Streaming Pattern Matching with d Wildcards*

Shay Golan[1], Tsvi Kopelowitz[2] and Ely Porat[3]

1 **Bar Ilan University, Ramat Gan, Israel**
golansh1@cs.biu.ac.il
2 **University of Michigan, Ann Arbor, Michigan, USA**
kopelot@gmail.com
3 **Bar Ilan University, Ramat Gan, Israel**
porately@cs.biu.ac.il

Abstract

In the pattern matching with d wildcards problem we are 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.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases wildcards,don't-cares,streaming pattern matching,fingerprints

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.44

1 Introduction

We investigate one of the basic problems in pattern matching, the *pattern matching with d wildcards problem* (PMDW), in the *streaming model*. Let Σ be an alphabet and let $'?' \notin \Sigma$ be a special character called the *wildcard* which matches any character in Σ. The PMDW problem is defined as follows. Given a *text* string $T = t_0 t_1 \dots t_{n-1}$ over Σ 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 [21, 35, 26, 28, 18, 9].

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*

* Supported in part by NSF grants CCF-1217338, CNS-1318294, and CCF-1514383.

© Shay Golan, Tsvi Kopelowitz, and Ely Porat;
licensed under Creative Commons License CC-BY

model [1, 25, 34, 29]. 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 any matching substrings right after the last relevant text character has arrived, but before the next text character arrives. Another closely related model is the *online* model, which is the same as the streaming model without the constraint of using sublinear space.

Following the breakthrough result of Porat and Porat [36], there has recently been a rising interest in solving pattern matching problems in the streaming model [6, 19, 33, 7, 27, 13, 14]. However, this is the first paper to directly consider the important wildcard variant. 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 [32, 37, 2, 12, 11, 16, 14]. The most recent result by Clifford et al. [14] for the k-mismatch problem in the streaming model implies a solution for PMDW in the streaming model that uses $O(d^2 \operatorname{polylog} m)$ words[1] of space and $O(\sqrt{d}\log d + \operatorname{polylog} m)$ time per character. Notice that Clifford et al. [14] focused on solving a more general problem.

1.1 New results and Related Work

We improve upon the work of Clifford et al. [14], for the special case that applies to PMDW, by presenting the following algorithms (the $\tilde{O}$ notation hides logarithmic factors). Notice that Theorem 1 improves upon the results of Clifford et al. [14] whenever $\delta > 1/2$.

▶ **Theorem 1.** *For any constant* $0 \le \delta \le 1$ *there exists a 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.*

▶ **Theorem 2.** *There exists a 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.*

1.2 Algorithmic Overview and Related Work

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 proof that the candidate is not a valid occurrence (or until it is reported as a match). The algorithm of Theorem 2 works by obtaining such proofs efficiently. We discuss some of the ideas used in this algorithm after discussing the overview of Theorem 1.

Overview of algorithm for Theorem 1. The algorithm of Theorem 1 uses the algorithm of Theorem 2 (with a minor adaptation) combined with a new combinatorial perspective of *periodicity* that applies to strings with wildcards. The notion of periodicity in strings (without wildcards) and its usefulness are well studied [20, 31, 36, 6, 23, 22]. However, extending the usefulness of periodicity to strings with wildcards runs into difficulties, since the notions

---

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

are either too inclusive or too exclusive (see [4, 3, 5, 8, 38]). 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 5. For a string with wildcards S, we denote the wildcard-period length of S by π_S.

Let P^* be the longest prefix of P such that $\pi_{P^*} \leq d^\delta$. The algorithm of Theorem 1 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 uses $\tilde{O}(d^{1+\delta})$ words of space. This is summarized in Theorem 18. Of particular interest is Lemma 17 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 18 to locate occurrences of P^*, and by maximality of P^*, occurrences of any prefix of P that is 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 and combining with a lazy approach, we obtain an $\tilde{O}(d^{1-\delta})$ amortized time cost per character.

Overview of algorithm for Theorem 2. For the streaming pattern matching problem without wildcards, the algorithms of Porat and Porat [36] and Breslauer and Galil [6] have three major components[2]. The first component is a partitioning of the pattern into *pattern intervals* of exponentially increasing lengths. The algorithm uses *text intervals* corresponding to the pattern intervals, which are the reversed pattern intervals and appear at the end of the text. When a new text character arrives, 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. A candidate is tested each time it leaves a text interval.

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. Notice that the notion of wildcard-period length, for example, has to do with the average of distances between occurrences, and so it does not apply to arithmetic progressions. 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 the arithmetic progression of some 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.

More related work. We mention that while our work is in the streaming model, in the closely related online model (see [17, 15]), Clifford et al. [10] presented an algorithm, known

[2] The algorithms of Porat and Porat [36] 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.

as the black box algorithm, that when applied to PMDW uses $O(m)$ words of space and $O(\log^2 m)$ time per arriving text character.

Full version. Due to space consideration some of the proofs and details have been omitted, for a full version of this paper see [24].

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

Due to space reasons, we omit proofs here, proofs appear in the full version of the paper.

▶ **Lemma 3.** *Let v be a string of length ℓ and let u be a string of length at most 2ℓ. If u contains at least three occurrences of v then:*

1. *v is a periodic string.*
2. *the distance between any two occurrences of v in u is a multiple of ρ_v.*

▶ **Lemma 4.** *Let u be a periodic string over Σ with principal period length ρ_u. If v is a substring of u of length $> 2\rho_u$ then $\rho_u = \rho_v$.*

Periods and wildcards. For strings with no wildcards there is an inverse relation between the maximum number of occurrences of u in a text of a given length and ρ_u. Here 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 Section 6. Let $occ(S', S)$ be the number of occurrences of a string S in a string S'.

▶ **Definition 5.** For a string S over $\Sigma \cup \{?\}$, its wildcard-period length is

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

Notice that for periodic string S without wildcards $\pi_S = \rho_S$.

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 [36] and Breslauer and Galil [6] proved the existence of a *sliding fingerprint function* $\phi : \bigcup_{i=0}^{n} \Sigma^i \to [n^c]$, for some constant $c > 0$, which is a function where:

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 introduce a generic algorithm (pseudo-code is given in Figure 1) for solving online pattern matching problems. With proper implementations of its components, this generic 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 arrives. An index continues to be a candidate until the algorithm encounters 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; a running example of the execution of the generic algorithm appears in Figures 2 and 3:

- **Pattern and text intervals.** The first part is an ordered partitioning $\mathcal{I} = (I_0, \ldots, I_k)$ of the interval $[0, m-1]$. Each interval $I \in \mathcal{I}$ is called a **pattern interval**. When a character t_α arrives then a candidate c is alive if and only if there is a pattern interval $I = [i, j] \in \mathcal{I}$ such that $t_c \cdots t_{c+i-1}$ matches $p_0 \cdots p_{i-1}$ and $\alpha - c + 1 \in [i, j]$. Notice that for any pattern interval $I = [i, j]$, any candidate c that is alive in I must be in exactly one **text interval** $c \in [\alpha - j + 1, \alpha - i + 1]$. When a new text character arrives, all the text intervals move one position ahead, and some candidates move between intervals.
- **Candidate queues.** The second conceptual part of the generic algorithm is an implementation of a **candidate-queue** data structure. This data structure supports the following operations on candidates that are in the same text interval $[\alpha - j + 1, \alpha - i + 1]$, where α is the index of the last character to arrive from T.

▶ **Definition 6.** Let α be the index of the last text character that has arrived. Then a candidate-queue on an interval $I = [i, j]$ supports the following operations.

1. *Enqueue*(c): Given candidate $c = \alpha - i + 1$ add c to the candidate-queue.
2. *Dequeue*$()$: Remove and return a candidate $c = \alpha - j$, if it exists.

Since there is a bijection between pattern intervals and text intervals we say that a candidate-queue that is associated with a given text interval is also associated with the corresponding pattern interval.

- **Assassinating candidates.** The third conceptual part is a mechanism for testing if a candidate is alive after it leaves one text interval, 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 is no more text intervals.

The implementation of each of these components controls the complexities of the algorithm. Minimizing the number of intervals reduces the number of candidates leaving an interval at any given time. Efficient implementations of the queue operations and testing if a candidate is alive 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.

If there are no wildcards in P, one can use a sliding fingerprint based approach (related to the Karp and Rabin [30] algorithm) for testing if a candidate is alive. In order to use these fingerprints, we maintain the *text fingerprint* which is the fingerprint of the text from its beginning up to the last arriving character. This maintenance uses only constant time per character and constant space.

Process-Character(t_α)
1 $Q_0.Enqueue(\alpha)$
2 **for** $h = 0$ **to** k
3 $c = Q_h.Dequeue()$
4 **if** c exists and c is still alive
5 **if** $h = k$
6 report c as a match
7 **else** $Q_{h+1}.Enqueue(c)$

Figure 1 Generic Algorithm.

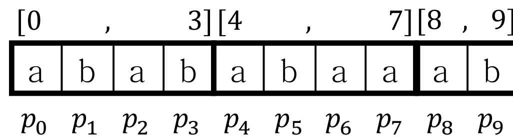

Figure 2 Example of a pattern and its arbitrary chosen pattern intervals. The pattern length is 10 and the pattern intervals are $[0,3]$, $[4,7]$ and $[8,9]$.

3.1 Fingerprints with Wildcards

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

Partitioning algorithm. We use a representation of P as $P = P_0?P_1?\dots?P_d$ where each subpattern P_i contains only characters from Σ (and may also be the empty string). Let $W = (w_1, w_2, \dots, w_d)$ be the indices such that $p_{w_i} = '?'$ and 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 intervals* 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 $[i,j]$, $t_{c+i} \dots t_{c+j} = p_i \dots p_j$. During the initialization of the algorithm we precompute and store the fingerprints for all of the subpatterns corresponding to regular intervals.

Testing liveness. Given the partition $\mathcal{J}$, the algorithm for testing if a candidate c is alive is as follows. Each time a candidate c is added to a candidate-queue for interval $[i,j] \in \mathcal{J}$ via the $Enqueue(c)$ operation, the algorithm stores the current text fingerprint $\phi(t_0 \dots t_{c+i-1})$ with the candidate c. When the character t_{c+j} arrives the text fingerprint is $\phi(t_0 \dots t_{c+j})$. At this time, the algorithm uses the $Dequeue()$ operation to extract c together with $\phi(t_0 t_1 \dots t_{c+i-1})$

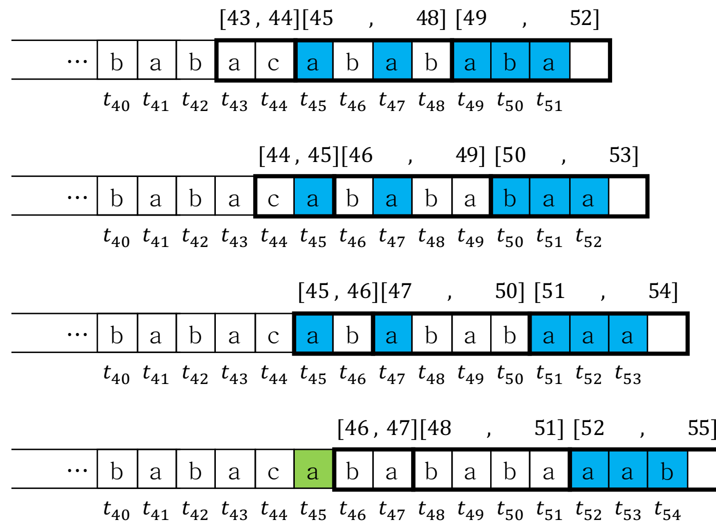

Figure 3 Example of the generic algorithm execution with the pattern of Figure 2. In each row a new text character arrives. The bold borders are on the text intervals, each blue cell is a position of a candidate and the green cell corresponds a match.
When t_{52} arrives the candidate $c_1 = 45$ is tested, since it exists a text interval and found to be alive because *abababaa* is a prefix of the pattern. At this time we can see that the candidate $c_2 = 47$ cannot be a valid occurrence of the pattern, however the algorithm will not remove it until it reaches the end of the text interval.
When t_{54} arrives, the candidates $c_1 = 45$ and $c_2 = 47$ are tested since they reach the end of their text intervals. c_2 is removed because the text *ababaaab* is not a prefix of the pattern. The candidate c_1 is still alive, and since it reaches the end of the last text interval, it reported as a match.

from the candidate-queue of interval $[i, j]$. If $[i, j]$ is a regular interval, then the algorithm tests if c is alive. This is done by applying the sliding property of the fingerprint function to compute $\phi(t_{c+i} \dots t_{c+j})$ from the current text fingerprint $\phi(t_0 t_1 \dots t_{c+j})$ and the fingerprint $\phi(t_0 t_1 \dots t_{c+i-1})$, and then testing if $\phi(t_{c+i} \dots t_{c+j})$ is the same as $\phi(p_i \dots p_j)$. If $[i, j]$ is a wildcard interval then c stays alive without any test.

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 time per character.

4 The Partitioning

The key idea of the new partitioning is to use the partitioning of Section 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 Section 5).

The following definition is useful in the next lemma.

▶ **Definition 7.** For an ordered set of intervals $\mathcal{I} = (I_0, I_1, \dots I_k)$ and for any integer $0 \leq i \leq k$, let $\mu_{\mathcal{I}}(i) = \max_{j=0}^{i} |I_j|$ be the length of the longest interval in the sequence $I_0, \dots I_i$. When $\mathcal{I}$ is clear from context we simply write $\mu(i) = \mu_{\mathcal{I}}(i)$

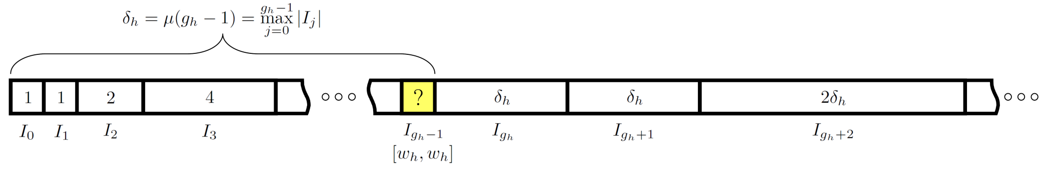

Figure 4 The general case: on each $J_h \in \mathcal{J}$ we first create two intervals of length δ_h and then we iteratively create pattern intervals where the length of each pattern interval is double the length of the previous pattern interval.

The following lemma shows 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. In the proof we introduce a specific partitioning which has all of these properties.

▶ **Lemma 8.** *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 $I = [i, j]$ 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.*
2. *$k = O(d + \log m)$.*
3. *For each regular pattern interval $I = [i, j]$ with $|I| > 1$, the length i prefix of P contains a consecutive sequence of $|I|$ non-wildcard characters.*
4. *$|\{\mu_{\mathcal{I}}(0), \mu_{\mathcal{I}}(1) \ldots \mu_{\mathcal{I}}(k)\}| = O(\log m)$.*

Proof. We introduce a secondary partitioning of the preliminary partitioning described in Section 3.1, and prove that it has all the required properties; see Figures 4, 5 and 6. 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, the secondary partitioning of J_h takes place only after the second partitioning of J_{h-1}.

When partitioning pattern interval $J_h = [i, j]$, let g_h be the number of pattern intervals in the secondary partitioning in $[0, i-1]$, and let $\delta_h = \mu_{\mathcal{I}}(g_h - 1)$ 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 $j \leq i + \delta_h - 1$ then the only pattern interval is all of J_h. If $j \leq i + 2\delta_h - 1$ 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 $[i + \delta_h, i + 2\delta_h - 1]$, 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 ℓ be the length of the last pattern interval we created. If the remaining part of the preliminary pattern interval is of length at most ℓ, then we create one pattern interval for all the remaining preliminary pattern interval. Otherwise we create two pattern intervals, the first pattern interval of length ℓ and the second pattern interval using the remaining part of J_h.

The secondary partitioning implies all of the desired properties, the proof appears in the full version of this paper. ◀

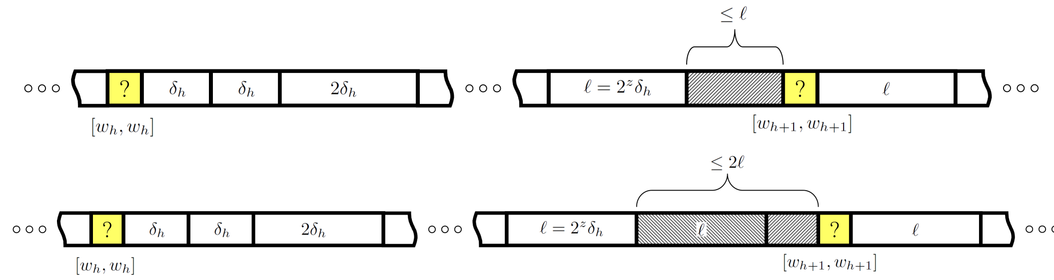

Figure 5 Once there is no more room left in J_h, if the remaining interval is of length at most ℓ (the top case), then we create one pattern interval for all the remaining interval. Otherwise (the bottom case) we create two pattern intervals, the first pattern interval of length ℓ and the second pattern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

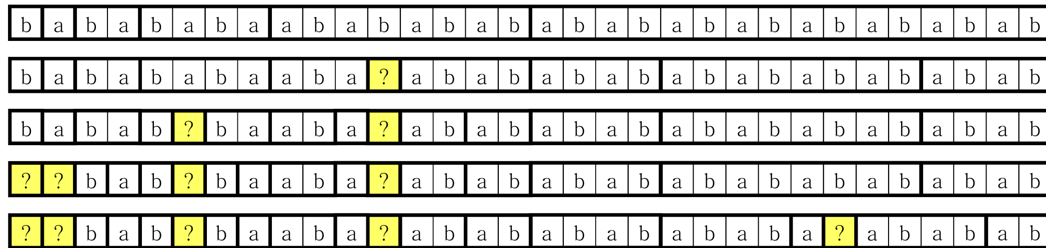

Figure 6 Example of patterns and their intervals in the secondary partitioning. Each bold rectangle corresponds to an interval in the partition.

5 The Candidate-fingerprint-queue

The algorithm of Theorem 2 is obtained via an implementation of the candidate-queues that uses $O(d \log m)$ 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 any explicit information per candidate, and in particular we cannot explicitly store fingerprints to quickly test if a candidate is still alive. On the other hand, we are still interested in using fingerprints in order to perform these tests.

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 alive. Thus, we extend the definition of a candidate-queue to a *candidate-fingerprint-queue* as follows.

▶ **Definition 9.** Let α be the index of the last text character that has arrived. Then a candidate-fingerprint-queue on an interval $I = [i, j]$ supports the following operations.

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

The implementation of the queue assumes that we use a partitioning that has the properties stated in Lemma 8. Let $I = [i, j]$ be a pattern interval in the partitioning and let c be a candidate which is maintained in the queue Q_I associated with I. For candidate c, the *entrance prefix* is the substring $t_c \dots t_{c+i-1}$, the *entrance interval* is $[c, c+i-1]$, and the *entrance fingerprint* is $\phi(t_c \dots t_{c+i-1})$. Since c was alive at the time it was inserted into Q_I, 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 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 this 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.

Entrance prefixes and arithmetic progressions. A key component of the queue data structure is Lemma 10. This lemma defines for each interval $I = [i, j] \in \mathcal{I}$ at most one unique entrance prefix u_I that is the only string that can be the entrance prefix of more than two candidates in Q_I at the same time. That is, the existence of an entrance prefix u_I and determination u_I if it exists **depends only** on the prefix pattern $p_0 \dots p_{i-1}$, *regardless* of the characters in the text. If I has such a string u_I we say that I is an *arithmetic interval*, since, as we prove in Lemma 11 the candidates in Q_I that have entrance prefix u_I form an arithmetic progression.

▶ **Lemma 10.** *For a pattern interval $I = [i, j]$ with queue Q_I, there exists at most one string u_I such that if there are more than two candidates in Q_I with the same entrance prefix then this entrance prefix must be u_I.*

Proof Sketch. By Lemma 8 there is a string of length $|I|$ containing only non-wildcard characters that is a substring of $p_0 \dots p_{i-1}$. Let v be this string. Notice that v must appear in the entrance prefix of every candidate in Q_I. If there are three candidates in Q_I then they must all appear in the text within a range of size $j - i + 1$, which is close enough to guarantee that v must be periodic.

Now, consider three candidates $c_4 < c_5 < c_6$ in Q_I that have the same entrance prefix u. Since c_4, c_5, and c_6 are all occurrences of u then $c_6 - c_5$ and $c_5 - c_4$ are period lengths of u. Thus, $\rho_u \le \min\{c_5 - c_4, c_6 - c_5\} \le \frac{c_6 - c_4}{2} \le \frac{j-i}{2} < \frac{|I|}{2} = \frac{|v|}{2}$. Therefore, by Lemma 4 $\rho_u = \rho_v$. Combined with the fact that u contains v at a particular location (since u is an entrance prefix), and v is longer than ρ_u (since $\rho_u = \rho_v$), the string u must be uniquely defined. ◀

▶ **Lemma 11.** *Let $I = [i, j]$ be an arithmetic interval. If there are $h \ge 3$ candidates $c_1 < c_2 < \dots < c_h$ in Q_I that have u_I as their entrance prefix, then the sequence $c_1, c_2, \dots, c_h$ forms an arithmetic progression whose difference is ρ_{u_I}.*

Implementation details. For an interval I, we use a linked list $\mathcal{L}_{Q_I}$ to store all of the candidates in Q_I together with their SI, except for when I is an arithmetic interval in which case candidates whose entrance fingerprint is $\phi(u_I)$ are not stored in $\mathcal{L}_{Q_I}$. Adding and

removing a candidate that belongs in $\mathcal{L}_{Q_I}$ together with its SI is straightforward. If I is an arithmetic interval, the candidates in Q_I whose entrance fingerprint is $\phi(u_I)$ are stored using a separate data structure that leverages Lemma 11. 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 candidates with entrance fingerprint $\phi(u_I)$.

▶ **Lemma 12.** *There exists an implementation of candidate-fingerprint-queues so that for any arithmetic interval I, the queue Q_I maintains all the candidates with entrance fingerprint $\phi(u_I)$ and their SI using $O(1)$ words of space.*

Space usage. The space usage of all of the queues has two components. The first component is the lists $\mathcal{L}_{Q_I}$ for all the intervals I. The second component is the data structure for storing the candidates that create the arithmetic progression of an arithmetic interval, for each such arithmetic interval. By Lemma 10 there is at most one such progression per arithmetic interval, and so all of these arithmetic progressions use $O(d + \log m)$ space. In the following lemma we prove that the total space usage of all of the lists is $O(d \log m)$.

▶ **Lemma 13.** $\sum_{I\in\mathcal{I}} |\mathcal{L}_{Q_I}| = O(d \log m)$.

Proof. By Lemma 8, we know that $|\{\mu(0), \dots, \mu(k)\}| = O(\log m)$. For each $\ell \in \{\mu(0), \dots, \mu(k)\}$ let $\mathcal{I}_\ell$ be the sequence of all pattern intervals in $\mathcal{I}$ that are between the leftmost interval of length ℓ, inclusive, and the first of either the leftmost interval of length larger than ℓ, exclusive, or the last interval in $\mathcal{I}$, inclusive. Notice that $|I| \leq \ell$ for each $I \in \mathcal{I}_\ell$. Let $\mathcal{D}_\ell$ be the set of queues for pattern intervals in $\mathcal{I}_\ell$. We show that $\sum_{I\in\mathcal{I}_\ell} |\mathcal{L}_{Q_I}| = O(|\mathcal{I}_\ell| + d)$. This implies that:

$$\sum_{I\in\mathcal{I}} |\mathcal{L}_{Q_I}| = \sum_{\ell\in\{\mu(0),\dots,\mu(k)\}} \sum_{I\in\mathcal{I}_\ell} |\mathcal{L}_{Q_I}| = \sum_{\ell\in\{\mu(0),\dots,\mu(k)\}} O(|\mathcal{I}_\ell| + d) = O(d \log m).$$

We focus on queues for which $|\mathcal{L}_{Q_I}| \geq 3$, since otherwise the bound is immediate. Notice that this includes all queues for wildcard intervals. Set $\ell \in \{\mu(0), \dots, \mu(k)\}$ and let α be the index of the last text character that has arrived.

We now establish that there exists a periodic string v of length ℓ that contains no wildcards, such that for any candidate c in any of the queues of $\mathcal{D}_\ell$ the entrance prefix of c contains v. Let $[i, j]$ be the leftmost interval in $\mathcal{I}_\ell$. Notice that $j = i + \ell - 1$ by the definition of $\mathcal{I}_\ell$. By Lemma 8, there is a string of length ℓ containing only non-wildcard characters that is a substring $p_0 \dots p_{i-1}$. Let r be the starting location of this string, and let $v = p_r \dots p_{r+\ell-1}$ be this string.

For each queue $Q_{I'} \in \mathcal{D}_\ell$ where $I' = [i', j']$ and for each candidate $c' \in Q_{I'}$, the entrance prefix of c' matches the prefix of P of length i'. Since $i' \geq i$ this means that $t_{c'+r} \dots t_{c'+r+\ell-1} = v$. The text substring $t_{\alpha-j'+r+1} \dots t_{\alpha-i'+r+\ell}$ contains at least three occurrences of v and its length is $|I'| + \ell \leq 2\ell$. Therefore, by Lemma 3, v is a periodic string and the distance between any two candidates in the same queue is a multiple of ρ_v. Notice that for any $Q_{I'} \in \mathcal{D}_\ell$ that contains at least three candidates $c_1 < c_2 < c_3$, we bound $\rho_v \leq \min\{c_3 - c_2, c_2 - c_1\} \leq \frac{c_3-c_1}{2} \leq \frac{j'-i'}{2}$.

Let $\hat{c}$ be the rightmost (largest index) candidate maintained in the queues of $\mathcal{D}_\ell$. In particular $t_{\hat{c}+r} \dots t_{\hat{c}+r+\ell-1} = v$. 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. Let the resulting substring be $t_{x+1} \dots t_{y-1}$. Unless, $x = -1$, the index x is called the *left violation* of v. Similarly, unless $y = \alpha+1$, the index y is called the *right violation* of v. Notice that $x < \hat{c}+r \leq \hat{c}+r+\ell-1 < y$.

For the following, let the entrance interval of a candidate c be $[c, e_c]$.

▶ **Claim 14.** *If $Q_{[i',j']} \in \mathcal{D}_\ell$ contains at least three candidates, then for each candidate c in $\mathcal{L}_{Q_{[i',j']}}$ either $x \in [c, e_c]$ or $y \in [c, e_c]$.*

Proof Sketch. There exists a text index $\beta = \hat{c} + r$, such that β appears in the entrance interval of any candidate in the queues of $\mathcal{D}_\ell$, and $x < \beta < y$. Therefore, for each candidate $c \in \mathcal{L}_{Q_{[i',j']}}$, if $x \notin [c, e_c]$ and $y \notin [c, e_c]$ it must be the case that $[c, e_c] \subseteq [x+1, y-1]$. Recall that the principal period length of $t_{x+1} \dots t_{y-1}$ is ρ_v. Since $u = t_c \dots t_{e_c}$ is a substring of $t_{x+1} \dots t_{y-1}$, it must be that $\rho_u \leq \rho_v$. Hence, one can prove that $[i', j']$ is an arithmetic interval with $u_{[i',j']} = u$, contradicting the fact that c is stored in $\mathcal{L}_{Q_{[i',j']}}$. ◀

Let $\mathcal{L}^x_{Q_I}$ ($\mathcal{L}^y_{Q_I}$) be the sets of candidates that are stored in $\mathcal{L}_{Q_I}$ such that their entrance interval contains x (y). $\mathcal{L}^x_{Q_I}$ and $\mathcal{L}^y_{Q_I}$ are not necessarily disjoint. Notice that by Claim 14, $\mathcal{L}^x_{Q_I} \cup \mathcal{L}^y_{Q_I}$ contains all the candidates stored in $\mathcal{L}_{Q_I}$.

▶ **Claim 15.** $\sum_{I \in \mathcal{I}_\ell} |\mathcal{L}^x_{Q_I}| = O(|\mathcal{I}_\ell| + d)$ *and* $\sum_{I \in \mathcal{I}_\ell} |\mathcal{L}^y_{Q_I}| = O(|\mathcal{I}_\ell| + d)$.

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

Notice that the contribution to $\sum_{I \in \mathcal{I}_\ell} |\mathcal{L}^x_{Q_I}|$ from all sets $\mathcal{L}^x_{Q_I}$ that have less than two candidates is at most $O(|\mathcal{I}_\ell|)$. Thus, for the following we assume that $\mathcal{L}^x_{Q_I}$ contains at least two candidates. Let $c_{I,x} = \max \mathcal{L}^x_{Q_I}$ be the most recent candidate in $\mathcal{L}^x_{Q_I}$. Let $c < c_{I,x}$ be a candidate in $\mathcal{L}^x_{Q_I}$. Since $c \in \mathcal{L}^x_{Q_I}$ we have that $p_{x-c} \approx t_{c+x-c} = t_x$ (recall that both x and c are indices in the text). Similarly, since $c_{I,x} \in \mathcal{L}^x_{Q_I}$ we have that $p_{x-c} \approx t_{c_{I,x}+x-c} = t_{x+(c_{I,x}-c)}$. Recall that the distance between any two candidates in Q_I is a multiple of ρ_v. In particular the distance $(c_{I,x} - c)$ is a multiple of ρ_v and $(c_{I,x} - c) \leq |I| \leq |v|$. Thus, $t_x \neq t_{x+(c_{I,x}-c)}$ since x violates the period of length ρ_v. Recall that $t_x \approx p_{x-c} \approx t_{x+(c_{I,x}-c)}$, and so p_{x-c} must be a wildcard. Therefore, each $c \in \mathcal{L}^x_{Q_I}$, except for possibly $c_{I,x}$, is in a position c such that p_{x-c} is a wildcard. Since x is the same for all of the candidates in all of the $\mathcal{L}^x_{Q_{I'}}$ for all $I' \in \mathcal{I}_\ell$, then the contribution to $\sum_{I \in \mathcal{I}_\ell} |\mathcal{L}^x_{Q_I}|$ of the candidates that are not the most recent in their set $\mathcal{L}^x_{Q_I}$ is at most $O(d)$. The contribution of the most recent candidates is at most $O(|\mathcal{I}_\ell|)$. Thus, $\sum_{I' \in \mathcal{I}_\ell} |\mathcal{L}^x_{Q_{I'}}| = O(|\mathcal{I}_\ell| + d)$. The proof that $\sum_{I' \in \mathcal{I}_\ell} |\mathcal{L}^y_{Q_{I'}}| = O(|\mathcal{I}_\ell| + d)$ is symmetric. ◀

Finally, $\sum_{I \in \mathcal{I}_\ell} |\mathcal{L}_{Q_I}| \leq \sum_{I \in \mathcal{I}_\ell} |\mathcal{L}^x_{Q_I}| + \sum_{I \in \mathcal{I}_\ell} |\mathcal{L}^y_{Q_I}| = O(|\mathcal{I}_\ell| + d)$. Thus, we have completed the proof of Lemma 13. ◀

6 The Tradeoff Algorithm

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 a randomized algorithm which expands this result for a parameter $0 \leq \delta \leq 1$ 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 $\pi_{P^*} \leq d^\delta$. The tradeoff algorithm first finds all the occurrences of P^* using a specialized algorithm for patterns with wildcard-period length at most d^δ. 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 $i \leq |P^*| \leq j$. Each occurrence of P^* in the text is inserted as a candidate in the algorithm of Theorem 2 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 means that the average

distance between each two consecutive candidates is at least d^δ, and so combined with a lazy approach we are able to obtain an $\tilde{O}(d^{1-\delta})$ amortized time cost per character.

In the rest of this section we describe an overview of the specialized algorithm for dealing with patterns whose wildcard-period length is at most d^δ. The rest of the details for the tradeoff algorithm appear in the full version of this paper.

6.1 Patterns with Small Wildcard-period Length

Let P be a pattern of length m with $\pi_P < d^\delta$. Let q be an integer, which for simplicity is assumed to divide m. Consider the conceptual matrix $M^q = \{m^q_{x,y}\}$ of size $\left\lceil \frac{m}{q} \right\rceil \times q$ where $m^q_{x,y} = p_{(x-1)\cdot q+y-1}$. For any integer $0 \le r < q$ the rth column corresponds to an *offset pattern* $P_{q,r} = p_r p_{r+q} p_{r+2q} \dots p_{m-q+r}$. Notice that some offset patterns might be equal. Let $\Gamma_q = \{P_{q,r} | 0 \le r < q, '?' \notin P_{q,r}\}$ be the set of all the offset patterns that do not contain any wildcards. Each unique offset pattern is associated with 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 is given a unique id. The columns of M^q define a *column pattern* P_q of length q, where the i'th character is the unique id of the i'th column, or '?' if no such id exists (the column has wildcards).

We partition T into q *offset texts*, where for every $0 \le r < q$ we define $T_{q,r} = t_r t_{r+q} t_{r+2q} \dots$. Using the dictionary matching streaming (DMS) algorithm of Clifford et al. [13] we find occurrences of offset patterns from Γ_q in each of the offset texts. Notice that we do *not* only find occurrences of $P_{q,r}$ in $T_{q,r}$ (since we cannot guarantee that the offset of T synchronizes with an occurrence of P). When the character t_α arrives, the algorithm passes t_α 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 unique id is the αth character in T_q. Otherwise, we use a dummy character for the αth character in T_q.

Notice that on 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 16.** *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 $\forall q \in Q : q = \tilde{O}(d)$.*

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 Γ_q we use the following lemma.

▶ **Lemma 17.** *If $q = \tilde{O}(d)$ and $\pi_P \le d^\delta$ then $|\Gamma_q| = O(d^\delta)$.*

Proof. Since $\pi_P \le d^\delta$, there exists a string $S = s_0 \dots s_{2m-2}$ that contains $\Omega(\frac{m}{d^\delta})$ occurrences of P. Using this string we show that $|\Gamma_q| = O(d^\delta)$.

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 $0 \le i < m$

let $S_i = s_i \dots s_{i+m-1}$. For every $0 \le r < q$ let $S_{i,q,r} = s_{i+r}s_{i+r+q} \dots s_{i+m-q+r}$, and so for any integer $0 \le \Delta < q - r$ 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 $S_{i,q,r} = P_{q,r} \neq P_{q,r'} = S_{i,q,r'}$. Thus, 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 $i + r - r_1$ is not occurrence of P. Therefore, every occurrence of P in S is associated with $|R_q| - 1$ indices that cannot be an occurrence of P. We further argue that each index i is associated with an occurrence of P (as just described) at most once. This is because if $S_{i,q,r_1} = P_{q,r}$ for some $r \in R_q$ then i can be associated only with an occurrence of P in index $i - (r - r_1)$. So the maximum number of instances of P in S is at most $\frac{|S|}{|R_q|} = \frac{2m-1}{|R_q|}$. However, S contains at least $\frac{m}{d^\delta}$ instances of P, so $\frac{m}{d^\delta} \le \frac{2m-1}{|R_q|}$ which implies that $|\Gamma_q| = |R_q| \le 2d^\delta = O(d^\delta)$. ◀

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 $O(d^\delta)$ strings (by Lemma 17). Since each instance of the DMS algorithm uses $\tilde{O}(d^\delta)$ words of space [13], the total space usage for all instances of the DMS algorithm is $\tilde{O}(d^{1+\delta})$ 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 $\tilde{O}(d^{1+\delta})$ words. Since $|Q| = O(\log d)$ the total space usage for all elements in Q is $\tilde{O}(d^{1+\delta})$ words, and the total time per arriving character is $\tilde{O}(1)$. Thus we have proven the following.

▶ **Theorem 18.** *For any $0 \le \delta \le 1$ the online d wildcards pattern matching problem can be solved for patterns P with $\pi_P < d^\delta$ with a randomized Monte Carlo algorithm, in $\tilde{O}(1)$ time per arriving text character and using $\tilde{O}(d^{1+\delta})$ words of space.*

References

1. Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.
2. Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. doi:10.1016/S0196-6774(03)00097-X.
3. Jean Berstel and Luc Boasson. Partial words and a theorem of fine and wilf. *Theor. Comput. Sci.*, 218(1):135–141, 1999. doi:10.1016/S0304-3975(98)00255-2.
4. Francine Blanchet-Sadri. *Algorithmic Combinatorics on Partial Words*. Discrete mathematics and its applications. CRC Press, 2008. URL: http://www.crcpress.com/product/isbn/9781420060928.
5. Francine Blanchet-Sadri and Robert A. Hegstrom. Partial words and a theorem of fine and wilf revisited. *Theor. Comput. Sci.*, 270(1-2):401–419, 2002. doi:10.1016/S0304-3975(00)00407-2.
6. Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.
7. Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. *Theor. Comput. Sci.*, 483:2–9, 2013. doi:10.1016/j.tcs.2012.11.040.

8. Sabin Cautis, Filippo Mignosi, Jeffrey Shallit, Ming-wei Wang, and Soroosh Yazdani. Periodicity, morphisms, and matrices. *Theor. Comput. Sci.*, 295:107–121, 2003. doi:10.1016/S0304-3975(02)00398-5.
9. Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007. doi:10.1016/j.ipl.2006.08.002.
10. Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. *Inf. Comput.*, 209(4):731–736, 2011. doi:10.1016/j.ic.2010.12.007.
11. Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. From coding theory to efficient pattern matching. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 778–784, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496855.
12. Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. Pattern matching with don't cares and few errors. *J. Comput. Syst. Sci.*, 76(2):115–124, 2010. doi:10.1016/j.jcss.2009.06.002.
13. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In Nikhil Bansal and Irene Finocchi, editors, *Proc. 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *LNCS*, pages 361–372. Springer, 2015. doi:10.1007/978-3-662-48350-3_31.
14. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. The k-mismatch problem revisited. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 2039–2052, 2016. doi:10.1137/1.9781611974331.ch142.
15. Raphaël Clifford, Markus Jalsenius, Ely Porat, and Benjamin Sach. Space lower bounds for online pattern matching. *Theor. Comput. Sci.*, 483:68–74, 2013. doi:10.1016/j.tcs.2012.06.012.
16. Raphaël Clifford and Ely Porat. A filtering algorithm for k-mismatch with don't cares. In *String Processing and Information Retrieval, 14th International Symposium, SPIRE 2007, Santiago, Chile, October 29-31, 2007, Proceedings*, pages 130–136, 2007. doi:10.1007/978-3-540-75530-2_12.
17. Raphaël Clifford and Benjamin Sach. Pseudo-realtime pattern matching: Closing the gap. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, pages 101–111. Springer, 2010. doi:10.1007/978-3-642-13509-5_10.
18. Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 592–601. ACM, 2002. doi:10.1145/509907.509992.
19. Funda Ergün, Hossein Jowhari, and Mert Saglam. Periodicity in streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 13th International Workshop, APPROX 2010, and 14th International Workshop, RANDOM 2010, Barcelona, Spain, September 1-3, 2010. Proceedings*, pages 545–559, 2010. doi:10.1007/978-3-642-15369-3_41.
20. Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.
21. Michael J Fischer and Michael S Paterson. String-matching and other products. Technical report, DTIC Document, 1974.
22. Zvi Galil and Joel I. Seiferas. Time-space-optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983. doi:10.1016/0022-0000(83)90002-8.

23 Pawel Gawrychowski. Optimal pattern matching in LZW compressed strings. *ACM Transactions on Algorithms*, 9(3):25, 2013. doi:10.1145/2483699.2483705.

24 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Streaming pattern matching with d wildcards. *CoRR*, abs/1605.16729, 2015. URL: http://arxiv.org/abs/1605.16729.

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

26 Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *39th Annual Symposium on Foundations of Computer Science, FOCS'98, November 8-11, 1998, Palo Alto, California, USA*, pages 166–173. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743440.

27 Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 – March 2, 2013, Kiel, Germany*, pages 400–411, 2013. doi:10.4230/LIPIcs.STACS.2013.400.

28 Adam Kalai. Efficient pattern-matching with don't cares. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 655–656. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545468.

29 Daniel M. Kane, Jelani Nelson, Ely Porat, and David P. Woodruff. Fast moment estimation in data streams in optimal space. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 745–754, 2011. doi:10.1145/1993636.1993735.

30 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

31 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

32 Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986. doi:10.1016/0304-3975(86)90178-7.

33 Lap-Kei Lee, Moshe Lewenstein, and Qin Zhang. Parikh matching in the streaming model. In *String Processing and Information Retrieval – 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, pages 336–341, 2012. doi:10.1007/978-3-642-34109-0_35.

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

35 S. Muthukrishnan and H. Ramesh. String matching under a general matching relation. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*, volume 652 of *Lecture Notes in Computer Science*, pages 356–367. Springer, 1992. doi:10.1007/3-540-56287-7_118.

36 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *50th Annual IEEE Symp. on Foundations of Computer Science, FOCS 2009, Oct. 25-27, 2009, Atlanta, Georgia, USA*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

37 Ely Porat and Ohad Lipsky. Improved sketching of hamming distance with error correcting. In *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings*, pages 173–182, 2007. doi:10.1007/978-3-540-73437-6_19.

38 William F. Smyth and Shu Wang. A new approach to the periodicity lemma on strings with holes. *Theor. Comput. Sci.*, 410(43):4295–4302, 2009. doi:10.1016/j.tcs.2009.07.010.