Online Parameterized Dictionary Matching with One Gap

Avivit Levy and B. Riva Shalom

Department of Software Engineering
Shenkar College, Ramat Gan, Israel
avivitlevy@shenkar.ac.il, rivash@shenkar.ac.il

Abstract. We study the online Parameterized Dictionary Matching with One Gap problem (PDMOG) which is the following. Preprocess a dictionary D of d patterns, where each pattern contains a special *gap* symbol that can match any string, so that given a text that arrives online, a character at a time, we can report all of the patterns from D that parameterized match to suffixes of the text that has arrived so far, before the next character arrives. Two equal-length strings are a parameterized match if there exists a bijection on the alphabets, such that one string matches the other under the bijection. The gap symbols are associated with bounds determining the possible lengths of matching strings. Online Dictionary Matching with One Gap (DMOG) captures the difficulty in a bottleneck procedure for cyber-security, as many digital signatures of viruses manifest themselves as patterns with a single gap. Parameterized match captures possible encryption of the patterns. We also define and study the *strict* PDMOG problem, in which sub-patterns of the same dictionary pattern should be parameterized matched via the same bijection. This captures situations where sub-patterns of a dictionary pattern are encoded simultaneously.

Keywords: pattern matching, dictionary matching, online dictionary matching with gaps, parameterized matching

1 Introduction

Cyber security is a critical modern concern. Network intrusion detection systems (NIDS) perform protocol analysis, content searching and content matching, in order to detect harmful software. Such malware may appear on several packets, hence the need for *gapped matching* [25].

A *gapped pattern* P is one of the form $lp\ \{\alpha, \beta\}\ rp$, where each sub-pattern lp, rp is a string over alphabet Σ, and $\{\alpha, \beta\}$ matches any substring of length at least α and at most β, which are called the *gap bounds*. Gapped patterns may contain more than one gap, however, those considered in NIDS systems typically have at most *one* gap, and are a serious bottleneck in such applications [8]. Consider for example the SNORT software, which is a free open source network intrusion detection system and intrusion prevention system. Analyzing the set of gapped patterns considered by the SNORT software rules shows that 77% of the patterns have at most one gap, and more than 44% of the patterns containing gaps have only one gap [7].

For this reason, Amir et al. [9,8] defined the Dictionary Matching with One Gap problem (DMOG) as follows. Preprocess a dictionary D of total size $\mathfrak{D}$ over alphabet Σ consisting of d gapped patterns each containing a single gap, so that given a query text T of length n over alphabet Σ, we can output all locations ℓ in T, where any gapped pattern ends. Note that, $\mathfrak{D}$ is the sum of lengths of all patterns in the dictionary, *not including* the gaps sizes. For example, let D be $\{P_1 = a\ b\ a\ \{2, 4\}\ d\ d,$

Avivit Levy, B. Riva Shalom: Online Parameterized Dictionary Matching with One Gap, pp. 41–55.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8 © Czech Technical University in Prague, Czech Republic

$P_2 = a\ b\ \{2,4\}\ c\ d$, $P_3 = b\ a\ \{2,4\}\ c\}$. Then, the text $T = c\ \ d\ \ a\ \ b\ \ a\ \ b\ \ e\ \ b\ \ c\ \ d\ \ a\ \ c$ has occurrences of P_2 ending at location 10 with gap length 4 and gap length 2, and of P_3 ending at location 9 with gap length 3.

We study an extension to the dictionary matching with one gap (DMOG) problem suggested by Shalom [31], where every pattern in the dictionary has a single gap, in which the gapped malware is encrypted in order to evade virus scanners. We consider the situation where units of plain text are replaced with ciphertext according to a fixed system, i.e., a parameterized mapping is used as a strategy of encryption. Parameterized matching is a well-known problem in computer science [12]. Two equal-length strings are a parameterized match if there exists a bijection on their alphabet symbols under which one string matches the other.

The Parameterized Matching problem (PM) is formally defined as follows. Given a Text T of length n and a pattern P of length m, both over alphabet $\Sigma' \cup \Sigma$, s.t. $\Sigma' \cap \Sigma = \emptyset$, output all locations ℓ in T, where there exists a bijection $f : \Sigma \to \Sigma$ and the following hold: (1) $\forall P[i] \in \Sigma'$, $P[i] = T[\ell + i - 1]$, and (2) $\forall P[i] \in \Sigma$, $f(P[i]) = T[\ell + i - 1]$. For example, let $\Sigma' = \{a, b\}, \Sigma = \{x, y, z\}$ for text $T = x\ x\ y\ b\ z\ y\ y\ x\ b\ z\ x$ and pattern $P = z\ z\ x\ b$ there are two p-matches ending at locations $\{4, 9\}$. The former implies mapping function $f(z) = x, f(x) = y$, while the latter implies mapping function $f(z) = y, f(x) = x$. Throughout the paper we denote a parameterized match by *p-match*.

We thus study the Parameterized Dictionary Matching with One Gap (PDMOG) problem formally defined below. However, unlike Shalom [31], who studied the offline scenario where all the text is given in advance, we focus on the online setting.

Definition 1. The Parameterized Dictionary Matching with One Gap problem (PDMOG)*:*

Preprocess: *A dictionary D consisting of d gapped patterns $\{P_i\}$ over alphabet $\Sigma' \cup \Sigma$, s.t. $\Sigma' \cap \Sigma = \emptyset$, where every P_i is of the form $lp_i\{\alpha_i, \beta_i\}rp_i$ and α_i, β_i, are P_i's gap boundaries.*

Query: *A text T of length n over alphabet $\Sigma' \cup \Sigma$, $\Sigma' \cap \Sigma = \emptyset$*

Output: *All locations ℓ in T, where a p-match of gapped pattern $P_i \in D$* ends*, i.e., there exist bijections $f_1, f_2 : \Sigma \to \Sigma$ and the following hold for some P_i and a gap length $g \in [\alpha_i, \beta_i]$:*
(1) $\forall lp_i[j] \in \Sigma'$, $lp_i[j] = T[\ell - |rp_i| - g - |lp_i| + j]$.
(2) $\forall lp_i[j] \in \Sigma$, $f_1(lp_i[j]) = T[\ell - |rp_i| - g - |lp_i| + j]$.
(3) $\forall rp_i[j] \in \Sigma'$, $rp_i[j] = T[\ell - |rp_i| + j]$.
(4) $\forall rp_i[j] \in \Sigma$, $f_2(rp_i[j]) = T[\ell - |rp_i| + j]$.

Note, that the gapped pattern parts lp_i, rp_i need to be p-matched separately, hence, each can be matched using a different matching function. For example, let $\Sigma' = \{a, b\}, \Sigma = \{q, u, v, w, z\}$ for text $T = a\ u\ v\ b\ u\ b\ a\ z\ w\ w\ z$ and $D = \{P_1 = z\ x\ b\ z\{2,4\}u\ u\ q$, $P_2 = u\ b\ q\{1,4\}a\ u\ v\}$. We have two p-matches ending at locations $\{11, 9\}$. The first p-matches P_1 using matching function $f(z) = u, f(x) = v$ for lp_1, a gap of length 3 and a matching function $f(u) = w, f(q) = z$ for rp_1. The second p-matches P_2 using $f(u) = v, f(q) = u$ for matching lp_2, a single character gap and $f(u) = z, f(v) = w$ for matching rp_2. For simplicity, we assume that $\Sigma' = \emptyset$. Our solutions can be easily adapted to $\Sigma' \neq \emptyset$ case.

The Strict PDMOG Problem. In some situations it is more reasonable that the encodings of both sub-patterns of the same dictionary pattern are done simultaneously and, therefore, equal. Hence, we suggest another formalization of the PDMOG definition, which we call *strict* PDMOG, that enforces the requirement of both left and right sub-patterns have the same parameterized matching function. The strict PDMOG formal definition is identical to Definition 1, with the additional requirement that $f_1 = f_2$, implying that both sub-patterns of a gapped pattern are parameterized matched via the same bijection function.

In the above example, the parameterized occurrence of P_1 in the text T ending at location 11 is a strict parameterized occurrence, since the bijection $f(z) = u, f(x) = v$ for lp_1, and the bijection $f(u) = w, f(q) = z$ for rp_1 do not contain collisions, i.e., matching of the same character to different characters. In contrary, the p-matching of P_2 ending at location 9 using a matching function $f(u) = v, f(q) = u$ for lp_2 and a mapping function $f(u) = z, f(v) = w$ for rp_2 contains a collision matching the character u to two different characters in lp_2 and rp_2, and therefore, is not a strict parameterized occurrence.

Throughout the paper we use the following notations. Let $D = \{P_1, \ldots, P_d\}$, where every P_i is a gapped pattern of the form $lp_i\{\alpha_i, \beta_i\}rp_i$. We denote $\beta^* = \max_i \beta_i$, $\alpha^* = \min_i \alpha_i$. If D has uniform gap boundaries $\{\alpha, \beta\}$, then $\forall 1 \leq i \leq d$, $\alpha_i = \alpha$, $\beta_i = \beta$.

1.1 Related Previous Work and the Current Work

Dictionary Matching with Gaps. Dictionary matching has been amply researched (see e.g. [1,2,3,4,6,15]). The problem definition varies and many parameters affect the complexity when patterns are gapped. [30] [14] and [13] solve the problem, yet their solutions include a factor of *socc* – the total number of occurrences of the sub-patterns in the text, which can be very large. Others [26,33] solve the problem of matching a set of patterns with variable length of don't cares, yet, they report only a leftmost occurrence of a pattern if there exists one. In [21] an online algorithm for the problem is given, however, at most one occurrence for each pattern at each text position is reported.

The first results on the DMOG problem are due to [9], which solved the offline DMOG problem for a single set of gap boundaries reporting all appearances of all gapped patterns. They suggest an algorithm using range queries and an additional algorithm using a look-up table. The solution is generalized to variable-length gaps dictionaries achieving linear space in [22]. Finally, the online DMOG problem is considered in [8,7] and a connection to the 3SUM conjecture is shown. The *conditional lower bound* (CLB) provides insight for the inherent difficulty in DMOG, and reveals that the CLB from the 3SUM conjecture can be phrased in terms of a new parameter of the problem – $\delta(G_D)$, where G_D is a graph representing the input dictionary. $\delta(G_D)$ turns out to be a small constant in some input instances considered by NIDS. In fact, $\delta(G_D)$ is not greater than 5 in the graph created using SNORT software rules [7]. This leads to designing algorithms whose runtime can be expressed in terms of $\delta(G_D)$, and can therefore be helpful in such practical settings. Online Dictionary Recognition with One Gap (DROG), where each gapped pattern is reported at most once during the entire online text scan, is considered in [10].

Parameterized Matching. The problem was initially defined as a tool for software maintenance, motivated by the observation that programmers introduce duplicate code into large software systems when they add new features or fix bugs, thus slightly modify the duplicated sections [12]. The problem has many application in various fields, such as Image processing, where parameterized matching can help searching an icon on the screen, or improving ergonomy of databases of URLS [28] and extensive research followed (see [27,28]). Among the extensions are: suggesting a parameterized version of KMP [5], studies of maximal p-matches over a threshold length and a p-suffix tree [11,12], parameterized fixed and dynamic dictionary problems presented [23] and improved by [20], efficient parameterized text indexing [18], p-suffix arrays [17], parameterized LCS [24], and many more.

Parameterized Dictionary Matching with One Gap (PDMOG). Shalom [31] first formalized the extension of the dictionary matching with one gap (DMOG) problem to parameterized dictionary matching with one gap (PDMOG), in which the gapped malware is encrypted in order to evade virus scanners. [31] study the *offline* scenario where all the text is given in advance, and give two solutions. The first solves offline PDMOG for dictionaries with non-uniform gap boundaries with $O(n(\beta^* - \alpha^*) \log^2 d + occ)$ query time, where n is the size of the text, d is the number of gapped patterns in the dictionary, $\beta^* - \alpha^*$ is the maximal gap size and occ is the number of the gapped patterns reported as output. The second offline PDMOG solution is for dictionaries with uniform gap boundaries with $O(n(\beta - \alpha) + occ)$ query time, where n is the size of the text, $\beta - \alpha$ is the gap size and occ is the output size.

This Paper Contributions. In this paper, we focus on the *online* setting of PDMOG and *strict* PDMOG, where the text arrives a character at a time, and the requirement is to report all gapped patterns that parameterize-match to suffixes of the text that has arrived so far, before the next character arrives. This is the more realistic situation in NIDS applications. The main contributions of this paper are:

- Formalizing the online PDMOG and strict PDMOG problems, which are natural extensions to the online DMOG problem.

- Obtaining algorithms for online PDMOG that are fast for some practical inputs. A basic property of suffixes of any dictionary pattern is that they form a chain where each is a proper suffix of the other. This property, that was crucial in the online DMOG solutions [8,7], no longer holds for parameterized suffixes. Nevertheless, we show that it is possible to by-pass this difficulty.

- Obtaining algorithms for online strict PDMOG that are fast for some practical inputs where dictionary sub-patterns contain the same alphabet symbols. Enforcing the requirement that both left and right sub-patterns of a dictionary pattern are p-matched using the same parameterized matching function necessitates representation and maintenance of these functions.

Paper Organization. The paper is organized as follows. Section 2 describes our results for the online PDMOG problem. Section 3 details the algorithms for solving online strict PDMOG problem. Section 4 concludes the paper and poses some open questions.

2 Solving Online PDMOG

In this section we describe the online PDMOG solution. We detail the changes and modifications that should be made to the basic scheme of [8,7] in order to adapt it to our problem.

The Bipartite Graph G_D. We use [8,7] dictionary representation as a graph $G_D = (V, E)$: sub-patterns are represented by vertices and there is an edge $(u, v) \in E$ if and only if there is a pattern $P_i \in D$, where lp_i is associated with node u and rp_i is associated with v. The graph $G_D = (V, E)$ is converted to a bipartite graph by creating two copies of V called L (left vertices) and R (right vertices) as follows. For every edge $(u, v) \in E$, an edge is added to the bipartite graph from $u_L \in L$ to $v_R \in R$, where u_L is a copy of u and v_R is a copy of v.

P-Matches Detection. Parameterized matching does not require exact matches between the characters, but rather to capture the characters order. Therefore, Baker [12] defined a *p-string* over a string $S = s_1, s_2 \cdots$ using the *prev* function, where $prev(s_i) = s_i$ in case $s_i \in \Sigma'$, but for $s_i \in \Sigma$, $prev(s_i) = 0$ if s_i is the leftmost position in S of s_i, and $prev(s_i) = i - k$ if k is the previous position to the left at which s_i occurs. For example, let $\Sigma' = \{a, b\}$, $\Sigma = \{u, v\}$ and $S = a\ b\ u\ v\ a\ b\ u\ v\ u$, then the p-string of S is $prev(S) = a\ b\ 0\ 0\ a\ b\ 4\ 4\ 2$.

Lemma 2. [12] *Strings S_1, S_2 have $prev(S_1) = prev(S_2)$ if and only if they are p-matched.*

[23] construct a modified Aho-Corasick automaton (AC) [1] suitable for p-strings similarly to the original AC construction, with modifications to goto and fail links adapting it to work with p-strings. Their automaton is constructed in time $O(\mathfrak{D} \log |\Sigma|)$, takes $O(m \log m) = O(\mathfrak{D} \log \mathfrak{D})$ bits, where m is the number of automaton states, and reports all p-matches of dictionary D patterns in text T in $O(|T| \log |\Sigma| + occ)$ time, where occ is the number of reported occurrences. If only the longest pattern located for each text location is reported, the query is answered in $O(|T| \log |\Sigma|)$.[1]

In the preprocessing, we calculate in linear time the p-string, $prev(lp_i)$, $prev(rp_j)$ for every lp_i, rp_j of some $P_i, P_j \in D$, and construct a parameterized AC automaton upon them, denoted by pAC. Using a standard binary encoding technique each character costs $O(\log |\Sigma|)$ worst-case time. However, for simplicity of exposition, $|\Sigma|$ is assumed to be constant, whenever this assumption is not critical, i.e, if $|\Sigma|$ only appears as a logarithmic factor due to this binary encoding. Note that, the *prev* function does not preserve the suffix relation of the strings it is applied to. Consider sub-patterns x, y, where x is a suffix of y, then, $prev(x)$ is not necessarily a suffix of $prev(y)$. For example, consider $lp_i = uuua$ and its suffix uua. It holds that $prev(lp_i) = 011a$, yet $prev(uua) = 01a$, which is not a suffix of $011a$. Nevertheless, p-suffixes of all dictionary sub-patterns can be traced using fail links of pAC automaton.

The pAC automaton consists of states representing p-strings of prefixes of dictionary sub-patterns. A state representing *prev* function of a sub-pattern lp_i or rp_j is called an accepting state. An arriving character may correspond to several arriving

[1] [19] suggest a space efficient data structure for the parameterized dictionary matching, improving the pAC automaton of [23] by using sparsification technique. Due to our cyber security motivation, we prefer the data structure of [23] for its faster query time.

parameterized sub-patterns, since *prev* function of a sub-pattern could be a proper p-suffix of *prev* function of another sub-pattern. We therefore, phrase complexities in terms of $plsc$ – the maximum number of sub-patterns that their prev functions are p-suffixes of each other, implying vertices in the bipartite graph that arrive due to a character arrival. A similar (probably smaller) lsc factor was used in [8,7] DMOG solutions, and even for simplified DMOG relaxations [21]. While lsc (and $plsc$) could theoretically be as large as d, in many practical situations it is very small. A graph created using SNORT software rules has lsc not greater than 5 [7]. In natural languages dictionaries such as the English dictionary lsc is also a small constant. While it is possible to find a suffix chain of English words with length 7, it is difficult (if possible) to find chains of greater length.

At each time unit, at most $plsc$ vertices are handled, as follows. A vertex $u \in L$, representing a longest sub-pattern associated with the current accepting pAC automaton state, is handled. Other (not necessarily proper) p-suffixes of that sub-pattern are also handled. The preprocessing enabling this procedure uses a p-graph structure.

The P-Graph Structure. We construct a graph pG among the sub-patterns associated with vertices of G_D, where an edge $(u', u) \in E(pG)$ if and only if the sub-pattern associated with u' is a p-suffix of the sub-pattern associated with u. An additional *end* vertex corresponding to the empty string is added to the graph pG, since it is a p-suffix of every sub-pattern. The graph pG can be constructed in linear time while constructing the pAC automaton of D. The bipartite graph G_D vertices arriving due to a text character arrival correspond to vertices on a BFS scan of pG from a vertex u associated with the pAC accepting state (one of the longest sub-patterns having the same p-suffix recognized by this state), creating a BFS-tree rooted at u, not including the *end* leaves of the BFS-tree.

Text Scan. This phase online detects *p-matching* sub-patterns in the text, while saving in adequate data structures occurrences of sub-patterns represented by $u \in L$ nodes that where located during the proper bounds of time units ago (which are calculated differently for the uniform/non-uniform gap bounds cases). When a sub-pattern represented by a $v \in R$ node is p-matched, parameterized occurrences of all gapped patterns P_i where rp_i is represented by $v \in R$ and lp_i is represented by $u \in L$ saved in the data structures, are reported. During text scan phase, $prev(T)$ is calculated online using a $|\Sigma|$-sized array preserving for each $\sigma \in \Sigma$ its last occurrence. Scanning $prev(T)$ using pAC enables finding all sub-patterns p-matching $T[1..\ell]$ *ending* at ℓ. Note, that even for non-fixed alphabets, calculating $prev(T)$ requires $O(|T|)$ time by using perfect hash tables for latest occurrence position of a character in T. Due to synchronization reasons described in [8], removal from the data structures of vertices u that become non-relevant is delayed by $M - 1$ time units, where M is the length of the longest sub-pattern corresponding to a vertex in R.

2.1 PDMOG via Graph Orientations

Graph Orientation. As in [8,7], the graph G_D is preprocessed using linear time greedy algorithm suggested by Chiba and Nishizeki [16] to obtain a $\delta(G_D)$-orientation of the graph G_D, where every vertex has out-degree at most $\delta(G_D) \geq 1$. Orientation is viewed as assigning "responsibility" for data transfers occurring on an edge to one

of its endpoints, depending on the direction of the edge in the orientation. If an edge $e = (u, v)$ is oriented from u to v, the vertex u is called a *responsible-neighbour* of v and v an *assigned-neighbour* of u. The notion of graph *degeneracy* $\delta(G_D)$ is defined as follows. The degeneracy of a graph $G = (V, E)$ is $\delta(G) = \max_{U \subseteq V} \min_{u \in U} d_{G_U}(u)$, where d_{G_U} is the degree of u in the subgraph of G induced by U. Hence, the degeneracy of G is the largest minimum degree of any subgraph of G. A non-multi graph G with m edges has $\delta(G) = O(\sqrt{m})$, and a clique has $\delta(G) = \Theta(\sqrt{m})$. The degeneracy of a multi-graph can be much higher.

The construction and use of the data structures in the algorithms is done as in [8,7], except for the use of the auxiliary pG for recognizing the actual arriving vertices when an accepting state of pAC is reached. This gives Theorem 3.

Theorem 3. *1. The online PDMOG problem with uniformly bounded gap borders can be solved in: $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $O(\delta(G_D) \cdot plsc + pocc)$ time per text character, where pocc is the number of parameterized patterns reported due to character arrival, and $O(\mathfrak{D} + plsc \cdot (\beta + M))$ space.*

2. The online PDMOG problem with non-uniformly bounded gap borders can be solved in: $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $\tilde{O}(plsc \cdot \delta(G_D) + pocc)$ time per text character, where pocc is the number of parameterized patterns reported due to character arrival, and $O(\mathfrak{D} \log |\Sigma| + plsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + plsc \cdot \alpha^*)$ space.

2.2 PDMOG via Threshold Orientations

Subsection 2.1 focuses on orientations whose out-degree is bounded by $\delta(G_D)$. Thus, when $\delta(G_D) = \sqrt{d}$ the PDMOG algorithms basically take $O(plsc \cdot \sqrt{d})$ time. In the non-uniform case the degeneracy can be much larger, since the same sub-patterns can represent different gapped patterns if they have different gaps boundaries, thus two vertices can be connected by more than one edge. Moreover, the $plsc$ factor maybe larger than the lsc factor used in DMOG solutions. Therefore, in this subsection we reduce the factor of $plsc \cdot \delta(G_D)$ to $\sqrt{plsc \cdot d}$, by using a different graph orientation method, referred to as a *threshold* orientation.

Definition 4. *A vertex in G_D is* heavy *if it has more than $\sqrt{d/plsc}$ neighbors, and* light *otherwise.*

Two key properties are used: light vertices have at most $\sqrt{d/plsc}$ neighbors, and the number of heavy vertices is less than $\sqrt{plsc \cdot d}$. We orient all edges that touch a light vertex to leave that vertex, breaking ties arbitrarily if both vertices are light. Thus, every edge e connecting a light vertex with a light/heavy vertex, the light vertex is the responsible-neighbor, and the heavy vertex, if exists in e, is the assigned-neighbor. We handle differently edges with at most one heavy vertex as an endpoint and edges connecting two heavy vertices.

Edges Connecting at Most One Heavy Vertex. Data structures used for dealing with edges where at most one of its endpoints is heavy when considering uniformly bounded gaps, are as in Subsection 2.1 in uniformly bounded gaps (ordered reporting lists $\mathcal{L}_v$ for each $v \in R$, ordered lists τ_u of the time stamps for each $u \in L$ and the list $\mathcal{L}_\beta$ of the last $\beta + M$ vertices $u \in L$). Data structures used for dealing with edges where at most one of their endpoints is heavy when considering non-uniformly

bounded gaps, are as in Subsection 2.1 in non-uniformly bounded gaps (Range query data structures S_v for each $v \in R$, ordered lists τ_u of the time stamps for each $u \in L$ and the list $\mathcal{L}_{\beta^*}$ of the last $\beta + M$ vertices $u \in L$).

Edges Connecting two Heavy vertices. The set of heavy vertices is less than $\sqrt{plsc \cdot d}$, and so even if the number of vertices from L arriving at the same time can be as large as $plsc$ and the set of their neighbors can be very large, the number of vertices in R is still less than $\sqrt{plsc \cdot d}$. Thus, using a batched scan on all of R keeps the time cost low, after some preprocessing. In addition, at each time unit, we handle only a single $u \in L$ currently arriving, one representing a longest sub-pattern found by the pAC automaton at that time, which is a sub-pattern associated with the current accepting state. Other sub-patterns, which are (not necessarily proper) p-suffixes of that sub-pattern are handled implicitly, without increasing the time complexity unless they are reported. The preprocessing that enables this procedure uses a p-heavy-graph structure phG (replacing the tree-structure T used in [8,7]).

The p-heavy-Graph Structure. A graph phG among the sub-patterns associated with *heavy* vertices from L is constructed, where an edge $(u', u) \in E(phG)$ if and only if the sub-pattern associated with u' is a p-suffix of the sub-pattern associated with u. An additional *end* vertex corresponding to the empty string is added to the graph phG, since it is a p-suffix of every sub-pattern. The graph phG can be constructed in linear time during the construction of pG. The bipartite graph G_D *heavy* vertices arriving due to a text character arrival correspond to vertices on a BFS scan of phG from some vertex u, creating an $O(plsc)$-size BFS-tree rooted at u, not including the *end* leaves of the BFS-tree.

The construction and use of the data structures in the algorithms is done as in [8,7] except for the replacement of T by phG and the use of the auxiliary pG for recognizing the actual arriving vertices when an accepting state of pAC is reached. This gives Theorem 5.

Theorem 5. *1. The online PDMOG problem with uniform gap borders can be solved in $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $O(plsc + \sqrt{plsc \cdot d} + pocc)$ time per text character, and $O(\mathfrak{D} + plsc(\beta + M))$ space.*

2. The online PDMOG problem with non-uniform gap borders can be solved in $\tilde{O}(\mathfrak{D} + d(\beta^* - \alpha^*))$ preprocessing time, $\tilde{O}(plsc + \sqrt{plsc \cdot d}(\beta^* - \alpha^* + M) + pocc)$ time per text character, and $O(\mathfrak{D} \log |\Sigma| + d(\beta^* - \alpha^*) + \sqrt{plsc \cdot d}(\beta^* + M))$ space.

3 Solving Online Strict PDMOG

In this section we study online strict PDMOG problem, requiring both sub-patterns of a gapped pattern to be p-matched via the same function. We solve the problem for dictionaries where every sub-pattern contains all characters of Σ. The basic algorithmic scheme is as Section 2, however, there are now additional issues to handle due to the new requirement.

The Matching Permutation. The pAC automaton reports an occurrence of parameterized match of sub-patterns in the text, yet it does not report the matching function used. We define the *matching permutation*, $\pi_{u,t}$, via which the current p-matching of sub-pattern represented by node u to the suffix ending at time t in the

text occurred. $\pi_{u,t}$ can be represented as a $|\Sigma|$-length array, where entry i contains σ' if $f(\sigma_i) = \sigma'$ for the current p-match. Hence, we can save at every step the last M symbols of the text, where M is the length of the longest sub-pattern in the dictionary. $\pi_{lp_i,t}$ of a sub-pattern lp_i can be calculated in $|\Sigma_{lp_i}|$ time, where Σ_{lp_i} contains all distinct symbols that appear in lp_i.

3.1 Uniformly Bounded Gaps

The Permutation Tree. Matching permutations are saved in a *permutation tree* data structure. The permutation tree T_u maintains a set S of permutations of Σ used to p-match occurrences of sub-pattern associated with node u. A permutation tree can be basically maintained as a y-fast trie [32] with additional linked lists in its leaves.

The data structures used in this case are:

1. For each $u \in L$ we save **a y-fast trie** T_u containing all matching permutations $\pi_{u,t}$ via which the sub-pattern represented by u was p-matched to the text ending at location t, at least α and at most β time units ago. For every node l in T_u, representing the matching permutation π_l, we save an ordered list τ_{u,π_l} of time stamps of the occurrences of u p-matching the text via π_l.
2. For each vertex $v \in R$, we save **a y-fast trie** T_v containing all matching permutations $\pi_{u,t}$ via which the sub-pattern represented by u was matched to the text ending at location t, at least α and at most β time units ago, where u is **a responsible-neighbours of** v. For every node l' in T_v, representing a permutation $\pi_{l'}$, we save an ordered list $\mathcal{L}_{v,\pi'_l}$ of links to the nodes representing permutation $\pi_{l'}$ in trees T_u, where u is a responsible neighbour of v.
3. The **list** $\mathcal{L}_\beta$ of delayed vertices $u \in L$ for at least α time units before they are considered. To each node u in $\mathcal{L}_\beta$ we attach the matching permutation $\pi_{u,t}$ via which the sub-pattern represented by u was p-matched to the text ending at time stamp t.

The details of the construction and use of the data structures in the algorithms are as follows.

When the *pAC* automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a sub-pattern that its *prev* function is a p-suffix of the *prev* function of the sub-pattern represented by state s,

1. If the arrived vertex is $v \in R$ that was p-matched to the text via $\pi_{v,t}$,
   (a) Search the matching permutation $\pi_{v,t}$ in T_v.
   (b) if $\pi_{v,t}$ appears in T_v at node l',
   Let $l^* = \mathcal{L}_{v,\pi_{l'}}.first$ (a link to a node $l \in T_u$, where $\pi_l = \pi_{l'}$).
      i. Let $t' = \tau_{u,\pi_l}.first$
      ii. while $t' \geq t - m_v - \beta - 1$
         A. Report edge (u, v) with matching permutation $\pi_{v,t}$, where the nodes u, v appearances are at locations t', t.
         B. If all appearances of the gapped pattern associated with edge (u, v) are required, continue the scan of t' elements of τ_{u,π_l} while $t' \geq t - m_v - \beta - 1$.
         C. Let l^* be the next link in the list $\mathcal{L}_{v,\pi_{l'}}$.
   (c) For every vertex u which v is its responsible-neighbour.
      i. If T_u contains a node l where $\pi_l = \pi_{v,t}$, let $t' = \tau_{u,\pi_l}.first$

   ii. If $t - m_v - \beta - 1 \leq t'$, report edge (u, v) with matching permutation $\pi_{v,t}$ where the nodes u, v appearances are at locations t', t.
   iii. If all appearances of the gapped pattern associated with edge (u, v) are required, continue the scan of the elements of τ_{u,π_l} while $t - m_v - \beta - 1 \leq t'$ where t' is the next element in τ_{u,π_l}.
2. If the arrived vertex is $u \in L$, $(u, \pi_{u,t})$ is inserted into $\mathcal{L}_\beta$.

In addition, the data structures are updated by perviously arrived nodes $u \in L$ saved in $\mathcal{L}_\beta$ that have become relevant.
For vertices $u \in L$, where $(u, \pi_{u,t-\alpha-1})$ was inserted into $\mathcal{L}_\beta$, $\alpha + 1$ time units before time t,

1. Search for node l, representing the matching permutation $\pi_{u,t-\alpha-1}$ in T_u.
2. If $\pi_{u,t-\alpha-1}$ is not saved in T_u, insert a new node l representing $\pi_{u,t-\alpha-1}$ to T_u.
3. For every $v \in R$ that is an assigned neighbour of u,
   (a) Search for node l', representing the matching permutation $\pi_{u,t-\alpha-1}$ in T_v,
   (b) If $\pi_{u,t-\alpha-1}$ is not saved in T_v, insert a new node l' representing $\pi_{u,t-\alpha-1}$ to T_v.
   (c) Add to the beginning of $\mathcal{L}_{v,\pi_{l'}}$ a link to the node l in T_u, where $\pi_l = \pi_{l'}$.
   (d) If τ_{u,π_l} is not empty, remove the previous link to node l of T_u, from $\mathcal{L}_{v,\pi_{l'}}$.
4. The time stamp $t - \alpha - 1$ is added to the beginning of τ_{u,π_l} saved for the node l in T_u.

For vertices $u \in L$,where $(u, \pi_{u,t-\beta-M-1})$ was inserted into $\mathcal{L}_\beta$, exactly $\beta + M + 1$ time units before time t,

1. $(u, \pi_{u,t-\beta-M-1})$ is removed from $\mathcal{L}_\beta$.
2. Search T_u for node l representing the matching permutation $\pi_{u,t-\beta-M-1}$.
3. The time stamp $t - \beta - M - 1$ is removed from the end of the listτ_{u,π_l}, attached to node l.
4. If τ_{u,π_l} becomes empty,
   (a) The node l is deleted from T_u.
   (b) For every T_v, where v is an assigned neighbour of u, the link to node $l \in T_u$ (for $\pi_l = \pi_{u,t-\beta-M-1}$), is removed from the end of $\mathcal{L}_{v,\pi_{l'}}$ where $l' \in T_v$ and $\pi_{l'} = \pi_{u,t-\beta-M-1}$.

This gives Theorem 6.

Theorem 6. *The online strict PDMOG problem with uniformly bounded gap borders can be solved in: $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $O(plsc \cdot \delta(G_D) \log(|\Sigma| \log |\Sigma|) + pocc)$ time per text character, and $O(\mathfrak{D} \log \mathfrak{D} + \delta(G_D) \cdot plsc \cdot |\Sigma|(\beta - \alpha + M) + plsc \cdot \alpha)$ space.*

Proof. In preprocessing, the *pAC* automaton is built in time $O(\mathfrak{D} \log |\Sigma|)$. The query algorithm scans the text *prev* function using *pAC* in $O(|T| \log |\Sigma|)$. At time/location t, the vertex representing the sub-pattern *prev* function is recognized by *pAC*, and all its possible $O(plsc)$ vertices representing p-suffixes sub-patterns in the dictionary. Every such vertex requires $O(\delta(G_D))$ operations on y-fast trie (search - when $\pi_{v,t}$ of $v \in R$ is searched in T_v as well as in all the T_u of its assigned neighbours $u \in L$, insert - when $\pi_{u,t}$ of $u \in L$ is inserted into T_u as well as a link to the node representing $\pi_{u,t}$ inserted to all the T_v of its assigned neighbours $v \in R$, delete - when $\pi_{u,t}$ of $u \in L$ is deleted from all the T_v of its assigned neighbours $v \in R$, and from T_u when the p-matches of u become irrelevant.) Search, insert and delete operations applied to a

y-fast trie, require $O(\log(|\Sigma| \log |\Sigma|))$ time each, as the number of possible matching permutations is $O(|\Sigma|^{|\Sigma|})$ and each operation on a y-fast trie costs $O(\log \log U)$, where U is the maximum value of an element [32].

Reporting the output: A vertex $v \in R$ that was p-matched via permutation $\pi_{v,t}$ scans the list $\mathcal{L}_{v,\pi_{l'}}$ of node $l' \in T_v$ representing permutation $\pi_{v,t}$, where each element in the list is a link to a node $l \in T_u$, representing permutation $\pi_{v,t}$ for some responsible neighbour u of v. If τ_{u,π_l} of node l is scanned, each time stamp represents additional parameterized occurrence of u. $\mathcal{L}_{v,\pi_{l'}}$ scan terminates when a link to τ_{u,π_l} is reached, where the gap between the newest time stamp of τ_{u,π_l} and $t - m_v + 1$ (the current v occurrence beginning) is larger than β, as the rest of the elements of $\mathcal{L}_{v,\pi_{l'}}$ are older. Hence, each considered element in $\mathcal{L}_{v,\pi_{l'}}$, except the last one, is a parameterized occurrence. In addition, each scanned element in the τ_{u,π_l} lists of nodes in T_u trees, where u is an assigned-neighbor of v and $\pi_l = \pi_{v,t}$, is reported.

Regarding space: The *pAC* automaton requires $O(\mathfrak{D} \log \mathfrak{D})$ space. Each T_u maintains distinct permutations of all p-matches of u in T at the last $\beta - \alpha + M$ locations. Since at every time there are at most $plsc$ p-matches of sub-patterns simultaneously, all T_u for $u \in L$ have at most $plsc \cdot (\beta - \alpha + M)$ nodes and the same total time stamps number. For each T_u node, the saved matching permutations represented by it require $|\Sigma|$ space. Hence, the total T_u trees size is $O(plsc \cdot |\Sigma| \cdot (\beta - \alpha + M))$. The space of all permutation trees T_v, $v \in R$, is $O(\delta(G_D) \cdot plsc \cdot |\Sigma|(\beta - \alpha + M))$, since a single $l \in T_u$ can be linked to $\delta(G_D)$ leaves of T_v, where u is a responsible-neighbor of v. Additional $O(plsc \cdot \alpha)$ is required for the $u \in L$ vertices maintained by $\mathcal{L}_\beta$ for α time units until they are considered as arrived.

3.2 Non-Uniformly Bounded Gaps

Non-uniformly bounded gapped patterns yield a multi-graph, where each edge $e = (u, v)$ has its own boundaries $\{\alpha_e, \beta_e\}$, as (u, v) with boundaries $[3, 5]$ is a distinct edge from (u, v) with boundaries $[4, 10]$. A framework similar to Subsection 3.1 is used, yet using permutation trees is not efficient as the information saved at the leaves of the trees should be checked and not necessarily be reported, due to the different boundaries of the edges. Fortunately, we can overcome this by exploiting an alphabetical ordering of the permutations which maps each permutation π into a unique number $num(\pi)$ in $O(|\Sigma|)$ time. Thus, a fully dynamic data structure S_v supporting 6-sided 3-dimensional orthogonal range reporting queries, is used (instead of the 4-sided 2-dimensional S_v in Subsection 2.1) for saving occurrences of responsible neighbour of v. Similar structures for 2-dimensional orthogonal range reporting queries are used for the time stamps of the occurrences of $u \in L$ nodes.

The data structures used in this case are:

1. For each vertex $v \in R$, **a data structure** S_v maintaining points from $\mathbb{R}^3$, representing all time intervals in which an occurrence of v implies a gapped pattern occurrence due to a previous occurrence of u, a responsible-neighbour of v, if the matching permutations of u and v are the same.
2. For each $u \in L$ we save **a data structure** S_u maintaining points from $\mathbb{R}^2$ representing time stamps t of the occurrences of u with the number of the matching permutation $\pi_{u,t}$.
3. The **list** $\mathcal{L}_{\beta^*}$ of the last $\beta^* + M$ vertices $u \in L$ with their matching permutation. They are delayed for at least α^* time units before they are considered.

The details of the construction and use of the data structures in the algorithms are as follows.

When the pAC automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a sub-pattern that its *prev* function is a p-suffix of the *prev* function of the sub-pattern represented by state s,

1. If the arrived vertex is $v \in R$, that was p-matched to the text via $\pi_{v,t}$,
   (a) A range query of $[0, t - m_v + 1] \times [t - m_v + 1, \infty] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ is performed over S_v.
   (b) The edges representing the range output are reported.
   (c) For every vertex u which v is its responsible-neighbour, such that $e = (u, v)$,
      i. A range query of $[t - m_v - \beta_e, t - m_v - \alpha_e] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ is performed over S_u.
      ii. The edges representing the range output are reported.
2. If the arrived vertex is $u \in L$, $(u, \pi_{u,t})$ is inserted into $\mathcal{L}_{\beta^*}$.

In addition, the *active window* is maintained by updating $\mathcal{L}_{\beta^*}$ and acknowledging arrived nodes $u \in L$ that have become relevant.
For vertices $u \in L$ where $(u, \pi_{u,t-\alpha^*-1})$ was inserted into $\mathcal{L}_{\beta^*}$, exactly $\alpha^* + 1$ time units before time t,

1. The point $(t - \alpha^* - 1, num(\pi_{u,t-\alpha^*-1})$ is inserted to S_u.
2. For every $v \in R$ that is an assigned neighbour of u, such that $e = (u, v)$, the point $(t - \alpha^* + \alpha_e, t - \alpha^* + \beta_e, num(\pi_{u,t-\alpha^*-1}))$ is inserted to S_v.

For vertices $u \in L$, where $(u, \pi_{u,t-\beta^*-M-1})$ was inserted into $\mathcal{L}_{\beta^*}$, exactly $\beta^* + M + 1$ time units before time t,

1. $(u, \pi_{u,t-\beta-M-1})$ is removed from the end of $\mathcal{L}_{\beta^*}$.
2. The point $(t - \beta - M - 1, num(\pi_{u,t-\beta^*-M-1}))$ is removed from S_u.
3. For every v that is an assigned neighbour of u, such that $e = (u, v)$, the point $(t - \beta^* - M + \alpha_e, t - \beta^* - M + \beta_e, num(\pi_{u,t-\beta-M-1}))$ is removed from S_v.

This gives Theorem 7.

Theorem 7. *The online strict PDMOG problem with non-uniformly bounded gap borders can be solved in: $\tilde{O}(\mathfrak{D})$ preprocessing time, $\tilde{O}(plsc \cdot \delta(G_D) + pocc)$ time per text character, and $\tilde{O}(\mathfrak{D} + plsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + plsc \cdot \alpha^*)$ space.*

Proof. Preprocessing is similar to the uniformly bounded gaps case, detailed in 3.1. The query algorithm scans the text *prev* function using pAC. At each time/location t, the vertex representing the sub-pattern *prev* function recognized by pAC at t and all its possible $O(plsc)$ p-suffixes, are considered and their data structures are updates or queried. To implement the data structures, we use Mortensen's data structure [29] supporting the set of $|S|$ points from $\mathbb{R}^d$ with $O(|S| \log^{d-2+\epsilon} |S|)$ words of space, insertion and deletion time of $O(\log^{d-2+\epsilon} |S|)$ and $O((\frac{\log |S|}{\log \log |S|})^{d-1} + op)$ time for range reporting queries on S, where op is the output size.

When a vertex $u \in L$ arrives, i.e., it was p-matched via the matching permutation $\pi_{u,t}$ in time t, the point $(t, num(p_{u,t}))$ is inserted to S_u in $O(\log^{\epsilon} |S_u|)$ time. In addition, for each assigned neighbour v of u, where $e = (u, v)$, the point $(t + \alpha_e + m_v, t + \beta_e + m_v, num(\pi_{u,t}))$, where m_v is the length of the sub-pattern represented by node v, is

inserted into S_v. This insertion requires $O(\log^{1+\epsilon}|S_v|)$ time, yielding the time requires for the $u \in L$ nodes is $O(plsc \cdot (\log^{\epsilon} plsc(\beta^* - \alpha^* + M) + \delta(G_D) \log^{1+\epsilon} plsc(\beta^* - \alpha^* + M)))$.

When a vertex $v \in R$ arrives at time t, a range query $[0, t] \times [t, \infty] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ over S_v returns the points that have (x, y, p) coordinates in the given range, thus a parameterized appearance. The range query is applied to S_v containing at most $plsc(\beta^* - \alpha^* + M)$ points, thus requires $O(\frac{\log^2(plsc \cdot (\beta^* - \alpha^* + M))}{\log\log^2(plsc(\beta^* - \alpha^* + M))} + pocc)$ time. Additional time is required for considering all the u assigned-neighbors of v and applying range query $[t - m_v - \beta_e, t - m_v - \alpha_e] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ on the S_u structures in order to report all occurrences sharing the same matching permutations within the gaps boundaries. The total number of time stamps saved in all T_u trees is $O(plsc(\beta^* - \alpha^* + M))$, thus, the total time required for the $v \in R$ nodes is $O(plsc \cdot (\delta(G_D) \cdot \frac{\log plsc(\beta^* - \alpha^* + M)}{\log\log plsc(\beta^* - \alpha^* + M)} + \frac{\log^2 plsc(\beta^* - \alpha^* + M)}{\log\log^2 plsc(\beta^* - \alpha^* + M)} + pocc)$.

Regarding space: The pAC requires $O(\mathfrak{D} \log \mathfrak{D})$ space. Each S_u maintains all appearances u in the text, at the last $\beta^* - \alpha^* + M$ locations in the text. Hence, all S_u for $u \in L$ have at most $plsc(\beta^* - \alpha^* + M)$ points, thus the space required for them is $O(plsc(\beta^* - \alpha^* + M) \log^{\epsilon} plsc(\beta^* - \alpha^* + M))$. S_v contains points only from its responsible neighbor, thus, each of the $plsc$ vertices that were located at each of the last $\beta^* - \alpha^* + M$ locations in the text can be inserted to $\delta(G_D)$ structures of S_v, yielding the space of all the S_v lists is $O(plsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) \log^{1+\epsilon} plsc \cdot \delta(G_D)(\beta^* - \alpha^* + M))$. The additional space usage is required for the $O(plsc)$ vertices maintained for $O(\alpha^*)$ time units by $\mathcal{L}_{\beta^*}$ until they can be considered as arrived.

4 Conclusion and Open Problems

We presented the online PDMOG and strict PDMOG problems and described efficient algorithms for their solution in some practical inputs. As demonstrated in this paper, online PDMOG and strict PDMOG pose additional challenges and difficulties to overcome while designing algorithms for their solutions. It is an open question whether there exist better solutions or efficient solutions for other practical inputs. Additional open research direction is to consider other types of encryption instead of parameterized mapping.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: An aid to bibliographic search.* Commun. ACM, 18(6) 1975, pp. 333–340.
2. A. Amir and G. Călinescu: *Alphabet-independent and scaled dictionary matching.* J. Algorithms, 36(1) 2000, pp. 34–62.
3. A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park: *Dynamic dictionary matching.* J. Comput. Syst. Sci., 49(2) 1994, pp. 208–222.
4. A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer: *Improved dynamic dictionary matching.* Inf. Comput., 119(2) 1995, pp. 258–282.
5. A. Amir, M. Farach, and S. Muthukrishnan: *Alphabet dependence in parameterized matching.* Inf. Process. Lett., 49(3) 1994, pp. 111–115.
6. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh: *Text indexing and dictionary matching with one error.* J. Algorithms, 37(2) 2000, pp. 309–325.
7. A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, and B. R. Shalom: *Mind the gap! online dictionary matching with one gap.* Algorithmica, 2019.

8. A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, and B. R. Shalom: *Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap*, in 27th International Symposium on Algorithms and Computation, ISAAC, Sydney, Australia, December 12-14, 2016, pp. 12:1–12:12.
9. A. Amir, A. Levy, E. Porat, and B. R. Shalom: *Dictionary matching with a few gaps.* Theor. Comput. Sci., 589 2015, pp. 34–46.
10. A. Amir, A. Levy, E. Porat, and B. R. Shalom: *Online recognition of dictionary with one gap*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, August 28-30, 2017, pp. 3–17.
11. B. S. Baker: *Parameterized duplication in strings: Algorithms and an application to software maintenance.* SIAM J. Comput., 26(5) 1997, pp. 1343–1362.
12. B. S. Baker: *A theory of parameterized pattern matching: algorithms and applications*, in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, CA, USA, May 16-18, 1993, pp. 71–80.
13. P. Bille, I. L. Gørtz, H. W. Vildhøj, and D. K. Wind: *String matching with variable length gaps.* Theor. Comput. Sci., 443 2012, pp. 25–34.
14. P. Bille and M. Thorup: *Regular expression matching with multi-strings and intervals*, in Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, Austin, Texas, USA, January 17-19, 2010, pp. 1297–1308.
15. G. S. Brodal and L. Gasieniec: *Approximate dictionary queries*, in 7th Annual Symposium on Combinatorial Pattern Matching CPM, Laguna Beach, California, USA, June 10-12, 1996, pp. 65–74.
16. N. Chiba and T. Nishizeki: *Arboricity and subgraph listing algorithms.* SIAM Journal on Computing (SICOMP), 14(1) 1985, pp. 210–223.
17. S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, and M. Takeda: *Parameterized suffix arrays for binary strings*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, September 1-3, 2008, pp. 84–94.
18. P. Ferragina and R. Grossi: *The string b-tree: A new data structure for string search in external memory and its applications.* J. ACM, 46(2) 1999, pp. 236–280.
19. A. Ganguly, W. Hon, K. Sadakane, R. Shah, S. V. Thankachan, and Y. Yang: *Space-efficient dictionaries for parameterized and order-preserving pattern matching*, in 27th Annual Symposium on Combinatorial Pattern Matching CPM, Tel Aviv, Israel, June 27-29, 2016, pp. 2:1–2:12.
20. A. Ganguly, W. Hon, and R. Shah: *A framework for dynamic parameterized dictionary matching*, in 15th Scandinavian Symposium and Workshops on Algorithm Theory SWAT, Reykjavik, Iceland, June 22-24, 2016, pp. 10:1–10:14.
21. T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen: *Online dictionary matching with variable-length gaps*, in 10th International Symposium on Experimental Algorithms SEA, Kolimpari, Chania, Crete, Greece, May 5-7, 2011, pp. 76–87.
22. W. Hon, T. W. Lam, R. Shah, S. V. Thankachan, H. Ting, and Y. Yang: *Dictionary matching with a bounded gap in pattern or in text.* Algorithmica, 80(2) 2018, pp. 698–713.
23. R. M. Idury and A. A. Schäffer: *Multiple matching of parameterized patterns.* Theor. Comput. Sci., 154(2) 1996, pp. 203–224.
24. O. Keller, T. Kopelowitz, and M. Lewenstein: *On the longest common parameterized subsequence.* Theor. Comput. Sci., 410(51) 2009, pp. 5347–5353.
25. M. Krishnamurthy, E. S. Seagren, R. Alder, A. W. Bayles, J. Burke, S. Carter, and E. Faskha: *How to cheat at securing linux*, Syngress Publishing, Inc., Elsevier, Inc., 2008.
26. G. Kucherov and M. Rusinowitch: *Matching a set of strings with variable length don't cares.* Theor. Comput. Sci., 178(12) 1997, pp. 129–154.
27. M. Lewenstein: *Parameterized pattern matching.* Encyclopedia of Algorithms, 2016, pp. 1525–1530.
28. J. Mendivelso and Y. Pinzón: *Parameterized matching: Solutions and extensions*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, August 24-26, 2015, pp. 118–131.
29. C. W. Mortensen: *Fully dynamic orthogonal range reporting on RAM.* SIAM J. Comput., 35(6) 2006, pp. 1494–1525.

30. M. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth: *Finding patterns with variable length gaps or don't cares*, in 12th Annual International Conference on Computing and Combinatorics COCOON, Taipei, Taiwan, August 15-18, 2006, pp. 146–155.
31. B. R. Shalom: *Parameterized dictionary matching with one gap*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, 2018, pp. 103–116.
32. D. E. Willard: *Log-logarithmic worst-case range queries are possible in space* $\theta(n)$. Information Processing Letters, 17(2) 1983, pp. 81–84.
33. M. Zhang, Y. Zhang, and L. Hu: *A faster algorithm for matching a set of patterns with variable length don't cares.* Inf. Process. Lett., 110(6) 2010, pp. 216–220.