Towards Optimal Approximate Streaming Pattern Matching by Matching Multiple Patterns in Multiple Streams[*]

Shay Golan
Bar Ilan University, Ramat Gan, Israel
golansh1@cs.biu.ac.il

Tsvi Kopelowitz
Bar Ilan University, Ramat Gan, Israel
kopelot@gmail.com

Ely Porat
Bar Ilan University, Ramat Gan, Israel
poratelly@cs.biu.ac.il

Abstract

Recently, there has been a growing focus in solving approximate pattern matching problems in the streaming model. Of particular interest are the pattern matching with k-mismatches (KMM) problem and the pattern matching with w-wildcards (PMWC) problem. Motivated by reductions from these problems in the streaming model to the *dictionary matching problem*, this paper focuses on designing algorithms for the dictionary matching problem in the *multi-stream model* where there are several independent streams of data (as opposed to just one in the streaming model), and the memory complexity of an algorithm is expressed using two quantities: (1) a read-only *shared memory* storage area which is shared among all the streams, and (2) local *stream memory* that each stream stores separately.

In the dictionary matching problem in the multi-stream model the goal is to preprocess a dictionary $D = \{P_1, P_2, \ldots, P_d\}$ of $d = |D|$ patterns (strings with maximum length m over alphabet Σ) into a data structure stored in shared memory, so that given multiple independent streaming texts (where characters arrive one at a time) the algorithm reports occurrences of patterns from D in each one of the texts as soon as they appear.

We design two efficient algorithms for the dictionary matching problem in the multi-stream model. The first algorithm works when all the patterns in D have the same length m and costs $O(d \log m)$ words in shared memory, $O(\log m \log d)$ words in stream memory, and $O(\log m)$ time per character. The second algorithm works for general D, but the time cost per character becomes $O(\log m + \log d \log \log d)$. We also demonstrate the usefulness of our first algorithm in solving both the KMM problem and PMWC problem in the streaming model. In particular, we obtain the first almost optimal (up to poly-log factors) algorithm for the PMWC problem in the streaming model. We also design a new algorithm for the KMM problem in the streaming model that, up to poly-log factors, has the same bounds as the most recent results that use different techniques. Moreover, for most inputs, our algorithm for KMM is significantly faster on average.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Streaming approximate pattern matching, Dictionary matching

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.65

[*] This work is supported in part by ISF grant 1278/16. 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).

1 Introduction

In the popular *streaming* model [2, 48] the input is given as a sequence of elements (the data stream) that may be scanned only once, the storage space is limited, and the amount of time spent on each element needs to be minimized. In many problems there is also a preprocessing phase involved. For example, in the basic streaming pattern matching problem, the goal is to find occurrences of a given pattern (to be preprocessed) of size m in the data stream [52, 18]. The preprocessing phase receives the pattern and creates a sub-linear sized data structure that is used to locate the pattern in streaming input. Following the breakthrough result of Porat and Porat [52], there has recently been a rising interest in solving pattern matching problems in the streaming model [18, 30, 47, 19, 42, 23, 24, 37, 38].

Approximate Streaming Pattern Matching. While Porat and Porat [52] and Breslauer and Galil [18] addressed the exact match case, which is the purest form of streaming pattern matching, several papers have focused on *approximate* versions in the streaming model. The term *approximate pattern matching* refers to any pattern matching problem that is not exact matching. Examples include pattern matching with up to k-mismatches [46, 53, 11, 22, 21, 27, 24, 26, 28], pattern matching with w-wildcards [33, 49, 41, 43, 29, 20, 37], pattern matching with up to e-edits [56], parameterized pattern matching [6, 12, 42, 15, 16, 17, 39], function pattern matching [13, 4], swapped matching [10, 3, 5] and many more.

Remarkably, recent results in the streaming model for both the pattern matching with k-mismatches (KMM) problem [24] and the pattern matching with w-wildcards (PMWC) problem [37] (both formally defined below) use a similar approach which, in particular, reduce the approximate pattern matching problem in the streaming model that is being solved to the *dictionary matching* (DM) problem in the streaming model. In the dictionary matching problem ([23, 31, 8, 9, 45, 34, 14, 35, 32, 7, 38]) the goal is to preprocess a *dictionary* $D = \{P_1, P_2, \ldots, P_d\}$ of $d = |D|$ patterns (strings over alphabet Σ) so that given a text T we quickly report all of the occurrences of patterns from D in T. In the streaming model [23, 38] the text T arrives online, one character at a time, and the goal is to report, for each arriving character, the id of the longest pattern ending at this character[1]. Moreover, a pattern must be reported as soon as it appears.

The motivation for this paper is due to realizing that the reductions to the dictionary matching problem mentioned above all suffer from an inefficiency that is due reducing a single stream approximate pattern matching problem to several instances of the dictionary matching problem, where all of the instances use the same dictionary but have different input texts. The results in [24] and [37] use a separate block of space for each instance of the dictionary matching, even though the dictionaries are the same. If it would be possible to share the space usage representing the dictionary among all of the instances then that would imply an immediate improvement in the total space usage. More formally, we introduce the *dictionary matching in the multi-stream model* that captures this challenge.

Dictionary matching in the multi-stream model. In the dictionary matching in the multi-stream (DMMS) problem the input is a dictionary D to be processed, there are s independent input streams of text $T_1, T_2, \ldots, T_s$, and the goal is to report all of the occurrences of

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

patterns from D in any T_i, for $1 \leq i \leq s$, as soon as the occurrence arrives. An algorithm for the DMMS problem is allowed to set up a read-only block of *shared memory* during a preprocessing phase, whose contents depend solely on D, and s blocks of *stream memory*, one for each text stream, to be used privately for each text stream as the text is being processed. Notice that it is enough to describe an algorithm that works on one stream, as long as the description details which data is stored in each type of memory. Also notice that a naïve algorithm would be to use a separate solution for dictionary matching in one stream for each one of the text streams, where each instance is stored completely in stream memory and there is no use of the shared memory. The most efficient algorithm for dictionary matching in the streaming model is due to Golan and Porat [38] using $O(d \log m)$ words and the time per character is $O(\log \log |\Sigma|)$, which could be as large as $O(\log \log(m \cdot d))$. All of these complexities are in the worst-case, and their algorithm is correct with high probability. With the naïve method, the algorithm of [38] implies a solution for DMMS that uses a total of $O(s \cdot d \log m)$ words. This space complexity is inherent in the algorithm of [38] since, in particular, their algorithm always stores the last $\Theta(d \log m)$ text characters in each stream, which does not benefit from shared memory. Thus, algorithms are only of interest if they can beat this naïve method.

1.1 Our Results

We introduce a new algorithm for the dictionary matching problem in the multi-stream model, which is summarized in the following theorem.

▶ **Theorem 1.** *There exists an algorithm for the multi-stream dictionary matching problem where each pattern has length m that uses $O(d \log m)$ words of shared memory, $O(\log m \log d)$ words of stream memory, and $O(\log m)$ time per character. All of these complexities are in the worst-case, and the algorithm is correct with high probability.*

Notice that we focus on the case where all of the patterns in D have the same length m, since this case suffices for our applications. Due to space limitations, the following extension of the results to different length patterns is left for the full version.

▶ **Theorem 2.** *There exists an algorithm for the multi-stream dictionary matching problem that uses $O(d \log m)$ words of shared memory, $O(\log m \log d)$ words of stream memory, and $O(\log m + \log d \log \log d)$ time per character, where $m = m_D$ is the length of the longest pattern in D. All of these complexities are in the worst-case, and the algorithm is correct with high probability.*

Thus, if there are s streams of data, the total space usage becomes $O(d \log m + s \log m \log d)$ words, which is substantially less than the total space usage of the naïve method. By using the algorithm of Theorem 1 we are able to reduce the space usage for solving several approximate streaming pattern matching problems, as we discuss next.

Streaming pattern matching with w wildcards. A *wildcard* character, denoted by $'?' \notin \Sigma$, is a special character that matches every character in Σ. In the streaming PMWC problem the goal is to preprocess a pattern $P[1..m]$ that contains w wildcard characters, so that given a streaming text T the algorithm reports, for each arriving character, whether the current text suffix of length m matches the pattern. The most efficient known algorithm for the PMWC problem was given in [37] where we introduced a trade-off algorithm, which for every $0 \leq \delta \leq 1$ uses $\tilde{O}(w^{1-\delta})$ amortized time per character and $\tilde{O}(w^{1+\delta})$ words of space, where $\tilde{O}$ hides poly-logarithmic factors. Our results use a reduction from the PMWC problem to the DMMS problem. Using Theorem 1 we are able to obtain the following result which is optimal, up to *polylog*(m, w) factors.

▶ **Theorem 3.** *There exists a randomized Monte Carlo algorithm for the pattern matching with w-wildcards problem in the streaming model that succeeds with high probability, uses $\tilde{O}(w)$ words of space and spends $\tilde{O}(1)$ time per arriving text character. Moreover, any algorithm which solves the streaming pattern matching with w wildcards with high probability must use $\Omega(w)$ bits of space.*

The proof of the upper bound for Theorem 3 is obtained by directly plugging the algorithm from Theorem 1 into the reduction of [37]. The lower bound is based on a straightforward reduction from the communication complexity INDEXING problem the proof of Theorem 3 is left for the full version.

Streaming pattern matching with k-mismatches. Another application is the KMM problem in the streaming model. In this problem the goal is to preprocess a pattern $P[1..m]$ so that given a streaming text T the algorithm reports for each arriving character whether the number of mismatches between P and the current text suffix of length m is at most k, and if so then the algorithm also reports the number of mismatches. The most efficient algorithm currently published for this problem is by Clifford et al. [24]. This algorithm uses $O(k^2 \operatorname{polylog} m)$ words of space and takes $O(\sqrt{k}\log k + \operatorname{polylog} m)$ time per character. Their results use a reduction from the streaming KMM problem to the DMMS problem. Using Theorem 1 we are able to obtain the following result.

▶ **Theorem 4.** *There exists a randomized Monte Carlo algorithm for the streaming k-mismatch problem that succeeds with high probability, uses $\tilde{O}(k)$ words of space and spends $\tilde{O}(k)$ time per arriving text character.*

A proof of Theorem 4 is obtained by plugging the algorithm from Theorem 1 into the reduction of [24] (with some minor details) which uses group testing techniques [54, 53, 51, 50, 36]. The space usage of this algorithm is optimal up to $\operatorname{polylog} m$ factors [40]. Clifford, Kociumaka and Porat [26] recently posted another algorithm that obtains the same complexities (ignoring poly-logarithmic factors) but using different techniques. Nevertheless, we provide a second stronger result.

k-mismatches and periodicity. Clifford et al. [24] introduced the notion of x-period which captures the generalization of periodicity to work with a bounded number of mismatches. The number of mismatches between two equal length strings S and S' is known as the Hamming Distance and is denoted by $\mathsf{Ham}(S, S')$. The x-period of a string P of length m is the smallest integer $\pi > 0$ such that $\mathsf{Ham}(P[1+\pi..m], P[1..m-\pi]) \leq x$. Notice that for small x, most strings have a high x-period.

▶ **Theorem 5.** *There exists a randomized Monte Carlo algorithm for the streaming k-mismatch problem that succeeds with high probability, uses $\tilde{O}(k)$ words of space and spends $\tilde{O}(k)$ time per arriving text character. Moreover, if the $4k$-period of P is $\Omega(k)$, then the algorithm spends an average of $\tilde{O}(1)$ time per character.*

Since the typical assumption is that k is fairly small, the algorithm of Theorem 5 spends an average of $\tilde{O}(1)$ time per character for most patterns. Notice that Theorem 5 immediately implies Theorem 4. Due to space considerations, the proof of Theorem 5 is left for the full version.

Organization. In the rest of this paper we give an overview focusing on intuition and the general ideas of how to prove Theorem 1. The missing proofs and details are left for the full version.

1.2 Related Work

As mentioned above, the current most efficient algorithm for DM in the streaming model is due to Golan and Porat [38] where the space usage is $O(d \log m)$ words and the time per character in T is $O(\log\log|\Sigma|)$. Another relevant result is that of Clifford et al. [25], which deals with pattern matching (one pattern) in multiple streams, but not in the classic streaming model (since the space usage is not sublinear). They show how for s streams and a pattern of size m, one can report occurrences of the pattern in all of the streams, concurrently, using $O(m+s)$ words of space.

We emphasize that the Aho-Corasick automata [1] is not a multi-stream solution for the dictionary matching problem since the shared memory usage is not sub-linear. However, the stream memory usage is $O(1)$ words.

2 Preliminaries

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

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

Fingerprints For a natural number n we denote $[n] = \{1, 2, \dots, n\}$. For the following let $u, v \in \bigcup_{i=0}^{n} \Sigma^i$ be two strings of size at most n. Porat and Porat [52] and Breslauer and Galil [18] extended the fingerprint method of Karp and Rabin [44], 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 $u \ne v$ then $\phi(u) \ne \phi(v)$ with high probability (at least $1 - \frac{1}{n^{c-1}}$).
2. *The sliding property:* Let w=uv be the concatenation of u and v. If $|w| \le n$ then given the length and the fingerprints of any two strings from u,v and w, one can compute the fingerprint of the third string in constant time.

For two strings u and v and a fingerprint function ϕ, we say that the *fingerprint concatenation* of $\phi(u)$ and $\phi(v)$ is $\phi(uv)$. If we are given the lengths of u and v then computing the fingerprint concatenation takes constant time due to the sliding property.

Remark. Our algorithm often uses fingerprints in order to quickly test if two strings are equal or not. To ease presentation, in the rest of the paper we assume that fingerprints never give false positives. This assumption is acceptable since the algorithm is allowed to fail with small probability.

3 Same Length Patterns – Proof of Theorem 1

Throughout the paper, let q denote the current index of the last character in T. The algorithm initially considers every text location as a *candidate* for an occurrence of a pattern. Conceptually, a text location c is considered to be a candidate until the algorithm encounters proof that there cannot be any pattern in D that appears at location c. This leads to a naïve solution which stores all of the candidates, and each time a new character arrives the candidates are tested to see if they are still candidates. Regardless of the method of testing, this solution is too expensive, in terms of both space and time, since the number of candidates could be $\Omega(m)$.

In order to reduce the time cost per character, we borrow a technique introduced in [52] which considers prefixes of the dictionary strings of exponentially growing length. This technique has been extensively used for streaming pattern matching algorithms [30, 18, 23, 24, 37, 38, 42, 26, 55], but in our case the details are more delicate than usual. Our algorithm makes use of an increasing sequence of $O(\log m)$ shift values $\Delta = (\delta_0, \delta_1, \ldots, \delta_{|\Delta|-1})$ where

$$\delta_k = \begin{cases} 25 & \text{if } k = 0 \\ \min(5\lfloor \frac{6}{25}\delta_{k-1} \rfloor, m) & \text{otherwise.} \end{cases}$$

Notice that for $1 \leq k < |\Delta| - 1$ we have $\delta_k - \delta_{k-1} \leq \delta_k/5$. Denote $\ell_k = \delta_k/5$. For each $0 \leq k \leq| \Delta| - 1$ let $D_k = \{P[1..\delta_k] | P \in D\}$ be the set of prefixes of patterns from D of length δ_k. Let $F_k = \{\phi(P) | P \in D_k\}$ be the set of fingerprints of patterns in D_k.

The intuition behind the sets D_k for $\delta_k \in \Delta$ is that our algorithm first finds occurrences of patterns from D_{k-1}, and those occurrences are then used for finding occurrences of patterns from D_k. Notice that D_0 contains only constant sized prefixes of patterns (of length 25). It is straightforward to find occurrences of these prefixes using $O(d)$ words in shared memory, $O(1)$ words in stream memory and $O(1)$ time per text character. Also notice that $D_{|\Delta|-1} = D$, and so once a pattern from $D_{|\Delta|-1}$ is found, it is reported immediately. Most of the technical work is on patterns in D_k for $1 \leq k < |\Delta| - 1$. Thus, the rest of the discussion primarily focuses on $\delta_1, \delta_2, \ldots, \delta_{|\Delta|-2}$.

Testing candidates. Testing whether a candidate c is still a candidate takes place only when $q = c + \delta_k - 1$ for some $\delta_k \in \Delta$ (so there are only $O(\log m)$ tests per each arrival of a text character). At this point in time, the suffix of T starting at location c is of length δ_k. Notice that for c to end up being an occurrence of some pattern it must be that $T[c..q] \in D_k$. The *text fingerprint* $\phi(T[1..q])$ is the fingerprint of the text up to the last character that has arrived, and is maintained with $O(1)$ space and in $O(1)$ time per character. The *candidate fingerprint* $\phi(T[1..c-1])$ is the fingerprint of the text prefix up to location c. With access to the candidate fingerprint (which we describe below) and the text fingerprint, the algorithm uses the sliding property to compute in constant time the fingerprint $\phi(T[c..q])$, and then tests whether $\phi(T[c..q]) \in F_k$ (thereby testing whether $T[c..q] \in D_k$) via a static hash table.

One almost trivial way of providing access to the candidate fingerprints is to store (in local stream memory) the candidates via a linked list together with some additional $O(1)$ information per candidate to help compute the candidate fingerprint. Unfortunately, the number of candidates could be as large as $\Omega(m)$, and so we cannot afford to store explicit information for each candidate. Instead, we devise a new method for implicitly storing the candidates so that whenever a new text character arrives we can quickly infer which candidates need to be tested (if any), and then quickly extract the candidate fingerprints of the tested candidates. This task is accomplished with the aid of *guiding graphs*.

The guiding graph. For each D_k the algorithm stores a directed edge-weighted graph G_k in shared memory, called a *guiding graph*. In order to simplify the presentation, we focus on a simplified version of the guiding graph. A *pseudo-forest* is an undirected graph where each connected component contains at most one cycle. G_k is a directed pseudo-forest in which the out-degree of each vertex is at most 1. Each edge e in G_k has a weight $w(e)$ and label $\lambda(e)$ such that there exists a non-empty *edge string* S_e where $\lambda(e) = \phi(S_e)$ and $w(e) = |S_e| \geq 1$. For each $P \in D_k$ there is an associated vertex $v_P \in G_k$. The total size of G_k is $O(|D_k|) = O(d)$.

The algorithmic usefulness of G_k is due to a *directed path* (DP) property which captures the combinatorial guarantees given by the guiding graph. We first state a stronger version of the DP property, which unfortunately we do not know how to guarantee as stated. In Section 4 we give a weaker version, which we do know how to guarantee, but the weaker version introduces an extra $O(\log d)$ factor in stream memory.

▶ **Property 1** (Strong Directed Path Property). *Let S be a string where $\delta_k \leq |S| < \delta_{k+1}$, such that the prefix and suffix of S of length δ_k are $P_b, P_e \in D_k$, respectively. Then there exists a single directed path π in G_k from v_b to v_e (which may contain cycles) such that the concatenation of the edge strings for the sequence of edges on π is exactly $S[1..|S| - \delta_k]$, which is the prefix of S until the occurrence of the suffix P_e.*

If $P \in D_k$ is a substring *of S at location h, then the path starting from v_b with total edge weight $h - 1$ must exist and end at v_P, so $v_P \in \pi$*[2]*. Moreover, for any prefix of π, with total edge weight w, the concatenation of the edge strings on this path prefix is $S[1..w]$.*

The intuition behind the usage of the guiding graph is that the guiding graph cleverly represents all possible linked lists of candidates in the *text interval* $I_k = (q-\delta_{k+1}+1, q-\delta_k+1]$ that can ever be encountered by the algorithm. For a location c in I_k, let the *entrance prefix* of c be $T[c..c + \delta_k - 1]$. Notice that c is a candidate if and only if the entrance prefix of c is some pattern $P \in D_k$. For a candidate c with entrance prefix $P \in D_k$, we denote $v_c = v_P$. Thus, all of the candidates in I_k are the candidates c for which the last verification took place when the character $T[c + \delta_k - 1]$ arrived, and so the entrance prefixes of candidates in I_k are patterns from D_k. Let L be the list of candidates in I_k. Let c_b (c_e) be the first (last) candidate in L, and let $P_b \in D_k$ ($P_e \in D_k$) be the string of length δ_k occurring at location c_b (c_e) in T. Both c_b and c_e are in I_k, and so $c_e - c_b < \delta_{k+1} - \delta_k$. Since $|P_b| = |P_e| = \delta_k$, the substring $S = T[c_b..c_e + |P_e| - 1]$ has P_b and P_e as its prefix and suffix, respectively, and $\delta_k \leq |S| < \delta_{k+1}$. Thus, by the strong DP property, there exists a path π_L in G_k from v_{c_b} to v_{c_e}, and the concatenation of edge strings on π_L is exactly $S[1..|S| - \delta_k] = T[c_b..c_e - 1]$. Moreover, for any candidate c in L we have $v_c \in \pi_L$ and the concatenation of the edge strings on a prefix π_c of π_L from v_{c_b} to v_c is exactly $S[1..c - c_b] = T[c_b..c - 1]$. Being that $T[1..c - 1]$ is the concatenation of $T[1..c_b - 1]$ and $T[c_b..c - 1]$, the candidate fingerprint of c is derivable from the candidate fingerprint of c_b and concatenation of the edge strings on π_c. Thus, if G_k is stored in shared memory, then we are able to recover the candidate fingerprints of all of the candidates in L by storing, in stream memory, a pointer to the beginning and of π_L together with locations c_b, c_e[3], and the fingerprint candidate of c_b. In some sense this feature allows us to access information about the history of the text stream, although this information is not stored explicitly.

Phantom Candidates. While the strong DP property guarantees that every candidate c in L has a corresponding vertex in π_L, the property does not guarantee that every vertex in π_L corresponds to a candidate in L. In particular, let $\pi_L = (v_1, v_2, \ldots, v_x)$ and notice that π_L may contain duplicate vertices (since π_L may contain a cycle). By the strong DP property, if the ordered list of candidates in L is $(c_1, c_2 \ldots, c_y)$ then there exist indices $1 = i_1 < i_2 < \cdots < i_y = x$ such that for all $1 \leq j \leq y$, v_{i_j} corresponds to c_j. However, for

[2] Notice that if π contains a cycle, then there may be several prefixes of π ending at v_P but only one of them can have a specific weight.

[3] The reason for storing c_e is in order to know where π_L ends. This is particularly important when G_k contains a cycle.

every $1 \leq z \leq x$ such that for every $1 \leq j \leq y$, $z \neq i_j$, we have that v_z at location z in the list[4] does not correspond to a candidate in L. This means that the implicit representation of L through π_L may contain irrelevant information.

To overcome this issue, we allow the vertices on π_L that do not correspond to candidates to be considered *as if* they are candidates, which we call *phantom* candidates. A phantom candidate is a text location that failed a test in the past, but now is implicitly considered again because its corresponding vertex lies on a directed path that is implicitly stored via the path's endpoints. A crucial aspect of phantom candidates is that they do *not* affect the complexity bounds or correctness of our algorithm. This will be made clear in the complexity analysis. If c is a phantom candidate due to vertex $v_i \in \pi_L$, we say that $v_c = v_i$.

Updating π_L. If L is empty and a new candidate c enters I_k, then π_L becomes the single vertex v_c. If L is not empty, let c_b and c_e be the first and last candidates in L, respectively. At this time it is possible that c_b is a phantom candidate, but we guarantee that c_e is not. Assume by induction that π_L is currently the path from v_{c_b} to v_{c_e} where the sum of the weights on edges of π_L is $c_e - c_b$.

If a new candidate c enters I_k then by the strong DP property there must exist a path from v_{c_e} to v_c. Moreover, by the strong DP property, the concatenation of edge strings (as seen when traversing π_L) from v_{c_e} to v_c is $T[c_e..c-1]$. Thus, the only change in memory needed for storing π_L is changing the stored location of the last candidate in I_k to be c, which is not a phantom candidate.

If c_b leaves I_k, then v_{c_b} is removed from the beginning of π_L. If c_b was the only candidate in I_k then π_L becomes empty and we are back to the base case. Otherwise, the new first candidate c in I_k is reached by following the single outgoing edge $e = (v_{c_b}, v_c)$ in G_k. By the strong DP property, $c = c_b + w(e)$, and $\phi(T[1..c-1])$ is the fingerprint concatenation of $\phi(T[1..c_b - 1])$ and $\lambda(e)$.

Information stored in shared memory. For each D_k the data structure stores the guiding graph G_k. In addition the data structure stores the fingerprints in F_k via a perfect hash table that maps each fingerprint $\phi(P)$ to the id of P. Since the space usage per each $\delta \in \Delta$ is $O(d)$ words, the total space used in shared memory is $O(d \log m)$ words.

Information stored in stream memory. The algorithm maintains the text fingerprint of the entire text which is the fingerprint of the text up to the last character that has arrived. This information is updated in constant time per each new character arrival, using the sliding property of ϕ.

The algorithm uses a separate data structure for each of the $O(\log m)$ text intervals. The data structure for text interval I_k maintains all the candidates in I_k by storing a pointer to the beginning of π_L, the locations of the first and last candidates in I_k, and the candidate fingerprint of the first candidate.

Thus, the total space usage per each text interval I_k is $O(1)$, and the total amount of stream memory used is $O(\log m)$.

Character Arrival. We now describe the fairly straightforward processing of a new text character. In particular, we show that the algorithm spends $O(1)$ time per text interval each time a text character arrives, for a total of $O(\log m)$ time per character. Let $T[q]$ be

[4] The reason for addressing the index of v_z directly is due to the possibility of cycles.

the new character that arrives to the stream of text T. The algorithm first updates the text fingerprint of T in the stream memory, and if $\phi(T[q-24..q]) \in F_0$ then the algorithm inserts $q-24$ as a candidate into the first text interval (recall that the last 25 characters in the text are dealt with via maintaining their fingerprint for this purpose). The candidate fingerprint for $q-24$ is computed via the sliding window property from the text fingerprint and $\phi(T[q-24..q])$.

For $\delta_k \in \Delta$ the algorithm checks for the existence of a candidate $c = q - \delta_k + 1$ by probing the head of the path for I_k. If so, the algorithm has to (1) test c, (2) remove c from I_k, and (3) if c is still a candidate and $k \le |\Delta| - 3$ then add c to the data structure for I_{k+1}, while if $k = |\Delta| - 2$ then use $F_{|\Delta|-1}$ to report the id of a pattern occurrence.

To test a candidate $c = q - \delta_k + 1$ the algorithm uses the sliding property of ϕ to compute $\phi(T[c..q])$ from the candidate fingerprint of c (which is stored in stream memory) and the text fingerprint. This takes constant time. We assume from now that c is a candidate. The process for removing a candidate from I_k and adding a candidate to I_{k+1} is described above, and costs $O(1)$ time.

Phantom candidates do not affect complexities and correctness. The treatment of phantom candidates is exactly the same as the treatment of non-phantom candidates since in our algorithm we cannot distinguish between the two. In particular, a location $c = q - \delta_{k+1} + 1$ that was a *phantom* candidate before the last character arrived is tested to see if c is an occurrence of a pattern in D_{k+1}. Since c is a phantom candidate this test **must** fail and c will not be added to the data structure of D_{k+1} (but c could potentially become a phantom candidate again later on). Thus, allowing for phantom candidates does not affect the correctness of the algorithm. Notice that allowing for phantom candidates to exist does not increase the space or time complexities: the space usage is unaffected since the phantom candidates are maintained implicitly within the directed paths, and the time complexity is unaffected since for each arriving character and for each $\delta \in \Delta$, at most one candidate (phantom or not) needs to be considered.

4 Constructing Guiding Graphs

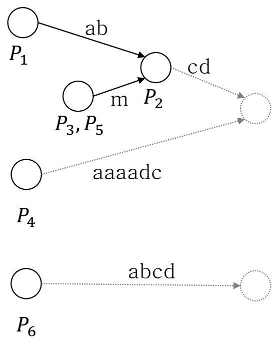
The main technical difficulty is in constructing the guiding graphs. We focus on the construction of the guiding graph for D_k. Recall that $\ell_k = \frac{\delta_k}{5}$ and that the length of every string in D_k is δ_k. A string $P \in D_k$ whose prefix of length $3\ell_k$ appears at least twice in P is said to be of type τ_{pr}, (the "pr" stands for "prefix repetition"). Since the goal here is to convey the main ideas and intuition of how to construct G_k we begin by making a simplifying assumption that there are no strings in D_k of type τ_{pr}. Removing this assumption requires introducing periodicity properties of strings, which we briefly address at the end of this section.

We define the *distance* between two strings $P, P' \in D_k$ to be the smallest possible distance between an occurrence of P and an occurrence of P' in any text. Notice that distances are never negative, and they do not define a metric (P may be at distance 1 from P' and P' may be at distance 1 from P'' but P and P'' might be at distance much larger than 2).

Intuition. The general idea is based on the following observation: if two strings in D_k are within distance at most ℓ_k in the text, then the two strings must share a common substring of length at least $4\ell_k$. However, the converse is not true. That is, not every two strings that share a common substring of length at least $4\ell_k$ have distance at most ℓ_k; see Figure 1.

P_1 = prefsharedsubstringsuf
P_2 = efsharedsubstringsuffi
P_3 = insharedsubstringabcde
P_4 = minsharedsubstringabcd
P_5 = abcdesharedsubstringxy

Figure 1 An example of strings in the same cluster. Notice that P_1 and P_2 could occur in a text within distance 2. Similarly P_3 and P_4 could occur in a text within distance 1. However, every other pair of strings cannot appear in any text within distance less than 15.



P_1 = abcdxyzxyzxyzxyaab
P_2 = cdxyzxyzxyzxyaabbc
P_3 = mcdxyzxyzxyzxyaabx
P_4 = aaaadcxyzxyzxyzxyx
P_5 = mcdxyzxyzxyzxyxyx
P_6 = abcdxxxxxxxxxxxxba

Figure 2 Examples of tries for clusters of strings of type τ_{npr}.

Nevertheless, we would still benefit from clustering the strings in D_k in a way that guarantees the following two properties: (1) if two strings from D_k are at distance at most ℓ_k then these strings appear in the same cluster, and (2) all of the strings in the same cluster share a common substring (possibly at different locations) of length $3\ell_k$ which we call the *seed* of the cluster. Notice that a string in a cluster may contain more than one occurrence of the seed. In order for seeds to be useful, we require that for a given seed of a cluster and a string P in that cluster, the location of the seed in P is the location of the *first* occurrence of the seed in P.

Given such a clustering, for each cluster separately we construct part of the guiding graph as follows (for an example, see Figure 2). For any pair of strings in the same cluster with distance at most ℓ_k, the algorithm synchronizes the two strings, based on the position of the seed in each one of the strings, as follows. Consider the prefix of each such string up to the occurrence of the seed. Then one of the prefixes must be a suffix of the other prefix. Thus, we consider all of the prefixes of all of the strings in a cluster, where each prefix of a string ends right before the occurrence of the seed in that string. We then construct a compacted trie from the reversal of all of the prefixes and associate each string $P \in D_k$ with the vertex in the trie corresponding to the reverse prefix of P. Each vertex in the trie has a single outgoing edge e to its parent (the root has out-degree 0), and the edge string of e is exactly the string corresponding to that edge in the compacted trie. Let T be a text and let c and c' be any two non-phantom candidates in I_k. Let P and P' be the entrance prefixes of c and c', respectively. The distance between P and P' is at most $c' - c \leq \ell_k$, and so P and P' must be in the same cluster. Moreover, $T[c..(c'-1)]$ corresponds exactly to the concatenation of the labels in the compacted trie on the single path from v_P to $v_{P'}$[5].

[5] Notice that if we were to allow strings of type τ_{pr} then this statement would no longer be true.

4.1 Creating Clusters

We consider two separate cases for the strings in D_k that are not of type τ_{pr}. The first are strings in $P \in D_k$ that have a substring of length $3\ell_k$ that occurs at least twice, but since we do not consider strings from type τ_{pr} then the prefix of P of length $3\ell_k$ does not occur more than once in P. Such strings are said to be of type τ_{npr} (the “npr" stands for “non-prefix repetition"). The second are strings that do not have a substring of length $3\ell_k$ that occurs at least twice. Such strings are said to be of type τ_{nr} (the “nr" stands for “no repetition").

Clustering for type $\boldsymbol{\tau_{npr}}$. Notice that a string P of type τ_{npr} could potentially have several substrings of length $3\ell_k$ such that each one of them appears at least twice in P. In order to remove the ambiguity, we treat the leftmost repeated string of length $3\ell_k$ in P as the *only* one that counts, and call it the *base* of P.

We cluster the strings of type τ_{npr} according to their base. That is, all of the strings in the same cluster have the same base, and this base is the seed of the cluster. Thus, we only need to show that if two strings of type τ_{npr} are at distance at most ℓ_k then these strings appear in the same cluster. The proof that this property holds is based on periodicity properties and is left for the full version.

Clustering for type $\boldsymbol{\tau_{nr}}$. Unfortunately, for type τ_{nr} it is impossible to guarantee both desired clustering properties at the same time. To see this, consider $S_1, S_2, S_3, \ldots, S_7 \in D_k$ and a text that contains all of these 7 strings, where for every $1 \leq i \leq 6$, the occurrence of S_i is exactly $\ell_k - 1$ positions before the occurrence of S_{i+1}. Then based on the properties that we are aiming for, all of these strings must appear in the same cluster. However, it is straightforward to construct such an example in which S_1 and S_7 do not share a single common character.

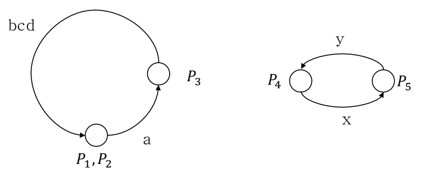
To solve this problem we modify the definition of guiding graphs by generalizing the definition of v_P for $P \in D_k$, using a weaker version of the DP property, and refining the properties that we require from the clustering. Instead of requiring each $P \in D_k$ to have a single associated vertex v_P, we now allow P to be associated with a set of vertices V_P. Recall that it is possible for a vertex to be associated with more than one string.

▶ **Property 2** (Weak Directed Path Property). *Let S be a string where $\delta_k \leq |S| < \delta_k + \left\lfloor \frac{\ell_k}{\log d} \right\rfloor$, such that the prefix and suffix of S of length δ_k are $P_b, P_e \in D_k$, respectively, where P_b and P_e are of type τ_{nr}. Then there exists $v_b \in V_b$ to $v_e \in V_e$ such that there exists a single directed path π in G_k from v_b to v_e and the concatenation of the edge strings for the sequence of edges on π is exactly $S[1..|S| - \delta_k]$, which is the prefix of S until the occurrence of the suffix P_e.*

If $P \in D_k$ of type τ_{nr} is a substring *of S at location h, then the path starting from v_b with total edge weight $h - 1$ must exist and end at a vertex $v_P \in V_P$, so $v_P \in \pi$. Moreover, for any prefix of π with total edge weight w, the concatenation of the edge strings on this prefix is $S[1..w]$.*

The *strong* DP property refers to strings of distance up to $\delta_{k+1} - \delta_k$, but the *weak* DP property only refers to strings of distance up to $\left\lfloor \frac{\ell_k}{\log d} \right\rfloor \leq \left\lfloor \frac{\delta_{k+1} - \delta_k}{\log d} \right\rfloor$. Thus, the algorithm uses $O(\log d)$ separate paths in order to cover the candidates in I_k which is of length $\delta_{k+1} - \delta_k$. This increases the space usage by a $O(\log d)$ factor, but the time cost remains the same.

Finally, the two properties we require from the clustering for type τ_{nr} are: (1) if two strings of type τ_{nr} are at distance at most $\lfloor \frac{\ell_k}{\log d} \rfloor$ then these strings appear together in *some* cluster, and (2) all of the strings in a cluster share a common seed of length $3\ell_k$. The details for finding such a clustering are non-obvious and are left for the full version.



P_1 = abcdabcdabcdabcccd
P_2 = abcdabcdabcdabcdab
P_3 = bcdabcdabcdabcdabc
P_4 = xyxyxyxyxyxyxyxyx
P_5 = yxyxyxyxyxyxyxyxy

Figure 3 Example of connected components for clusters of pattern prefixes of type τ_{pr}.

Clustering for type τ_{pr}. The general idea for treating strings of type τ_{pr} is based on the following properties. Recall the definition of *base* from the clustering of τ_{npr}. Let $P \in D_k$ be of type τ_{pr}. Notice that Since P is of length $5\ell_k$ and the base of P is of length $3\ell_k$, then the two leftmost occurrences of the base in P (one of which is the prefix of P) must overlap. Denote the location of the second occurrence of the base by $r(P) + 1$. We prove (in the full version) that the prefix of P which ends after the *second* occurrence of the base in P (that is at location $r(P) + 3\ell_k$) must be periodic, and the principal period of this prefix is exactly the prefix $P[1..r(P)]$, which is the prefix of P up to the second occurrence of the base. Using periodicity techniques we are able to prove that for every two strings $P, P' \in D_k$ of type τ_{pr}, if the distance between the strings is at most $\ell_k \geq \delta_{k+1} - \delta_k$, then: (1) $r(P) = r(P')$, and (2) $P[1..r(P')]$ is a cyclic shift of $P'[1..r(P')]$. Thus, we cluster the strings in τ_{pr} such that for every two strings P and P' in the same cluster: (1) $r(P) = r(P')$, and (2) $P[1..r(P')]$ is a cyclic shift of $P'[1..r(P')]$. This will help us guarantee that the *strong* DP property holds. Finally, the cyclic shift naturally defines a directed cycle in G_k which captures the synchronization between the strings in a cluster, see Figure 3. Notice that it is possible that the same string will occur several times in a text at locations in a range shorter than ℓ_k. This case occurs only for strings that have a short period (less than ℓ_k), and is straightforward to show that all of these strings must be of type τ_{pr}. Thus, the cyclic graph description captures the relationship between possible occurrences of such a periodic string in the text, but also leads to the possibility of having a non-simple path represent many candidates.

Combining the three types. For type τ_{nr} we are able to guarantee the weak DP property, while for τ_{npr} and τ_{pr} we are able to guarantee the strong DP property, but for each type separately. That is, for two strings $P, P' \in D_k$ of different types, there is no path in G_k from a vertex corresponding to P to a vertex corresponding to P'. Thus, our algorithm creates a separate instance for each one of the three types, and runs them concurrently. Notice that it is possible for a candidate c to be of one type when c is in I_k and of a different type when c enters I_{k+1}. This is permissable since there are separate graphs for different values of k.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

2 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.

3 Amihood Amir, Yonatan Aumann, Gad M. Landau, Moshe Lewenstein, and Noa Lewenstein. Pattern matching with swaps. *J. Algorithms*, 37(2):247–266, 2000. doi:10.1006/jagm.2000.1120.

4 Amihood Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM J. Comput.*, 35(5):1007–1022, 2006. doi:10.1137/S0097539702424496.

5 Amihood Amir, Estrella Eisenberg, and Ely Porat. Swap and mismatch edit distance. *Algorithmica*, 45(1):109–120, 2006. `doi:10.1007/s00453-005-1192-8`.

6 Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994. `doi:10.1016/0020-0190(94)90086-8`.

7 Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *Proceedings of the 27th International Symposium on Algorithms and Computation, ISAAC*, pages 12:1–12:12, 2016. `doi:10.4230/LIPIcs.ISAAC.2016.12`.

8 Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Dictionary matching with one gap. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM*, pages 11–20, 2014.

9 Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Dictionary matching with a few gaps. *Theor. Comput. Sci.*, 589:34–46, 2015.

10 Amihood Amir, Moshe Lewenstein, and Ely Porat. Approximate swapped matching. *Inf. Process. Lett.*, 83(1):33–39, 2002. `doi:10.1016/S0020-0190(01)00302-7`.

11 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`.

12 Amihood Amir and Gonzalo Navarro. Parameterized matching on non-linear structures. *Inf. Process. Lett.*, 109(15):864–867, 2009. `doi:10.1016/j.ipl.2009.04.012`.

13 Amihood Amir and Igor Nor. Generalized function matching. *J. Discrete Algorithms*, 5(3):514–523, 2007. `doi:10.1016/j.jda.2006.10.001`.

14 Tanver Athar, carl Barton, Widmer bland, Jia Gao, Costas S. Illopoulos, Chang Liu, and Solon P. Pissis. Fast circular dictionary-matching algorithm. *Mathematical Structures in Computer Science*, pages 1–14, 2015.

15 Brenda S. Baker. Parameterized pattern matching by boyer-moore-type algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 1995.*, pages 541–550, 1995. URL: `http://dl.acm.org/citation.cfm?id=313651.313816`.

16 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996. `doi:10.1006/jcss.1996.0003`.

17 Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997. `doi:10.1137/S0097539793246707`.

18 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`.

19 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`.

20 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`.

21 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*, pages 778–784, 2009. URL: `http://dl.acm.org/citation.cfm?id=1496770.1496855`.

22 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`.

23 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *23rd Annual European Symposium of Algorithms, ESA*, pages 361–372, 2015. `doi:10.1007/978-3-662-48350-3_31`.

24 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*, pages 2039–2052, 2016. doi:10.1137/1.9781611974331.ch142.

25 Raphaël Clifford, Markus Jalsenius, Ely Porat, and Benjamin Sach. Pattern matching in multiple streams. In *Proceedings of Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012*, pages 97–109, 2012. doi:10.1007/978-3-642-31265-6_8.

26 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. *CoRR*, abs/1708.05223, 2017. arXiv:1708.05223.

27 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.

28 Raphaël Clifford and Tatiana Starikovskaya. Approximate Hamming Distance in a Stream. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, pages 20:1–20:14, 2016. doi:10.4230/LIPIcs.ICALP.2016.20.

29 Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 592–601, 2002. doi:10.1145/509907.509992.

30 Funda Ergün, Hossein Jowhari, and Mert Saglam. Periodicity in streams. In *Proceedings of Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 13th International Workshop, APPROX 2010, and 14th International Workshop, RANDOM*, pages 545–559, 2010.

31 Guy Feigenblat, Ely Porat, and Ariel Shiftan. An improved query time for succinct dynamic dictionary matching. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM*, pages 120–129, 2014.

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

33 Michael J Fischer and Michael S Paterson. String-matching and other products. Technical report, DTIC Document, 1974.

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

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

36 Anna C. Gilbert, Hung Q. Ngo, Ely Porat, Atri Rudra, and Martin J. Strauss. 2/2-foreach sparse recovery with low risk. In *Proceedings of Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013*, pages 461–472, 2013. doi:10.1007/978-3-642-39206-1_39.

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

38 Shay Golan and Ely Porat. Real-time streaming multi-pattern search for constant alphabet. In *25th Annual European Symposium on Algorithms, ESA 2017*, pages 41:1–41:15, 2017. doi:10.4230/LIPIcs.ESA.2017.41.

39 Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. *ACM Trans. Algorithms*, 3(3):29, 2007. doi:10.1145/1273340.1273345.

40 Wei Huang, Yaoyun Shi, Shengyu Zhang, and Yufan Zhu. The communication complexity of the hamming distance problem. *Inf. Process. Lett.*, 99(4):149–153, 2006. doi:10.1016/j.ipl.2006.01.014.

41 Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98*, pages 166–173, 1998. doi:10.1109/SFCS.1998.743440.

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

43 Adam Kalai. Efficient pattern-matching with don't cares. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2002*, pages 655–656, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545468.

44 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.

45 Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct online dictionary matching with improved worst-case guarantees. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM*, pages 6:1–6:13, 2016.

46 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.

47 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, Proceedings*, pages 336–341, 2012. doi:10.1007/978-3-642-34109-0_35.

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

49 S. Muthukrishnan and H. Ramesh. String matching under a general matching relation. In *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*, pages 356–367, 1992. doi:10.1007/3-540-56287-7_118.

50 Hung Q. Ngo, Ely Porat, and Atri Rudra. Efficiently decodable error-correcting list disjunct matrices and applications - (extended abstract). In *Proceedings of Automata, Languages and Programming - 38th International Colloquium, ICALP 2011*, pages 557–568, 2011. doi:10.1007/978-3-642-22006-7_47.

51 Hung Q. Ngo, Ely Porat, and Atri Rudra. Efficiently decodable compressed sensing by list-recoverable codes and recursion. In *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012*, pages 230–241, 2012. doi:10.4230/LIPIcs.STACS.2012.230.

52 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

53 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.

54 Ely Porat and Amir Rothschild. Explicit nonadaptive combinatorial group testing schemes. *IEEE Trans. Information Theory*, 57(12):7982–7989, 2011. doi:10.1109/TIT.2011.2163296.

55 Jakub Radoszewski and Tatiana A. Starikovskaya. Streaming k-mismatch with error correcting and applications. In *2017 Data Compression Conference, DCC*, pages 290–299, 2017. doi:10.1109/DCC.2017.14.

56 Tatiana Starikovskaya. Communication and Streaming Complexity of Approximate Pattern Matching. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, pages 13:1–13:11, 2017. doi:10.4230/LIPIcs.CPM.2017.13.