, notice that $L < \hat{c} + r \leq \hat{c} + r + \ell - 1 < R$, since $v$ is a substring of $t_{L+1} \dots t_{R-1}$.

For a candidate $c \in$ `text_interval`$([i, j], \alpha)$ we define the *entrance interval* of $c$ to be $[c, c+i-1]$. In addition we denote $e_c = c+i-1$, so the entrance interval of $c$ is $[c, e_c]$. In the following claim we prove that the entrance interval of any candidate in $\mathcal{C}(I, \alpha)$ must contain location $\hat{c}+r$, which implies that the entrance prefix of any candidate in $\mathcal{C}(I, \alpha)$ must overlap with the occurrence of $v$ that corresponds to $\hat{c}$.

**Claim 1** *For any candidate $c \in \mathcal{C}(I, \alpha)$ where $I \in \mathcal{I}_\ell$ we have $\hat{c}+r \in [c, e_c]$.*

***Proof*** Let $c$ be a candidate in $\mathcal{C}(I, \alpha)$ for $I = [i, j] \in \mathcal{I}_\ell$. Recall that $\mathcal{C}(I, \alpha) \subseteq$ `text_interval`$(I, \alpha) = [\alpha - j + 1, \alpha - i + 1]$. Since $c$ is in this interval we have that $\alpha - j + 1 \leq c \leq \alpha - i + 1$. In particular, $e_c = c + i - 1 \geq \alpha - j + 1 + i - 1 = \alpha - (j - i + 1) + 1 = \alpha - |I| + 1$. By definition, since $I \in \mathcal{I}_\ell$, we have that $|I| \leq \ell$ and so $e_c \geq \alpha - \ell + 1$. Since $t_{\hat{c}+r} \dots t_{\hat{c}+r+\ell-1} = v$, it must be that $\hat{c}+r+\ell-1 \leq \alpha$. Thus, $\hat{c}+r \leq \alpha - \ell + 1 \leq e_c$. By the maximality of $\hat{c}$, it is obvious that $c \leq \hat{c} \leq \hat{c}+r$. Hence, we have that $c \leq \hat{c}+r \leq e_c$. □

In the following claim we argue that the entrance prefix of any candidate $c \in \overline{\mathcal{C}_{ap}}(I, \alpha)$ must overlap with either the left violation $L$ or the right violation $R$.

**Claim 2** *Suppose $I = [i, j] \in \mathcal{I}_\ell$ and $|\overline{\mathcal{C}_{ap}}(I, \alpha)| \geq 3$. Then for any candidate $c \in \overline{\mathcal{C}_{ap}}(I, \alpha)$ either $L \in [c, e_c]$ or $R \in [c, e_c]$.*

***Proof*** For $c \in \overline{\mathcal{C}_{ap}}(I, \alpha)$ let $u$ be the entrance prefix of $c$. Recall that $L < \hat{c} + r < R$. By Claim 1 it must be that $c \leq \hat{c} + r \leq e_c$ and so we cannot have both $L, R < c$ or both $L, R > e_c$.

Assume by contradiction that $L < c \leq e_c < R$. Recall that the principal period length of $t_{L+1} \ldots t_{R-1}$ is $\rho_v$. Since $u = t_c \ldots t_{e_c}$ is a substring of $t_{L+1} \ldots t_{R-1}$, it must be that $\rho_u \leq \rho_v \leq \frac{\ell}{2}$. Therefore, due to Lemma 13, since $u$ matches the $i$ length prefix of $P$ and $\rho_u \leq \frac{\ell}{2} = \frac{\mu(I)}{2}$ it must be that $u = u_I$, which contradicts $c \in \overline{\mathcal{C}_{ap}}(I, \alpha)$. □

Let $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ be the set of candidates in $\overline{\mathcal{C}_{ap}}(I, \alpha)$ whose entrance interval contains $L$, and let $\overline{\mathcal{C}_{ap}^{right}}(I, \alpha)$ be the set of candidates in $\overline{\mathcal{C}_{ap}}(I, \alpha)$ whose entrance interval contains $R$. $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ and $\overline{\mathcal{C}_{ap}^{right}}(I, \alpha)$ are not necessarily disjoint. Notice that by Claim 2, $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha) \cup \overline{\mathcal{C}_{ap}^{right}}(I, \alpha)$ contains all the candidates of $\overline{\mathcal{C}_{ap}}(I, \alpha)$.

**Claim 3** $\sum_{I \in \mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)\right| = O(|\mathcal{I}_\ell| + d)$ *and* $\sum_{I \in \mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}^{right}}(I, \alpha)\right| = O(|\mathcal{I}_\ell| + d)$.

***Proof*** Let $I \in \mathcal{I}_\ell$ and let $\approx$ denote the match relation between symbols in $\Sigma \cup \{?\}$.

Notice that the contribution to $\sum_{I \in \mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)\right|$ from all sets $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ that have less than two candidates is at most $O(|\mathcal{I}_\ell|)$. Thus, we will prove that for any set $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ with at least two candidates, it must be that for any candidate $c \in \overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$, except for possibly one candidate, we have that $p_{L-c}$ is a wildcard.

Suppose $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ contains at least two candidates and let $c_{left} = \max \overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ be the most recent candidate in $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$. Let $c < c_{left}$ be a candidate in $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$. Since $c \in \overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ we have that $p_{L-c} \approx t_{c+L-c} = t_L$ (recall that both $L$ and $c$ are indices in the text). Similarly, since $c_{left} \in \overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ we have that $p_{L-c} \approx t_{c_{left}+L-c} = t_{L+(c_{left}-c)}$. Recall that the distance between any two candidates in $\mathcal{C}(I, \alpha)$ is a multiple of $\rho_v$, since $\mathcal{C}(I, \alpha)$ contains at least 3 candidates. In particular the distance $(c_{left} - c)$ is a multiple of $\rho_v$ and $(c_{left} - c) \leq |I| \leq |v|$. Thus, $t_L \neq t_{L+(c_{left}-c)}$ since $L$ violates the period of length $\rho_v$. Recall that $t_L \approx p_{L-c} \approx t_{L+(c_{left}-c)}$, and so $p_{L-c}$ must be a wildcard. Therefore, each $c \in \overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$, except for possibly $c_{left}$, is in a position $c$ such that $p_{L-c}$ is a wildcard. Since $L$ is the same for all of the candidates in all of the $\overline{\mathcal{C}_{ap}^{left}}(I', \alpha)$ for all $I' \in \mathcal{I}_\ell$, then the contribution to $\sum_{I \in \mathcal{I}_\ell} |\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)|$ of the candidates that are not the most recent in their set $\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)$ is at most $d$. The contribution of the most recent candidates is at most $O(|\mathcal{I}_\ell|)$. Thus, $\sum_{I' \in \mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}^{left}}(I', \alpha)\right| = O(|\mathcal{I}_\ell| + d)$.

The proof that $\sum_{I' \in \mathcal{I}_\ell} |\overline{\mathcal{C}_{ap}^{right}}(I, \alpha)| = O(|\mathcal{I}_\ell| + d)$ is symmetric. □

Finally, by Claims 2 and 3, $\sum_{I \in \mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}}(I, \alpha)\right| \leq \sum_{I \in \mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}^{left}}(I, \alpha)\right| + \sum_{I \in \mathcal{I}_\ell} \left|\overline{\mathcal{C}_{ap}^{right}}(I, \alpha)\right| = O(|\mathcal{I}_\ell| + d)$. Thus, we have completed the proof of Lemma 18. □

## 6 The Algorithm of Theorem 3

The algorithm of Theorem 2 for PMDW uses $\tilde{O}(d)$ time per character and $\tilde{O}(d)$ words of space. In this section we introduce the algorithm of Theorem 3 which extends this result for a parameter $0 \le \delta \le 1$ to an algorithm that uses $\tilde{O}(d^{1-\delta})$ time per character and $\tilde{O}(d^{1+\delta})$ words of space.

An overview of a slightly modified version (for the sake of intuition) of the tradeoff algorithm is described as follows. Let $P^*$ be the longest prefix of $P$ such that $\pi_{P^*} \le d^\delta$. The tradeoff algorithm first finds all the occurrences of $P^*$ in $T$ using a specialized algorithm for patterns with bounded wildcard-period length. If $P^* = P$ then this completes the tradeoff algorithm. Otherwise, let $I = [i, j]$ be the interval in the secondary partitioning of Theorem 2 such that $i \le |P^*| - 1 \le j$. We first divide $I$ into two new intervals $[i, |P^*| - 1]$ and $[|P^*|, j]$. If $[|P^*|, j] = \emptyset$ then we discard $[|P^*|, j]$. It is straightforward to see that the properties of partitions that we define in Lemma 10 are still satisfied. Let $I^* = [i^* = |P^*|, j^*]$ be the interval immediately following $[i, |P^*| - 1]$. Each occurrence of $P^*$ in the text is inserted into the algorithm of Theorem 2 as a candidate directly into $Q_{I^*}$. Thus, the entrance prefixes of candidates in the queues match prefixes of $P$ that are longer than $P^*$ and, by maximality of $P^*$, these prefixes of $P$ have large wildcard-period length. This implies that the average distance between two consecutive candidates that are occurrences of $P^*$ is at least $d^\delta$, and so, combined with a carefully designed scheduling approach for verifying candidates, we are able to obtain an $\tilde{O}(d^{1-\delta})$ amortized time cost per character.

*Overview.* In Sect. 6.1 we describe the specialized algorithm for dealing with patterns whose wildcard-period length is at most $\tau$, for some parameter $\tau > 1$. In Sect. 6.2 we complete the proof of Theorem 3 by describing the missing details for the tradeoff algorithm. In particular, the proof of Theorem 3 uses the algorithm of Sect. 6.1 with $\tau = d^\delta$.

### 6.1 Patterns with Small Wildcard-Period Length

Let $P$ be a pattern of length $m$ with $d$ wildcards such that $\pi_P < \tau$. Let $q$ be an integer, which for simplicity is assumed to divide $m$ (see the remark at the end of this section where we discuss how to get rid of this assumption). Consider a conceptual matrix $M^q = \{m^q_{x,y}\}$ of size $\frac{m}{q} \times q$ where $m^q_{x,y} = p_{(x-1)\cdot q+y-1}$. An example is given in Fig. 9. For any integer $0 \le r < q$ the $r$'th column of $M^q$ represents an *offset pattern* $P_{q,r} = p_r p_{r+q} p_{r+2q} \ldots p_{m-q+r}$. Notice that some offset patterns might be equal. Let $\Gamma_q = \{P_{q,r} \mid 0 \le r < q$ and '?' $\notin P_{q,r}\}$ be the set of all the offset patterns that do not contain any wildcards. Each offset pattern in $\Gamma_q$ is given a unique id. The set of unique ids is denoted by $ID_q$. We say that index $i$ in $P$ is *covered* by $q$ if the column containing $p_i$ does not contain a wildcard, and so $P_{q,i \bmod q} \in \Gamma_q$. The columns of $M^q$ define a *column pattern* $P_q$ of length $q$, where the $j$'th character is the id of the $P_{q,j}$ column, or '?' if $P_{q,j} \notin \Gamma_q$ (since $P_{q,j}$ contains wildcards).

We partition $T$ into $q$ *offset texts*, where for every $0 \le r < q$ we define $T_{q,r} = t_r t_{r+q} t_{r+2q} \ldots$. Using the dictionary matching streaming (DMS) algorithm of Golan and Porat [25] we look for occurrences of offset patterns from $\Gamma_q$ in each of the offset texts. We emphasize that we do *not* only find occurrences of $P_{q,r}$ in $T_{q,r}$, since we

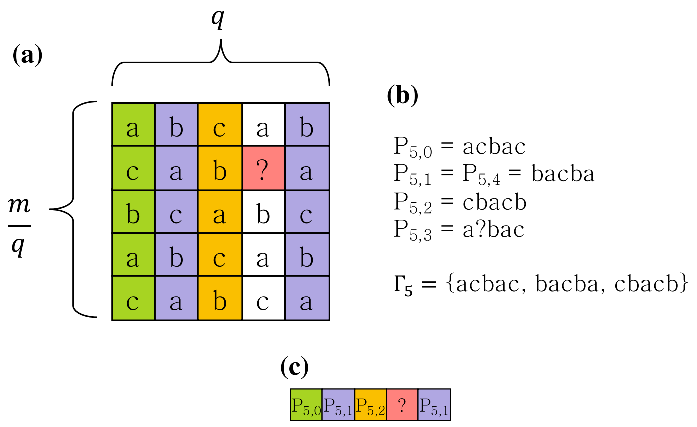


**Fig. 9** Example of the matrix representation for pattern $P = abcabcab?abcabcabcabcabca$ and $q = 5$. Each color represents a unique offset pattern. The offset patterns $P_{5,1}$ and $P_{5,4}$ are equal and therefore they have the same id (column color). Since $P_{5,3}$ contains a wildcard, it is not associated with any id (Color figure online)

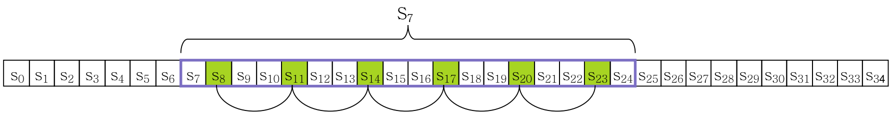
cannot guarantee that the offset of $T$ synchronizes with an occurrence of $P$. When the character $t_\alpha$ arrives, the algorithm passes $t_\alpha$ to the DMS algorithm for $T_{q,\alpha \bmod q}$. We also create a streaming *column text* $T_q$ whose characters correspond to the ids of offset patterns as follows. If one of the offset patterns is found in $T_{q,\alpha \bmod q}$, then its id is the $\alpha$'th character in $T_q$. Otherwise, we use a dummy character for the $\alpha$'th character in $T_q$.

*Full cover.* Notice that an occurrence of $P$ in $T$ necessarily creates an occurrence of $P_q$ in $T_q$. Such occurrences are found via the black box algorithm of Clifford et al. [10]. However, an occurrence of $P_q$ in $T_q$ does not necessarily mean there was an occurrence of $P$ in $T$, since some characters in $P$ are not covered by $q$. In order to avoid such false positives we run the process in parallel with several choices of $q$, while guaranteeing that each non-wildcard character in $P$ is covered by at least one of those choices. Thus, if there is an occurrence of $P_q$ at location $i$ in $T_q$ for all the choices of $q$, then it must be that $P$ appears in $T$ at location $i$. The choices of $q$ are given by the following lemma.

**Lemma 19** *There exists a set $Q$ of $O(\log d)$ prime numbers such that any index of a non-wildcard character in $P$ is covered by at least one prime number $q \in Q$, and each number in $Q$ is at most $\tilde{O}(d)$.*

***Proof*** The proof uses the probabilistic method: we show that the probability that the set $Q$ exists is strictly larger than 0. Since our proof is constructive it provides a randomized construction of $Q$.

It is well known that for a prime number $q$, every integer $0 \le z < q$ defines a congruence class which contains all integers $i$ such that $i \bmod q = z$. For any two distinct natural numbers $x, y \in \mathbb{N}$, let $D_{x,y}$ be the set of prime numbers $q$ such that $x$ and $y$ are in the same congruence class modulo $q$ (i.e. $x \bmod q = y \bmod q$). Notice that in the interpretation of the pattern columns in the conceptual matrix, if $q \in D_{x,y}$ then $p_x$ and $p_y$ are in the same column of the conceptual matrix $M^q$. Recall that $W$ is the set of occurrences of wildcards in $P$. Thus, if $0 \le j < m$ is an index such that $j \notin W$ and if



$S_7$
$s_0$ $s_1$ $s_2$ $s_3$ $s_4$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$ $s_{13}$ $s_{14}$ $s_{15}$ $s_{16}$ $s_{17}$ $s_{18}$ $s_{19}$ $s_{20}$ $s_{21}$ $s_{22}$ $s_{23}$ $s_{24}$ $s_{25}$ $s_{26}$ $s_{27}$ $s_{28}$ $s_{29}$ $s_{30}$ $s_{31}$ $s_{32}$ $s_{33}$ $s_{34}$

**Fig. 10** For string $S = s_0 \dots s_{34}$ for pattern of length $m = 18$, $S_7$ is marked by the blue rectangle and the green indices are the characters of $S_{7,3,1}$. Notice that $S_{7,3,1} = S_{8,3,0}$ (Color figure online)

$w \in W$ such that $q \in D_{j,w}$, then $j$ is surely not covered by $q$. By the Chinese remainder theorem, $|D_{j,w}| < \log m$ (otherwise for $\gamma = \prod_{q \in D_{j,w}} q > \prod_{q \in D_{j,w}} 2 \geq m$, and so $j \bmod \gamma = w \bmod \gamma$ implying that $j = w$).

For any $0 \leq j < m$ such that $j \notin W$, let $D_j = \bigcup_{w \in W} D_{j,w}$, so $|D_j| \leq \sum_{w \in W} |D_{j,w}| < |W| \log m = d \log m$. If $2d \geq \frac{m}{\log^2 m}$ then the proof is trivialized by choosing $Q$ to contain only the smallest prime number which is at least $m$. If $2d < \frac{m}{\log^2 m}$, by Corollary 1 in [39], then there are at least $2d \log m$ prime numbers whose values are upper bounded by $2d \log^2 m$. Let $\hat{Q}$ be the set of those prime numbers. For a random $q \in \hat{Q}$, the probability that a specific non-wildcard pattern index $j$ is not covered by $q$ is at most $\frac{|D_j|}{|\hat{Q}|} \leq \frac{d \log m}{2d \log m} = \frac{1}{2}$. Let $Q$ be a set of $2 \log m$ randomly chosen prime numbers from $\hat{Q}$. The probability that a specific non-wildcard pattern index $j$ is not covered by any of the prime numbers in $Q$ is less than $\frac{1}{2^{2 \log m}} \leq \frac{1}{m^2}$. Thus, the probability that there exists a non-wildcard pattern index $j$ which is not covered by any of the prime numbers in $Q$ is less than $\frac{m-d}{m^2} \leq \frac{1}{m}$. Therefore, there must exist a set $Q$ that covers all of the indices of non-wildcard characters from $P$. □

From a space usage perspective, we need the size of $|\Gamma_q|$ to be small, since this directly affects the space usage of the DMS algorithm which uses $\tilde{O}(k)$ space, where $k$ is the number of patterns in the dictionary. In our case $k = |\Gamma_q|$. In order to bound the size of $\Gamma_q$ we use the following lemma.

**Lemma 20** *If $\pi_P \leq \tau$ then for any $q \in \mathbb{N}$ we have $|\Gamma_q| \leq 2\tau$.*

***Proof*** Since $\pi_P \leq \tau$, there exists a string $S = s_0 \dots s_{2m-2}$ with no wildcards that contains $\Omega(\frac{m}{\tau})$ occurrences of $P$. Using the string $S$ we show that $|\Gamma_q| = O(\tau)$.

For each id in $ID_q$ we pick an index of a representative column in $M_q$ that has this id, and denote this set by $R_q$. Let $r_1$ be the minimum index in $R_q$. For every index $0 \leq i < m$ let $S_i = s_i \dots s_{i+m-1}$ (see Fig. 10). For every $0 \leq r < q$ let $S_{i,q,r} = s_{i+r} s_{i+r+q} \dots s_{i+m-q+r}$, and so for any integer $0 \leq \Delta < q - r$ we have $S_{i,q,r+\Delta} = S_{i+\Delta,q,r}$. Notice that if $S_i$ matches $P$ then $P_{q,r} = S_{i,q,r}$ for each $r \in R_q$.

Let $i$ be an index of an occurrence of $P$ in $S$. For any distinct $r, r' \in R_q$, it must be that $S_{i,q,r} = P_{q,r} \neq P_{q,r'} = S_{i,q,r'}$. In particular, for any $r \in R_q$ such that $r > r_1$, we have $P_{q,r_1} = S_{i,q,r_1} \neq S_{i,q,r} = S_{i+r-r_1,q,r_1}$. This implies that $i + r - r_1$ cannot be an occurrence of $P$. Hence, every occurrence of $P$ in $S$ eliminates $|R_q| - 1$ locations in $S$ from being an occurrence of $P$. We now show that the sets of eliminated locations defined by distinct occurrences are disjoint. Assume without loss of generality that $S$ contains at least two occurrences. Let $i_1$ and $i_2$ be two distinct occurrences of $P$ in $S$, and assume by contradiction that an index $j$ is eliminated by both of these occurrences. Since $s_{i_1} \dots s_{i_1+m-1}$ matches $P$, we have that $S_{i_1,q,j-i_1} = P_{j-i_1}$ and $j - i_1 \in R_q$. Similarly, we have that $S_{i_2,q,j-i_2} = P_{j-i_2}$ and $j - i_2 \in R_q$. Being that $S_{i_1,q,j-i_1} = S_{i_2,q,j-i_2}$ we have that $P_{j-i_2} = P_{j-i_1}$, contradicting the definition of $R_q$.

Therefore, the maximum number of occurrences of $P$ in $S$ is at most $\frac{|S|}{|R_q|} = \frac{2m-1}{|R_q|}$. Since $S$ contains at least $\frac{m}{\tau}$ instances of $P$, it must be that $\frac{m}{\tau} \le \frac{2m-1}{|R_q|}$ which implies that $|\Gamma_q| = |R_q| \le 2\tau$. □

*Complexities.* For a single $q \in Q$, the algorithm creates $q = \tilde{O}(d)$ offset patterns and texts. For each such offset text the algorithm applies an instance of the DMS algorithm with a dictionary of $O(\tau)$ strings (by Lemma 20). Since each instance of the DMS algorithm uses $\tilde{O}(\tau)$ words of space [25], the total space usage for all instances of the DMS algorithm is $\tilde{O}(d\tau)$ words. Moreover, the time per character in each DMS algorithm is $\tilde{O}(1)$ time, and each time a character appears we inject it into only one of the DMS algorithms (for this specific $q$). In addition, the algorithm uses an instance of the black box algorithm for $T_q$, with a pattern of length $q$. This uses another $O(q) = \tilde{O}(d)$ space and another $\tilde{O}(1)$ time per character [10]. Thus the total space usage due to one element in $Q$ is $\tilde{O}(d\tau)$ words. Since $|Q| = O(\log d)$ the total space usage for all elements in $Q$ is $\tilde{O}(d\tau)$ words, and the total time per arriving character is $\tilde{O}(1)$. Thus we have proven the following.

**Theorem 21** *For any $\tau \ge 1$, there exists a randomized Monte Carlo algorithm for PMDW on patterns $P$ with $\pi_P < \tau$ in the streaming model, which succeeds with probability $1 - 1/poly(n)$, uses $\tilde{O}(d\tau)$ words of space and spends $\tilde{O}(1)$ time per arriving text character.*

*Remark on dealing with $q \nmid m$* If $q \nmid m$, then the strings in $\Gamma_q$ have two possible lengths; either $\left\lfloor \frac{m}{q} \right\rfloor$ or $\left\lceil \frac{m}{q} \right\rceil$. This implies that one string in $\Gamma_q$ could be a proper suffix of another string in $\Gamma_q$. So if the longer one appears in an offset text, then both ids need to be given to $T_q$—a situation in which it is not clear what to do. So to avoid such scenarios, for each $q \in Q$ we run the algorithm twice, in parallel, where one instance uses the DMS algorithm for one length while the other instance uses the DMS algorithm on the other length. This creates two instances of $P_q$ and $T_q$, one for each length of columns under consideration. Notice that in order for the algorithm to work, when considering one specific length, all of the columns that correspond to the other length are treated as a '?' in the appropriate instance of $P_q$.

### 6.2 Proof of Theorem 3

In this section we combine the algorithm of Theorem 2 with the algorithm of Theorem 21 and introduce an algorithm for patterns with general wildcard-period length, thereby proving Theorem 3.

Prior to Sect. 6.1 we presented an almost accurate description of the algorithm. The only two parts of the description that require elaboration are regarding how to insert occurrences of $P^*$ into the appropriate candidate-fingerprint-queue efficiently, and how to schedule validations of candidates so that the amortized cost is low. We first focus on how to insert candidates and later we discuss the scheduling.

*Direct insertion of candidates.* The challenge with inserting occurrences of $P^*$ into $Q_{I^*}$ is that the candidate-fingerprint-queue data structure uses the SI of candidates, and so the straightforward ways for providing this information together with the new candidates (which are occurrences of $P^*$) cost either too much time or too much space. In order to meet our desired complexities, we first investigate the purposes of different parts of SI.

The SI for a candidate $c$ in $\mathcal{C}(I = [i, j], \alpha)$ consists of the candidate fingerprint, $\phi(t_0 \dots t_{c-1})$, and the entrance fingerprint, $\phi(t_c \dots t_{c+i-1})$. The SI has two purposes. The first is to validate a candidate after a `Dequeue`() operation, in which case the algorithm makes use of both parts of the SI in order to compute $\phi(t_{c+i} \dots t_{c+j})$ by combining the SI with the text fingerprint. The second purpose is to compute the next entrance fingerprints of candidates in order to distinguish between candidates that are stored as part of an arithmetic progression and candidates that are not. The entrance fingerprint is obtained, via the sliding property, from the candidate fingerprint in the SI and the current text fingerprint.

Notice that in order to validate $c$ the algorithm only needs the fingerprint of $\phi(t_0 \dots t_{c-i+1})$. Also notice that entrance prefixes are only used for determining whether a candidate is part of an arithmetic progression or not (see the implementation details in Sect. 5.1). Thus, for a specially chosen subset of strings $\Psi \subseteq \Sigma^{|P^*|}$ we precompute all of the fingerprints of strings in $\Psi$. The set $\Psi$ is chosen so that for any occurrence of $P^*$ that is injected as a candidate $c$ where $c$ is at some point part of a stored arithmetic progression, the occurrence of $P^*$ at location $c$ is in $\Psi$. We use the DMS algorithm [25] to locate strings from $\Psi$ in the text, and whenever such a string appears, we compute the SI for the corresponding candidate in constant time from the stored fingerprint and the current text fingerprint. We emphasize that not all of the candidates that correspond to strings in $\Psi$ will be a part of an arithmetic progression. However, in order to reduce the space usage, we require that $\Psi$ is not too large, and in particular $|\Psi| = O(d + \log m)$. For a candidate $c$ that does not correspond to a string in $\Psi$, instead of maintaining the SI of $c$, we explicitly maintain the fingerprint of $\phi(t_0 \dots t_{c-i+1})$ where $c \in \mathcal{C}(I = [i, j], \alpha)$. Notice that whenever such a candidate enters a new text interval, the text fingerprint at that time is exactly the information which we need to store.

*Creating* $\Psi$. Consider all pattern intervals $I = [i, j] \in \mathcal{I}$ with $i \geq i^*$. Notice that there are at most $O(d + \log m)$ such pattern intervals. For each such interval $I$, let $\psi_I$ be the prefix of $u_I$ of length $|P^*|$. Since, by definition of $\mathcal{C}_{ap}(I, \alpha)$, a candidate $c \in \mathcal{C}_{ap}(I, \alpha)$ implies an occurrence of $u_I$ at location $c$, then $\psi_I$ also appears at location $c$. Thus, we define $\Psi$ to be the set containing $\psi_I$ for all such pattern intervals $I$. Since any candidate in an arithmetic progression at time $\alpha$ must be in $\mathcal{C}_{ap}(I, \alpha)$ for some interval $I$, it is guaranteed that when $c$ corresponded to an occurrence of $P^*$, that occurrence must have been $\psi_I$, and so $\Psi$ has the required properties.

*Scheduling validations.* Since the only bound we have proven on the number of pattern intervals $I = [i, j] \in \mathcal{I}$ with $i \geq i^*$ is $O(d + \log m)$, if each time a new text character arrives we perform a `Dequeue`() operation for each one of the pattern intervals, then the time cost can be as large as $O(d + \log m)$ which is too much. The solution for reducing this time cost is to only perform a `Dequeue`() operation on $Q_I$ when a candidate $c$ actually leaves `text_interval`$(I, \alpha)$ and needs to be validated. This is implemented by maintaining a priority queue on top of the pattern intervals,where the key of any text interval is the next time a candidate exits this text interval. Each time a candidate leaves a text interval, the key for the queue of that interval is updated to the time the next candidate leaves (if such a candidate exists). When a candidate entering a text interval is the only candidate of that text interval, then the key for the queue of this text interval is also updated.

*Complexities.* Recall that $I^* = [i^*, j^*]$ is a pattern interval such that $i^* = |P^*|$, and that each time the algorithm finds an occurrence of $P^*$, the corresponding candidate is inserted into $Q_{I^*}$. Let $P'$ be the prefix of $P$ of length $j^* + 1$. By maximality of $P^*$, it must be that $\pi_{P'} > d^\delta$. We partition the time usage of the algorithm into three parts. The first is the amount of time spent on finding occurrences of $P^*$ using the algorithm of Theorem 21, which is $\tilde{O}(1)$. The second is the amount of time spent performing `Enqueue()` and `Dequeue()` operations on $Q_{I^*}$, which is also $\tilde{O}(1)$ since we perform $O(1)$ operations on this queue per each arriving character. The third is the amount of time spent on `Enqueue()` and `Dequeue()` operations on $Q_I$ for $I = [i, j]$ with $i > j^*$. These operations only apply to candidates that are occurrences of $P'$. For this part we use amortized analysis.

By definition of wildcard-period length, for any string $S$ of size $2|P'| - 1$, we have $d^\delta < \pi_{P'} \leq \left\lceil \frac{|P'|}{occ(S,P')} \right\rceil$. Being that $occ(S, P') \leq |P'|$, we have $d^\delta < \frac{2|P'|}{occ(S,P')}$. Notice that for a text $T$ of size $n \geq |P'|$, we must have $occ(T, P') < \frac{2n}{d^\delta}$. This is because otherwise, if $n \geq 2|P'| - 1$ then there exists a substring of $n$ of length $2|P'| - 1$ with at least $\frac{2|P'|}{d^\delta}$ occurrences of $P'$, and if $n < 2|P'| - 1$ then we can pad $T$ to create such a string. In both cases we contradict $d^\delta < \frac{2|P'|}{occ(S,P')}$ for any string $S$ of length $2|P'| - 1$.

The total amount of time spent on each occurrence of $P'$ is $\tilde{O}(d)$, and so the total cost for processing $T$ on candidates that are also occurrences of $P'$ is at most $\tilde{O}(occ(T, P') \cdot d) = \tilde{O}(\frac{2n}{d^\delta} d) = O(n \cdot d^{1-\delta})$. Thus, the amortized cost per character is $\tilde{O}(d^{1-\delta})$.

For the space complexity, the most expensive part is the use of the algorithm of Theorem 21 which takes $O(d \cdot d^\delta) = O(d^{1+\delta})$ words of space. This completes the proof of Theorem 3.

## Appendix Algorithm Pseudo-Code

In this appendix we give a pseudo-code of the algorithm of Theorem 2.

```
PRE- PROCESS($P$)
1  Create the partitioning $\mathcal{I}$. // As defined in the proof of Lemma 10.
2  for each $I \in \mathcal{I}$:
3      $Q_I$.INIT()
4      if $u_I$ exists
5          Store $\phi(u_I)$, $\rho_{u_I}$ and $\phi(u_{I,0} \dots u_{I,\rho_{u_I}-1})$
6  INIT()
```

```
$Q$.INIT()
1 $ap \leftarrow$ new arithmetic progression
2 $ll \leftarrow$ new linked list

INIT()
1 $\phi_T \leftarrow \phi(\varepsilon)$ // Text fingerprint; $\varepsilon$ is the empty string
2 $Q_0$.ENQUEUE($\phi_T$) // Insert the first position as a candidate

PROCESS- CHARACTER($t_\alpha$)
 1 Update $\phi_T$ with $t_\alpha$ // $\phi_T = \phi(t_0 \dots t_\alpha)$
 2 for $h = 0$ to $k$
 3     $[i, j] \leftarrow I_h$
 4     if $Q_h$.Has_Candidate($\alpha - j + 1$)
 5         $(c, f_1, f_2) \leftarrow Q_h$.Dequeue() // $f_1 = \phi(t_0 \dots t_{c-1})$ and $f_2 = \phi(t_0 \dots t_{c+i-1})$
 6         Compute $\phi(t_{c+i} \dots t_\alpha)$ from $f_2$ and $\phi_T$ // $\alpha = c + j$
 7         if $I_h$ is a wildcard interval or $\phi(t_{c+i} \dots t_\alpha) = \phi(p_i \dots p_j)$
 8             if $h = k$
 9                 report $c$ as a match
10             else $Q_{h+1}$.ENQUEUE($\phi(t_0 \dots t_{c-1})$)
11 $Q_0$.ENQUEUE($\phi_T$) // insert the new candidate

$Q_I$.ENQUEUE($f$) //$f = \phi(t_0 \dots t_{\alpha-i})$
1 Compute $\phi(t_{\alpha-i+1} \dots \alpha)$ from $f$ and $\phi_T$
2 if $\phi(t_{\alpha-i+1} \dots t_\alpha) = \phi(u_I)$
3     $ap$.Insert($\alpha - i + 1, f$) // $ap$ maintains $\mathcal{C}_{ap}(I, \alpha)$ using an arithmetic progression
4 else
5     $ll$.Insert($\alpha - i + 1, f, \phi_T$)// $ll$ maintains $\overline{\mathcal{C}_{ap}}(I, \alpha)$ using a linked list

$Q_I$.DEQUEUE()
1 $c \leftarrow \alpha - j$
2 if $c \in \mathcal{C}_{ap}(I, \alpha)$
3     Remove $c$ from the arithmetic progression
4     Return $c$ with $\phi(t_0 \dots t_{c-1})$ and $\phi(t_0 \dots t_{c+i-1})$
5 elseif $c \in \overline{\mathcal{C}_{ap}}(I, \alpha)$
6     Remove $c$ from the linked list
7     Return $c$ with $\phi(t_0 \dots t_{c-1})$ and $\phi(t_0 \dots t_{c+i-1})$
8 else // $c \notin \mathcal{C}(I, \alpha)$
9         do nothing
```

## References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. **58**(1), 137–147 (1999)
2. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. J. Algorithms **50**(2), 257–275 (2004)
3. Berstel, J., Boasson, L.: Partial words and a theorem of fine and wilf. Theor. Comput. Sci. **218**(1), 135–141 (1999)
4. Blanchet-Sadri, F.: Algorithmic Combinatorics on Words. Discrete Mathematics and Its Applications. CRC Press, Boca Raton (2008)
5. Blanchet-Sadri, F., Hegstrom, R.A.: Partial words and a theorem of fine and wilf revisited. Theor. Comput. Sci. **270**(1–2), 401–419 (2002)

6. Breslauer, D., Galil, Z.: Real-time streaming string-matching. ACM Trans. Algorithms **10**(4), 22:1–22:12 (2014)
7. Breslauer, D., Grossi, R., Mignosi, F.: Simple real-time constant-space string matching. Theor. Comput. Sci. **483**, 2–9 (2013)
8. Cautis, S., Mignosi, F., Shallit, J., Wang, M., Yazdani, S.: Periodicity, morphisms, and matrices. Theor. Comput. Sci. **295**, 107–121 (2003)
9. Clifford, P., Clifford, R.: Simple deterministic wildcard matching. Inf. Process. Lett. **101**(2), 53–54 (2007)
10. Clifford, R., Efremenko, K., Porat, B., Porat, E.: A black box for online approximate pattern matching. Inf. Comput. **209**(4), 731–736 (2011)
11. Clifford, R., Efremenko, K., Porat, E., Rothschild, A.: From coding theory to efficient pattern matching. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, pp. 778–784 (2009)
12. Clifford, R., Efremenko, K., Porat, E., Rothschild, A.: Pattern matching with don't cares and few errors. J. Comput. Syst. Sci. **76**(2), 115–124 (2010)
13. Clifford, R., Fontaine, A., Porat, E., Sach, B., Starikovskaya, T.A.: Dictionary matching in a stream. In: Proceedings of the 23rd Annual European Symposium on Algorithms, ESA, pp. 361–372 (2015)
14. Clifford, R., Fontaine, A., Porat, E., Sach, B., Starikovskaya, T.A.: The k-mismatch problem revisited. In: Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, pp. 2039–2052 (2016)
15. Clifford, R., Jalsenius, M., Porat, E., Sach, B.: Space lower bounds for online pattern matching. Theor. Comput. Sci. **483**, 68–74 (2013)
16. Clifford, R., Kociumaka, T., Porat, E.: The streaming k-mismatch problem. CoRR. arXiv:1708.05223 (2017)
17. Clifford, R., Porat, E.: A filtering algorithm for $k$-mismatch with don't cares. In: Proceedings of the 14th International Symposium on String Processing and Information Retrieval, SPIRE, pp. 130–136 (2007)
18. Clifford, R., Sach, B.: Pseudo-realtime pattern matching: closing the gap. In: Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching, CPM, pp. 101–111 (2010)
19. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, STOC, pp. 592–601 (2002)
20. Ergün, F., Jowhari, H., Saglam, M.: Periodicity in streams. In: Proceedings of the 14th International Workshop on Randomization and Computation RANDOM, pp. 545–559 (2010)
21. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proc. Am. Math. Soc. **16**(1), 109–114 (1965)
22. Fischer, M.J., Paterson, M.S.: String-matching and other products. Technical report, DTIC document (1974)
23. Galil, Z., Seiferas, J.I.: Time-space-optimal string matching. J. Comput. Syst. Sci. **26**(3), 280–294 (1983)
24. Gawrychowski, P.: Optimal pattern matching in LZW compressed strings. ACM Trans. Algorithms **9**(3), 25 (2013)
25. Golan, S., Porat, E.: Real-time streaming multi-pattern search for constant alphabet. In: 25th Annual European Symposium on Algorithms, ESA 2017, pp. 41:1–41:15 (2017)
26. Henzinger, M.R., Raghavan, P., Rajagopalan, S.: External Memory Algorithms, Chapter Computing on Data Streams, pp. 107–118. American Mathematical Society, Providence (1999)
27. Indyk, P.: Faster algorithms for string matching problems: matching the convolution bound. In: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, FOCS, pp. 166–173 (1998)
28. Jalsenius, M., Porat, B., Sach, B.: Parameterized matching in the streaming model. In: Proceedings of the 30th International Symposium on Theoretical Aspects of Computer Science, STACS, pp. 400–411 (2013)
29. Kalai, A.: Efficient pattern-matching with don't cares. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms SODA, pp. 655–656 (2002)
30. Kane, D.M., Nelson, J., Porat, E., Woodruff, D.P.: Fast moment estimation in data streams in optimal space. In: Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC, pp. 745–754 (2011)

31. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
32. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
33. Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. Theor. Comput. Sci. **43**, 239–249 (1986)
34. Lee, L.-K., Lewenstein, M., Zhang, Q.: Parikh matching in the streaming model. In: Proceedings of the 19th International Symposium on String Processing and Information Retrieval, SPIRE, pp. 336–341 (2012)
35. Muthukrishnan, S.: Data streams: algorithms and applications. Found. Trends Theor. Comput. Sci. **1**(2), 117–236 (2005)
36. Muthukrishnan, S., Ramesh, H.: String matching under a general matching relation. In: Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS, pp. 356–367 (1992)
37. Porat, B., Porat, E.: Exact and approximate pattern matching in the streaming model. In: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS, pp. 315–323 (2009)
38. Porat, E., Lipsky, O.L: Improved sketching of hamming distance with error correcting. In: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, CPM, pp. 173–182 (2007)
39. Rosser, B.J., Schoenfeld, L.: Approximate formulas for some functions of prime numbers. Illinois J. Math. **6**, 64–94 (1962)
40. Smyth, W.F., Wang, S.: A new approach to the periodicity lemma on strings with holes. Theor. Comput. Sci. **410**(43), 4295–4302 (2009)