Information and Computation 275 (2020) 104633

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Online recognition of dictionary with one gap ☆

Amihood Amir [a,b], Avivit Levy [c,*], Ely Porat [a], B. Riva Shalom [c]

[a] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
[b] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, United States of America
[c] Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel

ARTICLE INFO

Article history:
Received 28 October 2018
Received in revised form 10 September 2020
Accepted 16 October 2020
Available online 28 October 2020

Keywords:
Pattern matching
Dictionary matching
Network intrusion detection systems

ABSTRACT

We formalize and examine the online Dictionary Recognition with One Gap problem (DROG) which is the following. Preprocess a dictionary D of d patterns each containing a special *gap* symbol that matches any string, so that given a text arriving online a character at a time, all patterns from D which are suffixes of the text that has arrived so far and have not been reported yet, are reported before the next character arrives. The gap symbols are associated with *bounds* determining possible lengths of matching strings. Online DROG captures the difficulty in a bottleneck procedure for cyber-security, as many digital signatures of viruses manifest themselves as patterns with a single gap.

Following the work on the closely related online Dictionary Matching with One Gap problem (DMOG), we provide algorithms whose time cost depends linearly on $\delta(G_D)$, where G_D is a bipartite graph that captures the structure of D and $\delta(G_D)$ is the *degeneracy* of this graph. These algorithms are of practical interest since although $\delta(G_D)$ can be as large as $\sqrt{d}$, and even larger if G_D is a multi-graph, it is typically a small constant in practice.

1. Introduction

Cyber-security is a critical modern challenge. Network intrusion detection systems (NIDS) perform protocol analysis and content searching in order to detect harmful software. Such malware may appear non-contiguously, scattered across several packets, which necessitates matching *gapped* patterns.

A *gapped pattern* P is one of the form $P^1\ \{\alpha, \beta\}\ P^2$, where each subpattern P^1, P^2 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 [37,5,6]. 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 [6]. Therefore, an efficient solution for this case is of special interest.

Though the gapped pattern matching problem arose over 20 years ago in computational biology applications [33,19] and has been revisited many times in the intervening years (e.g. [10,11,16,27,30,32,34,39]), network intrusion detection systems applications necessitate a different generalization of the problem. These applications motivate the *dictionary matching with*

☆ A partial version of this paper appeared in the proceedings of PSC 2017.

* Corresponding author.
E-mail addresses: amir@cs.biu.ac.il (A. Amir), avivitlevy@shenkar.ac.il (A. Levy), porately@cs.biu.ac.il (E. Porat), rivash@shenkar.ac.il (B. Riva Shalom).

https://doi.org/10.1016/j.ic.2020.104633
0890-5401/

one gap (DMOG) problem defined by [5,6], which is a variant of the well-studied dictionary matching problem (see, e.g. [2–4,13,15]).

The DMOG problem was, therefore, studied [5,6,8,20,21] for both the offline and the online settings. Lower bounds on the complexity of this problem as well as (almost) matching upper bounds were described in [6]. These lower bounds expose a hidden parameter of the input dictionary that sheds light on the reason why this problem has resisted attempts at finding a definitive efficient solution on the one hand, while on the other hand, enables describing the solutions in terms of this parameter. The hidden parameter is a function of the *structure* of the gapped patterns, which is exposed by using a graph representation of the dictionary D. We elaborate on this issue in Section 2.

The definition of the DMOG problem requires reporting all occurrences of the dictionary patterns. This is a necessary requirement in order to remove all viruses from a given source. However, the size of the output may be quite large if dictionary patterns occur many times in the source. The process of malware detection is required to be very fast, and in many cases, a faster scan in order to determine whether the source stream is infected by viruses or not – is preferred [37]. We would also like to know which viruses attacked the source in case it is affected, so that an appropriate (slower) exhaustive infection recovery procedure can be applied on the source. Motivated by this need of NIDS applications, we focus in this paper on the recognition of the set of viruses that exists in the source, and formally define the *Dictionary Recognition with One Gap problem (DROG)* as follows:

Definition 1. *The Dictionary Recognition with One Gap problem (DROG)* is:

Input: A dictionary D of d gapped patterns $P_1, \ldots, P_d$ over alphabet Σ, where each pattern has at most one gap, and a text T of length $|T|$ over Σ.

Output: The maximal $S \subseteq D$, where each $P_i \in S$ appears at least once in T.

We study the more practical *online* DROG problem. The dictionary D can be preprocessed in advance, resulting in a data structure. Given this data structure the text T is presented one character at a time, and when a character arrives only the subset of patterns with a match ending at this character *that were not previously reported* should be reported before the next character arrives. Two variants of the gapped dictionary are considered having either uniformly bounded gap borders or non-uniformly bounded gap borders. In the former case, all gapped patterns of the dictionary have the same gaps borders $\{\alpha, \beta\}$, whereas in the latter, every pattern P_i has its own gap borders $\{\alpha_i, \beta_i\}$.

One may also consider the unbounded case, called *unbounded DROG*, where $\alpha_i = \alpha = 0$ and $\beta_i = \beta = \infty$ for every i, where setting β or β_i to ∞ actually means setting it to be of size $|T|$ in the offline version or to the (dynamically changing) current overall text size in the online version. We refer to this version in the discussion of the lower bounds in Section 4. Note that unbounded DROG is a different problem from both that of [25,39], which report only a single (the first) dictionary pattern that exists in a text location, and that of [18], which may report dictionary patterns more than once.

In our online setting, where the query text is arriving character by character, two time cost measures are of interest: a preprocessing time and a time per query text character that arrives. The time per text character is specified in terms of worst case bound as well as an additional report time that depends on the characters preceding each text character. However, in the offline setting the time spent on the query text is not measured per character. Therefore, in order to enable comparison with former offline algorithms as well, we refer in Table 1 to a measure of the total time spent on the query text. For the online algorithms it is simply the sum of the time per text character over all its characters (the total length of the text). This time is referred to as *query time* and is only used in this table for the sake of a consistent comparison. In the rest of the paper, we use the more appropriate measure of *time per query text character*.

Previous work. Finding efficient solutions for the Dictionary Matching with One Gap problem has proven to be a difficult algorithmic challenge as little progress has been obtained even though many researchers in the pattern matching community and the industry have tackled it. Table 1 describes a summary and comparison of previous work. The dictionary, which is the set of d gapped patterns to be detected, could be quite large. A dictionary D for SNORT patterns with one gap has: $\mathfrak{D} = 65073$ and $d = 2478$ [6]. Note that, in NIDS applications the total size of the query text may be huge as it arrives continuously in online manner, but other parameters have fixed values.

Table 1 illustrates that previous formalizations of the problem until that of [5], either do not enable detection of all intrusions or are incapable of detecting them in an online setting, and therefore, are inadequate for NIDS applications. The table also demonstrates that the upper bounds of [5,6] for the DMOG problem are essentially optimal (assuming some popular conjectures). Most importantly, it demonstrates that no previous work has been done on the DROG problem as formalized in this paper.

Our results. One may hope to have algorithms for the DROG problem that will outperform the algorithms for the DMOG problem. Unfortunately, we show that the lower bounds for DMOG shown in [5,6] also hold for the DROG problem. This leads to better understanding of the algorithmic goal of DROG algorithms, as well as the motivation for its definition. The lower bounds reveal that DROG algorithms are expected to work better only if the output size, i.e., the number of dictionary patterns that occur in the text are substantially larger than the number of dictionary patterns d. Such an improvement is maybe somewhat disappointing, however, it is still significant for virus scanners, which may frequently fail due to work

Table 1
Comparison of previous work and new results. The parameters: $\mathfrak{D}$ is the total length of the dictionary patterns (not including the gaps), *lsc* is the longest suffix chain of subpatterns in D (i.e., the longest sequence of dictionary subpatterns such that each is a proper suffix of the next), *socc* is the number of subpatterns occurrences in T, *op* is the number of pattern occurrences in T, α^* and β^* are the minimum left and maximum right gap borders in the non-uniformly bounded case, $\delta(G_D)$ is the degeneracy of the graph G_D representing dictionary D. Abbreviations: preprocess. = preprocessing, rep. = reports, occ. = occurrence, dict. = dictionary, pat. = pattern, loc. = location.

	Preprocess. time	Total query time	Algorithm type	Problem specification
[25]	none	$\tilde{O}(\|T\| + \mathfrak{D})$	online	rep. only first occ. of some dict. pattern
[39]	$O(\mathfrak{D})$	$\tilde{O}(\|T\| + d)$	online	rep. only first occ. of some dict. pattern
[18]	$O(\mathfrak{D})$	$O(\|T\| \cdot lsc + socc)$	online	rep. one occ. per pat. and loc.
[7]	$\tilde{O}(\mathfrak{D})$	$\tilde{O}(\|T\|(\beta - \alpha) + op)$	offline	DMOG
[20,21]	$O(\mathfrak{D})$	$\tilde{O}(\|T\|(\beta^* - \alpha^*) + op)$	offline	DMOG
[5,6]	$O(\mathfrak{D})$	$\tilde{O}(\|T\| \cdot \delta(G_D) \cdot lsc + op)$	online	DMOG
[5,6]	$O(\mathfrak{D})$	$O(\|T\| \cdot \sqrt{lsc \cdot d} \cdot (\beta^* - \alpha^*) + op)$	online	DMOG
[5,6]	$O(\mathfrak{D})$	$\Omega(\|T\| \cdot \delta(G_D)^{1-o(1)} + op)$	online	DMOG
	$O(\mathfrak{D})$	$\Omega(\|T\| \cdot (\beta - \alpha)^{1-o(1)} + op)$	or offline	
This paper	$O(\mathfrak{D})$	$\tilde{O}(\|T\| \cdot \delta(G_D) \cdot lsc + d)$	online	DROG
This paper	$O(\mathfrak{D})$	$O(\|T\| \cdot \sqrt{lsc \cdot d} \cdot (\beta^* - \alpha^*) + d(\beta^* - \alpha^*))$	online	DROG
This paper	$O(\mathfrak{D})$	$\Omega(\|T\| \cdot \delta(G_D)^{1-o(1)} + d)$	online	DROG
	$O(\mathfrak{D})$	$\Omega(\|T\| \cdot (\beta^* - \alpha^*)^{1-o(1)} + d)$	or offline	

overload in such situations [37]. In fact, such failures were the trigger for the initiation of the discussion and cooperation with the industry [37] that led to both the work of [7,5,6] and this paper.

Therefore, our goal in this paper is that the time per character cost would be, as much as possible, independent of the number of occurrences of dictionary patterns in the text. This is a nontrivial requirement as we can no longer afford costly operations that were accounted for by the size of the output for the detection of dictionary patterns at query time in the online DMOG solutions of [5,6]. In our case such costly operations can be afforded for newly detected patterns only. This raises the difficulty of limiting the detection process to the dynamically changing set of yet undetected dictionary patterns.

Moreover, our upper bounds demonstrate that the goal in the DROG problem might be more difficult than the one of DMOG. Note that, while we were able to achieve the same complexities as the DMOG solutions with reduced output size for both cases of dictionary represented by a sparse graph and also for the dictionary with uniform gap borders represented by a dense graph, it is not the case for dictionary with non-uniform gap borders represented by a dense graph. In this case, we were able to get the same query time complexity not including the report of dictionary patterns, however, each such report (which may have significantly smaller size than that of DMOG) has an overhead factor of $(\beta^* - \alpha^*)$.

Paper organization. In Section 2 we give a brief review of the solutions to the online DMOG problem suggested by [5,6]. Section 3 describes our solution for the online DROG problem, which is based on the solutions described in Section 2 with the important and delicate changes and adaptations. In Section 4 we show conditional lower bounds for the DROG problem. Section 5 concludes the paper and poses some open problems.

2. Background: an overview of the DMOG solutions

In this section we give a brief description of the DMOG solutions of [5,6]. The reader who is familiar with their ideas and techniques can skip this section.

The bipartite graph G_D**.** The first baseline idea of their solutions is to represent the dictionary D, consisting of gapped patterns of the form $P^1\{\alpha, \beta\}P^2$, as a bipartite graph $G_D = (V, E)$, where $V = L \cup R$, as follows. The subpatterns of the dictionary D are the graph vertices, where the left subpatterns belong to L and right subpatterns belong to R, and there is an edge $(u, v) \in E$, for $u \in L$ and $v \in R$, if and only if there is a pattern $P \in D$, where $u = P^1$ and $v = P^2$.

Graph orientations. The next baseline idea is to use graph orientations. We will first need to explain the notion of graph *degeneracy* $\delta(G_D)$ defined as follows. The degeneracy of an undirected 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. In words, 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 graph orientation procedure is specified in the next lemma.

Lemma 2. Graph orientation by Chiba and Nishizeki [14]
There exists a linear time greedy algorithm processing a graph $G = (V, E)$ to obtain a directed graph $G_{CN} = (V, E_{CN})$, where every vertex has out-degree at most $\delta(G)$.

Proof. The orientation greedy algorithm to obtain a directed graph $G_{CN} = (V, E_{CN})$ from a given graph $G = (V, E)$ is described in the proof of Lemma 2 of [14] and its claimed property is proven there. □

By Lemma 2, we can preprocess G_D in linear time to obtain a $\delta(G_D)$-orientation of the graph G_D, where every vertex has out-degree at most $\delta(G_D)$. The orientation is viewed as assigning "responsibility" for all 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, then vertex u is called a *responsible-neighbor* of v and v an *assigned-neighbor* of u.

The intuition behind the use of graph orientations is to exploit a small value of $\delta(G_D)$ in order to get efficient algorithms. $\delta(G_D)$ 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 an analysis of the graph created using SNORT software rules [37]. The statement of the algorithmic results is actually more complicated as it depends on other parameters of the input, namely *lsc*, the length of the *longest suffix chain* in the dictionary, i.e., the longest sequence of dictionary subpatterns such that each is a proper suffix of the next. We emphasize that we are not the first to introduce the *lsc* factor, which appeared even in solutions for simplified relaxations of the DMOG problem [18]. At each time unit only a work proportional to $lsc \cdot \delta(G_D)$ is done by their sparse-graph algorithms in order to determine the relevant output dictionary patterns, by balancing the needed checks and updates among the left and right vertices of the graph edges so that each vertex has only a limited "responsibility" and work to do upon its arrival.

Note that the parameter *lsc* could theoretically be as large as d, however, in many practical situations it is very small. Specifically, in an analysis of the graph created using SNORT software rules *lsc* is not greater than 5 [37]. Also, in natural languages dictionaries such as the English dictionary *lsc* is also a small constant [6]. Nevertheless, [6] also presents algorithms that in the most dense cases, where $\delta(D_G) \geq \sqrt{d}$ or *lsc* is large, reduce the dependence on *lsc* and d, by obtaining upper bounds that depend linearly on $\sqrt{lsc \cdot d}$. These algorithms become a vast improvement in the non-uniform case, where the parameter $\delta(G_D)$ could be much higher than $\sqrt{d}$.

For dense graphs, where $lsc \cdot \delta(G_D)$ maybe too large, the idea of orientation as assigning "responsibility" and balancing the needed checks and updates work among the left and right vertices of the graph edges is still used, however, an explicit alternative way to orient the edges, called *threshold orientation*, is used in order to assure low cost. In the threshold orientation, there can be indeed vertices that have a high degree, however, these vertices are not many. Thus, by explicitly assigning "responsibility" to a single such vertex on L (that changes dynamically at every time unit) and using a batch scan of these vertices on R, a low cost can still be achieved. Lemma 3 summarizes the main property of threshold orientation. The details are given in Subsection 2.2.

Lemma 3. Threshold orientation by Amir et al. [6]
Given a gapped dictionary D, there exists a linear time algorithm processing the bipartite graph $G_D = (V, E)$ to obtain a directed bipartite graph $G'_D = (V, E')$, where for every edge one of the following cases holds:

1. *It has an endpoint vertex that has out-degree at most $\sqrt{d/lsc}$. Such an endpoint is called a* light *vertex.*
2. *Both its endpoints are* heavy *vertices, having more than $\sqrt{d/lsc}$ neighbors. There is no (non-trivial)[1] guarantee on the out-degree of heavy vertices, however, their total number is less than $\sqrt{lsc \cdot d}$.*

Subpatterns detection mechanism. An Aho-Corasick (AC) Automaton [2] is used for determining when a subpattern arrives using a standard binary encoding technique, so that each character arrival costs $O(\log|\Sigma|)$ worst-case time for recognizing the arrival of a dictionary subpattern. For simplicity of exposition, $|\Sigma|$ is assumed to be constant. Since each arriving character may correspond to the arrival of several subpatterns when a subpattern is a proper suffix of another, the complexities are phrased in terms of *lsc*, which is the maximum number of vertices in the graph that arrive due to a character arrival.

[1] The trivial guarantee is d.

Two sets of solutions are described: for sparse graphs, where $\delta(G_D) = o(\sqrt{d})$, and for dense graphs. For each of these two cases we describe separately how to deal with uniformly and non-uniformly bounded gap borders. The solutions for these four cases are described hereafter.

2.1. DMOG for sparse graphs

Note that the text is processed online where characters arrive one by one, therefore, a text index is referred to as a time unit in our descriptions.

Uniformly bounded gaps. The data structures used in this case are:

1. For each vertex $v \in R$, a list $\mathcal{L}_v$ maintaining all responsible-neighbors of v, $u \in L$, that arrived at least α and at most β time units ago.
2. For each vertex $u \in L$, an ordered list of time stamps τ_u of the times u arrived within the appropriate gap to the current time unit (which is the current text index in the offline view of the text).
3. The list $\mathcal{L}_\beta$ of delayed vertices $u \in L$ for at least α time units before they are considered. The term *delayed vertices* refer to the fact that the vertices are not treated at the time unit they arrive but only after α times units have passed, in order to assure the minimum gap border is fulfilled. To this end the list $\mathcal{L}_\beta$ is used.

The $\mathcal{L}_v$ lists are updated by deleting vertices u that arrived more than β time units ago and inserting vertices u that arrived α time units and do not appear already in the data structure. Therefore, when an appearance of vertex v is detected, all the patterns $u\{\alpha, \beta\}v$ for $u \in \mathcal{L}_v$ are reported according to the time stamps in τ_u, as the output includes all appearances of the gapped patterns. In addition, the edges for which v is their responsible-neighbor are scanned, and those for which the assigned-neighbor u has a non-empty τ_u, are reported.

The synchronization parameter M**.** The removal of $u \in L$ from $\mathcal{L}_v$ must be delayed by at least $m_v - 1$ time units, where m_v is the length of the substring represented by v, since the gap border is from the beginning of the subpattern v but we do not know it arrived until after the subpattern ends. However, if u is removed from $\mathcal{L}_v$ after a delay of $m_v - 1$, then we may be forced to remove a large number of such vertices at a given time. Therefore, the removal of u is delayed by $M - 1$ time units, where M is the length of the longest subpattern that corresponds to a vertex in R. By synchronizing the deletion of each inserted vertex exactly $M - 1$ time units after the maximum gap border, we ensure that the number of vertices that are deleted at each time unit is bounded by *lsc*, because they are exactly the same as the bound on the number of vertices that may arrive and are inserted at each time unit.

Time and space complexity: [5,6] show that using the above data structures, the DMOG problem with uniformly bounded gap borders can be solved in $O(\mathfrak{D})$ preprocessing time, $O(\delta(G_D) \cdot lsc + op)$ time per text character, where *op* is the number of patterns that are reported due to the character arriving, and $O(\mathfrak{D} + lsc \cdot (\beta + M))$ space.

Non-uniformly bounded gaps. In the case of non-uniformly bounded gaps, each edge $e = (u, v)$ has its own boundaries $\{\alpha_e, \beta_e\}$, yielding a multi-graph, as (u, v) with boundaries $\{2, 4\}$ is a distinct edge from (u, v) with boundaries $\{5, 9\}$. Let α^* and β^* be the minimum left and maximum right gap borders in the non-uniformly bounded dictionary. We refer to the time units between at least α^* and at most β^* time units before time t as the *active window* of t, as these are the time units that occurrences of left subpattern might be relevant. A framework similar to the previous subsection is used, yet, instead of the list $\mathcal{L}_v$, a fully dynamic data structure S_v supporting 4-sided 2-dimensional orthogonal range reporting queries, is used for saving the occurrences of responsible neighbor of v. The properties of the required data structure are as follows.

An orthogonal range reporting data structure. For our purpose we need a data structure S with the following properties:

- Storage type – keeping n points in $\mathbb{R}^2$.
- Fully dynamic – enabling updates (insertions and deletions) of points in time logarithmic in n, the size of S.
- Orthogonal range queries – given a range $[t_1, t_2] \times [t_3, t_4]$, enables to report all points (x, y) in S, where $t_1 \leq x \leq t_2$ and $t_3 \leq y \leq t_4$, in time logarithmic in n.
- Space usage – linear (up to logarithmic factors) in n.

Having such a data structure S_v for each vertex $v \in R$, we deal with each responsible-neighbor $u \in L$ of v (where $e = (u, v)$) that arrived in the active window of time t as follows. The point $(t + \alpha_e + m_v, t + \beta_e + m_v)$ is inserted into S_v, where m_v is the length of the subpattern v, yielding the information saved is the time interval in which an occurrence of v implies an occurrence of a gapped pattern. When a vertex $v \in R$ arrives at time t, a range query $[0, t] \times [t, \infty]$ over S_v returns the points that have (x, y) coordinates in the given range. Therefore, the minimum gap bound for the edge associated with this point is before (or exactly at) time t and the maximum gap bound for the edge associated with this point is after (or exactly at) time t. Thus, such a point reveals a pattern occurrence.

The orthogonal range reporting data structure implementation. Many implementations are known for orthogonal range reporting data structures (e.g., [9,28,26,38,29,31,35]). The implementation of [29] supports keeping the set of n points from $\mathbb{R}^2$ with $O(n \log n)$ space, logarithmic update time and $O(\log n + op)$ time for range reporting queries on S, where op is the size of the output. [31] supports keeping the set of n points from $\mathbb{R}^2$ with $O(n \log^{7/8+\epsilon} n)$ space, insertion and deletion time of $O(\log^{7/8+\epsilon} n)$ and $O(\frac{\log n}{\log \log n} + op)$ time for range reporting queries on S. Using Nekrich's result [35] one can support search time $O(\log n + op)$, update time $O(\log^2 n)$, and space $O(n \log^{\epsilon} n)$ for any $\epsilon > 0$, thus, having better space complexity with slightly worse query and update time complexities. We chose to implement S_v using Mortensen's data structure [31] since we prefer improving query and update times over improving the space.

Time and space complexity: [5,6] show that using the above, the DMOG problem with non-uniformly bounded gap borders on a graph G_D with m edges (gapped patterns) and n vertices can be solved with $O(\mathfrak{D})$ preprocessing time, $\tilde{O}(lsc \cdot \delta(G_D) + op)$ time per query vertex, where op is the number of edges reported due to the vertex arriving, and $\tilde{O}(\mathfrak{D} + lsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + lsc \cdot \alpha^*)$ space.

2.2. DMOG for dense graphs

In the case of dense graphs where $\delta(G_D) = \Omega(\sqrt{d})$, the solutions described above require $\Omega(lsc \cdot \sqrt{d})$ time. For such cases a different method for orienting the graph is suggested by [5,6], referred to as a *threshold* orientation, where a vertex in G_D is defined as *heavy* if it has more than $\sqrt{d/lsc}$ neighbors, and *light* otherwise. Hence, the number of heavy vertices is less than $\sqrt{lsc \cdot d}$. An edge where at least one of its endpoints is light is oriented to leave the light vertex. For such edges the algorithms from the previous subsection are applied in $\tilde{O}(lsc + \sqrt{lsc \cdot d} + op)$ time complexity.

Reporting edges between two heavy vertices is done differently. Although the number of vertices from L that arrive at the same time can be as large as lsc and the number of neighbors of each such vertex can be very large, the number of heavy vertices in R is still less than $\sqrt{lsc \cdot d}$. So [5,6] use a batched scan on all vertices of R to keep the time cost low. In some sense, it is as if we give the "responsibility" to report edges between two heavy vertices to the vertices on L. However, to ensure that the time and space of dealing with reporting data structures of the vertices in R, we give this responsibility to only one vertex of L at each time unit – the one corresponding to the longest subpattern that arrives. The (at most lsc) additional vertices of L that arrive are treated in the reporting data structures of vertices in R implicitly using additional data structures. To this end, the vertices from L are ordered in a tree T according to suffix relations between the subpatterns vertices, where a vertex u is an ancestor of a vertex u' if and only if the subpattern represented by u is a suffix of the subpattern represented by u'.

Uniformly bounded gaps. Let $R = \{v_1, v_2, \ldots\}$, where $|R| = O(\sqrt{lsc \cdot d})$, since we only deal with heavy vertices.

For this case, the $\mathcal{L}_v, \mathcal{L}_\beta, \tau_u$ data structures are used as well as the framework of the solution to DMOG for bounded gaps in sparse graphs, as described in Subsection 2.1. In addition, in order to add vertices that are suffixes of each other to $\mathcal{L}_v$ in a single operation, the following data structures are also used:

1. For all vertices $u \in L$ and $v_i \in R$ such that $e = (u, v_i) \in E$, **a pointer** $next(e)$ is set to an edge $e' = (u', v_i)$ where u' is the lowest proper ancestor of u in T such that there is an edge from u' to v_i. If no such vertex u' exists then $next(e) = null$. All these pointers can be added in linear time, and their space usage is linear.
2. For each vertex $u \in L$, **an array** A_u of size $|R|$ is built, where $A_u[i]$ is defined as follows. If $e = (u, v_i) \in E$, then $A_u[i]$ points to $e = (u, v_i)$. Similarly, if there is no edge (u, v_i) then the entry of $A_u[i]$ points to the edge (u', v_i) where u' is the lowest proper ancestor of u in T such that there is an edge from u' to v_i. If no such edge exists then $A_u[i] = null$. The list of all ancestors of u in T that have edges touching v_i is obtained through the $next(\cdot)$ pointers.

The A_u arrays are constructed by filling $A_u[i]$ in increasing order of the length of the u vertices, thus when (u, v_i) is not an edge in E, $A_u[i]$ is filled according to $A_{u'}[i]$, where u' is a proper ancestor of u and $A_{u'}$ was already filled. In order to reduce space usage of the A_u arrays, T is partitioned (by greedily peeling small subtrees in the bottom of T) into $O(\sqrt{\frac{d}{lsc}})$ small subtrees such that each subtree's vertices have total degree of $\Theta(\sqrt{lsc \cdot d})$, except for possibly the subtree containing the root of T. The roots of the subtrees are denoted as *special vertices* and the A_u arrays are constructed during preprocessing time only for them. In case $A_u[]$ is needed for a non-special vertex u, $A_u[]$ is constructed during the query in $O(\sqrt{lsc \cdot d})$ time, by initializing $A_u[i]$ to a pointer to e for every $e = (u, v_i) \in E$ and all other $A_u[i]$ entries to *null*. Then, the closest ancestor of u in T, u' is considered, while it is not a special vertex, and for its edges $e' = (u', v_i) \in E$, *null* entries of $A_u[i]$ are set to point to e'. We stop proceeding to the next closest ancestor u' in T, when u' is a special vertex; at this point all *null* entries of $A_u[i]$, are set $A_u[i] = A_{u'}[i]$.

Time and space complexity: [5,6] show that the DMOG problem with one gap and uniform gap borders can be solved with $O(\mathfrak{D})$ preprocessing time, $O(lsc + \sqrt{lsc \cdot d} + op)$ time per text character, and $O(\mathfrak{D} + lsc(\beta + M))$ space.

Non-uniformly bounded gaps. Recall that for the non-uniform gaps, the gap boundaries of an edge e are denoted by α_e and β_e. Since G_D is a multi-graph, each edge is now uniquely represented by a quadruplet $e = (u, v_i, \alpha_e, \beta_e)$. For the case of dense graphs and unbounded gaps, [5,6] used different data structures:

1. For each $e = (u, v_i, \alpha_e, \beta_e) \in E$, **an array** $next_e$ of size $\beta_e - \alpha_e + 1$ is maintained, where for $\alpha_e \leq j \leq \beta_e$, $next_e[j]$ points to an edge $e' = (u', v_i, \alpha_{e'}, \beta_{e'})$, $e' \neq e$, where u' is the lowest ancestor of u in T (possibly u itself) such that there is an edge $e' = (u', v_i, \alpha_{e'}, \beta_{e'})$ where $\alpha_{e'} \leq j \leq \beta_{e'}$ and the pointers $next_e[j]$ do not form a loop. If no such edge exists then $next_e[j] = null$. (For simplicity, the indices of the array are treated as starting from α_e and ending at β_e, where in the pseudo-code we should explicitly reduce α^* when referring to an index in the arrays.)
2. For each pair of vertices $u \in L$ and $v_i \in R$, **an array** $W_{u,i}$ of size $\beta^* - \alpha^* + 1$ is maintained. If $e = (u, v_i) \in E$, then $W_{u,i}[j]$ is a pointer to e if its gap boundaries include j, and to $next_e[j]$ otherwise. The remaining entries of $W_{u,i}[j]$ are null.
   The $W_{u,i}$ arrays replace the $A_u[i]$ arrays that are used for the uniformly-bounded gaps case, and their purpose is basically the same – to enable reporting all edges (u', v_i) (dictionary patterns) that are known to occur due to occurrence of the edge (u, v_i) upon arrival of a heavy vertex $v_i \in R$, where u' is a suffix of u. Recall, however, that in the non-uniform case each edge has its own gap boundaries. Therefore, an occurrence of an edge (u, v_i) does not necessarily imply an occurrence of (u', v_i) due to non-overlapping gap intervals. Thus, the $W_{u,i}$ arrays consider separately each possible gap value j in the range between α^* and β^* to determine the occurrence implication relations.
3. For each vertex $v_i \in R$, **a cyclic *active window* array** AW_i of size $\beta^* - \alpha^* + m_{v_i} + 1$ is maintained, where $AW_i[j]$ is a pointer to a list of lists of edges that all need to be reported if v_i appears in $j - 1$ time units from now.
4. A list $\mathcal{L}_{\alpha^*}$ of delayed vertices (only the longest subpattern arriving at each time unit) $u \in L$ for exactly α^* time units before they are considered. After this time the vertex is inserted to each of active windows arrays AW_i. The use of this list is only to reduce the total space of the active windows.

The construction of the $W_{u,i}$ arrays is similar to the construction of A_u arrays described in the previous paragraph. The $W_{u,i}$ arrays are filled in preorder traversal over T, thus when (u, v_i) is not an edge in E, $W_{u,i}$ is filled according to $W_{u',i}$, where u' is a proper ancestor of u and $W_{u,i}$ was already filled. In order to reduce space usage of the $W_{u,i}$ arrays, T is partitioned (by greedily peeling small subtrees in the bottom of T) into $O(\sqrt{\frac{d}{lsc}})$ small subtrees such that for each subtree, the total number of entries $W_{u,i}[j] \neq null$ for all the subtree's vertices is $\Theta(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$, except for possibly the subtree containing the root of T. The roots of the subtrees are denoted as *special vertices* and the $W_{u,i}$ arrays are constructed during preprocessing time only for them. In case $W_{u,i}$ is needed for a non-special vertex u, $W_{u,i}$ is constructed during the query in $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$ time, by initializing $W_{u,i}[j]$ to a pointer to e for every $e = (u, v_i) \in E$, where $\alpha_e \leq j \leq \beta_e$, and all other $W_{u,i}[j]$ entries to *null*. Then, the closest ancestor of u in T, u' is considered, while it is not a special vertex, and for its edges $e' = (u', v_i) \in E$, *null* entries of $W_{u,i}[j]$ are set to point to e' if $\alpha_{e'} \leq j \leq \beta_{e'}$. The proceeding to the next closest ancestor in T terminates when u' is a special vertex, where all *null* entries of $W_{u,i}[j]$, are set $W_{u,i}[j] = W_{u',i}[j]$.

Time and space complexity: [5,6] show that the DMOG problem with non-uniform gap borders can be solved with $O(\mathfrak{D} + d(\beta^* - \alpha^*))$ preprocessing time, $O(lsc + \sqrt{lsc \cdot d}(\beta^* - \alpha^*) + op)$ time per query text character, and $O(\mathfrak{D} + d(\beta^* - \alpha^*) + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + \alpha^*)$ space.

Remark: Since the parameter $\mathfrak{D}$ dominates the time and space complexity whenever it appears in all algorithms, we only write $O(\mathfrak{D})$ for the preprocessing time complexity in Table 1.

3. Solving online Dictionary Recognition with One Gap

Our solution follows the framework of [5,6] showing that it is possible to make necessary changes in their algorithms in order to solve the DROG problem. Recall that the definition of the DROG problem requires reporting only a single appearance of each gapped pattern in the dictionary D, where each gapped pattern is represented as an edge in the bipartite graph G_D.

Our basic idea is, therefore, quite simple. In order to avoid repetitious reports of an edge (u, v), we delete it from the graph after the first time an edge is reported. We thus assure that u will not be inserted to the data structures maintaining v's responsible neighbors again. Avoiding the consideration of vertices whose associated edges are all reported, is done by adding two counters to each vertex v, $InDeg$ and $OutDeg$, which are initialized with the number of responsible neighbors of v (the in-degree of v) and with the number of neighbors that v is responsible for (the out-degree of v), respectively. Each report and deletion of an edge (u, v) implies a decrease in $OutDeg(u)$ and in $InDeg(v)$ or vice versa according to the orientation of the edge.

Nevertheless, this simple idea is not enough. We should distinguish between two different situations: (1) avoiding further information insertions to the data structures for edges that are already reported, and (2) avoiding redundant report of edges due to information that have already been inserted to the data structures prior to the report of the edge. Deleting an edge from the graph only prevents the first situation, i.e., considering it if its subpatterns appear later in the text. However, it does not take care of the second situation, i.e., it cannot guarantee prevention of additional report if the edge's subpatterns

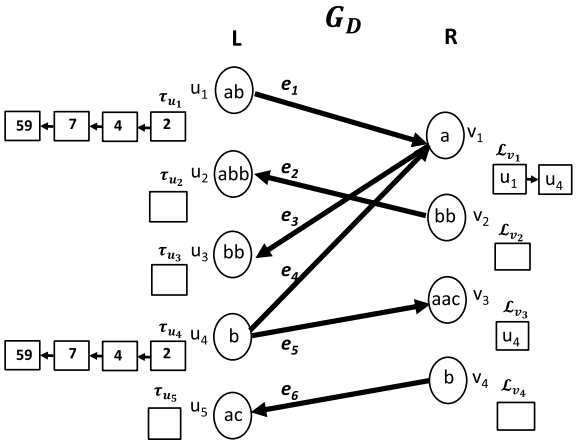

Fig. 1. Data structures used for the uniformly bounded gaps case of the DROG algorithm for sparse graphs, assuming that the arriving text characters of T are $ababaab\ldots ab$, where the first 7 characters are followed by a sequence of 50 other unspecified characters. Note that ab occurs at time units 2,4,7,59, and each such occurrence is also an occurrence of b.

already implicitly or explicitly appear several times in the data structures. Thus, the remaining task we should carefully take care of, is assuring that for every edge (u, v), u appears only once in the data structure of v, so that reporting the edge will be unique even if v occurs again. This is a nontrivial task that necessitates a deep understanding in full detail of the different baseline algorithms we intend to modify, since in some cases it is not possible to save a single copy of u in the data structure of v, otherwise, we might miss an occurrence of a gapped pattern. Therefore, the deletion of the edge (u, v) may require a careful deletion of additional occurrences of u from the data structure associated with v. This should be done without causing an unbearable overhead in the time complexity.

In the following subsections we consider the four solutions described in Section 2, giving for each of them a tailored adaptation for the online *Dictionary Recognition with One Gap* problem.

3.1. DROG for sparse graphs

Uniformly bounded gaps. Following the framework described in Subsection 2.1, we use the $\mathcal{L}_v$ lists to maintain at most a single appearance of the responsible neighbors of v. Hence, when going over the list and reporting edges in case v occurred, each edge is reported once without scanning the τ_u list of u appearance times except for a single gap boundaries validation. Therefore, the solution of the DROG problem for uniformly bounded gap borders is identical to the solution for DMOG with the additional task of deleting an edge $e = (u, v)$ from G_D, deleting u from $\mathcal{L}_v$ and decreasing by one the appropriate degree counters, after the report of the appearance of e. In addition, we apply the algorithm only on vertices whose $InDeg$ and $OutDeg$ values are non zero. Fig. 1 depicts the G_D graph built for the dictionary $D = \{ab\{\alpha, \beta\}a, b\{\alpha, \beta\}a, b\{\alpha, \beta\}aac, ac\{\alpha, \beta\}b, bb\{\alpha, \beta\}a, abb\{\alpha, \beta\}bb\}$, τ_u – the time stamps lists of occurrences in T, and $\mathcal{L}_v$ – the occurred responsible neighbors lists. The modified algorithm appears in Fig. 2.

This gives Theorem 4.

Theorem 4. *Online DROG for dictionary D with uniformly bounded gap borders can be solved in $O(\mathfrak{D})$ preprocessing time, $O(lsc \cdot \delta(G_D) + op^*)$ time per arriving text character where op^* is the number of distinct unreported dictionary patterns which are reported due to a character arrival, and $O(\mathfrak{D} + lsc \cdot (\beta + M))$ space.*

Proof. *Correctness:* The *AC* automaton built for the dictionary subpatterns in the preprocessing stage detects every subpattern $u \in L$ or $v \in R$ recognized at time t, where at each time t at most *lsc* subpatterns that form a suffix chain can be detected. When a subpattern $u \in L$ is detected it is inserted to $\mathcal{L}_\beta$ in lines 15-16 only if the vertex still has edges touching it, i.e., dictionary patterns that were not reported.

When a subpattern $v \in R$ is detected, all newly recognized dictionary patterns that have not been output before are output as follows. In lines 2-8 new edges (u, v) for which v is an assigned neighbor are reported and immediately deleted to block future reports of these edges. Note that lines 21-22 of the algorithm ensure that only one copy of such edge of a responsible vertex u that arrived appears in the $\mathcal{L}_v$ data structure to prevent double report of the same edge in lines 2-8. In lines 9-14 new edges (u, v) for which v is a responsible neighbor are reported and immediately deleted to block future reports of these edges.

In addition, at time t each vertex $u \in L$ that arrived α times units before time t and was inserted to $\mathcal{L}_\beta$ in order to delay its treatment until the minimum gap bound is reached, is treated in lines 17-23 of the algorithm. This treatment is only executed if the vertex still has edges touching it, and includes: (1) enabling a future report of the edges of which u is a responsible neighbor in lines 18-22, in case such edges are left, by adding the vertex to the $\mathcal{L}_v$ data structures of each

```
Uniformly Bounded Gaps DROG Algorithm for Sparse Graphs
When a vertex x arrives at time t (due to a text character arrival) do
1   if x is v ∈ R with InDeg(v) ≠ 0 or OutDeg(v) ≠ 0
2       if InDeg(v) ≠ 0
3           u ← L_v.first
4           while τ_u.first ≥ t − m_v − β − 1 do
5               Report edge (u, v)
6               Delete (u, v) from E
7               Decrease OutDeg(u) and InDeg(v) by one
8               u ← L_v.next
9       if OutDeg(v) ≠ 0
10          Scan edges (u, v) for which v is a responsible-neighbor
11              if τ_u.first ≠ null and τ_u.first ≥ t − m_v − β − 1
12                  Report edge (u, v)
13                  Delete (u, v) from E
14                  Decrease InDeg(u) and OutDeg(v) by one

15  if x is u ∈ L with InDeg(u) ≠ 0 or OutDeg(u) ≠ 0
16      Insert u into L_β

17  for vertices u ∈ L in L_β arriving exactly α time units before time t
            with InDeg(u) ≠ 0 or OutDeg(u) ≠ 0 do
18      if OutDeg(u) ≠ 0
19          for each assigned-neighbor v such that e = (u, v) ∈ E do
20              L_v.first ← u
21              if τ_u ≠ null
22                  Remove the older appearance of u from L_v
23      τ_u.first ← t − α

24  for vertices u ∈ L in L_β arriving exactly β + M + 1 time units
            before time t do
25      Remove u from L_β
26      Remove t − β − M − 1 from the end of τ_u
27      if τ_u = null
28          for each assigned-neighbor v, such that e = (u, v) ∈ E do
29              Delete u from L_v
```

Fig. 2. The DROG algorithm for dealing with uniformly bounded gaps dictionary represented by a sparse graph.

its assigned neighbors v (with duplications avoidance in lines 21-22), and (2) adding the time stamp of its arrival $t-\alpha$ to its occurrences preserving data structure τ_u in order to enable the check of gap borders validity of a future edge before reporting in lines 4 and 11.

Also, at time t each vertex $u \in L$ that arrived $\beta + M + 1$ times units before time t and was inserted to $\mathcal{L}_\beta$ in order to ensure its treatment only within gap borders, is treated again in lines 24-29 of the algorithm. Since the maximum gap bound is reached for these vertices, they are removed from the active window list $\mathcal{L}_\beta$ as well as from their occurrence preserving data structure τ and all $\mathcal{L}_v$ data structures of their responsible neighbors v.

Time complexity: The preprocessing includes linear time on the dictionary size for the construction of the *AC* automaton and the graph G_D as well as its orientation exactly as in [6], and is, therefore, the same.

As for the query time: at each time unit t, there are at most *lsc* vertices for which lines 1,15,17 and 24 are executed. In lines 2-8 of the algorithm only new previously unreported edges are reported, therefore, the total cost of these lines execution for all vertices arriving at time t is op^*, as claimed. In lines 9-14 of the algorithm the cost is $O(1)$ for each edge, for a total of at most $\delta(G_D)$ edges per vertex. The same holds for the work done at lines 18-22, where the removal of older appearance in line 22 and line 29 can easily be done in $O(1)$ time by adding a pointer from u to each of its (single) occurrences in the $\mathcal{L}_v$ data structures of its assigned neighbors when these occurrences are added to them at line 20. Thus, the total time complexity at each time unit t is $O(lsc \cdot \delta(G_D) + op^*)$, as claimed.

Space complexity: Since the data structures we use are the same as in [6] and in worst case their space complexity remains unchanged, the theorem is proven. □

Non uniformly bounded gaps. In this case, for every vertex $v \in R$, the data structure S_v supporting 4-sided 2-dimensional orthogonal range reporting queries saves the occurrences of responsible neighbors of v, as in Subsection 2.1. Yet, the deletion of a reported edge from G_D is not sufficient in order to assure a unique report of an edge occurrence, since the same vertex

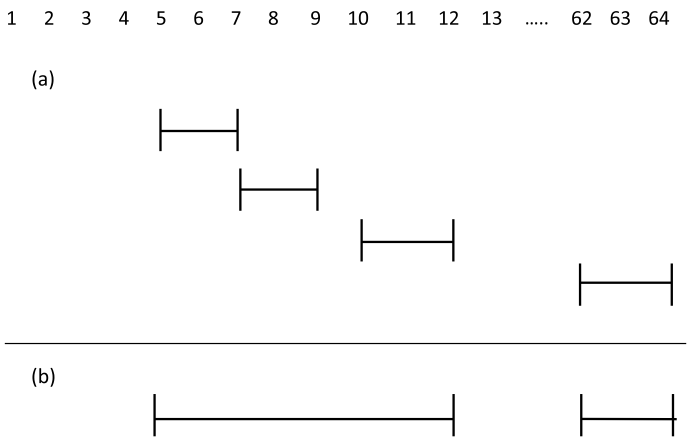

Fig. 3. Consider $e_1 = (ab\{2-4\}a)$ and $\tau_{ab} = \{2, 4, 7, 59\}$. (a) Depicts the time intervals where points exist in S_a due to occurrences of ab from the time they are inserted until they are deleted in the DMOG algorithm. (b) Depicts the time intervals where points exist in S_a due to occurrences of ab in the DROG algorithm. Note that here a new point replaces an older point, so that a single point is reported due to an occurrence of a.

u can be represented by several points in a certain S_v data structure due to several arrival times. If several points are within the query bounds, the range query will return all these occurrences of the edge (u, v), and this query time requirement is relative to the number of points in the range, thus more than op^*.

In order to avoid such redundant reports, and prevent an increase in the time complexity, we modify the algorithm as follows. When a vertex u arrives at time t, for every assigned-neighbor v, such that $e = (u, v)$, the point $(t+\alpha_e+m_v-1, t+\beta_e+m_v-1)$ is not automatically inserted to the data structure S_v, where m_v is the length of the subpattern represented by vertex v. We suggest uniting every two points representing overlapping or adjacent intervals into a single point. This procedure is delicately performed, as the intervals may be separated again, when one of the occurrences of u represented by an interval becomes irrelevant – when located beyond the gaps bounds. In addition, all distinct intervals associated with the same edge are linked, in order to enable deleting all of them, when the edge is reported due to one of the intervals. An example of the union effect on the appearances of u_1 in Fig. 1 is depicted in Fig. 3. The modified algorithm for the DROG problem appears in Fig. 4.

Theorem 5 follows.

Theorem 5. *Online DROG for dictionary D with non-uniformly bounded gap borders can be solved in $O(\mathfrak{D})$ preprocessing time, $\tilde{O}(\delta(G_D) \cdot lsc + op^*)$ time per text character, where op^* is the number of distinct unreported dictionary patterns reported due to character arrival, and $\tilde{O}(\mathfrak{D} + lsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + lsc \cdot \alpha^*)$ space.*

Proof. *Correctness:* The *AC* automaton built for the dictionary subpatterns in the preprocessing stage detects every subpattern $u \in L$ or $v \in R$ recognized at time t, where at each time t at most lsc subpatterns that form a suffix chain can be detected. When a subpattern $u \in L$ is detected it is inserted to $\mathcal{L}_{\beta^*}$ in lines 15-16 only if the vertex still has edges touching it, i.e., dictionary patterns that were not reported yet.

In addition, at time t each vertex $u \in L$ in $\mathcal{L}_{\beta^*}$ that arrived α^* times units before time t and was inserted to $\mathcal{L}_{\beta^*}$ in order to delay its treatment until the minimum gap bound is reached, is treated in lines 17-25 of the algorithms. This treatment is only executed if the vertex still has edges touching it, and includes:

1. Enabling a future report of the edges of which u is a responsible neighbor in lines 18-24, in case such edges are left, by adding a point representing the time interval on which this occurrence of u will be relevant according to the α_e and β_e gap bound of the edge (u, v) to the S_v data structures of each its assigned neighbors v (with duplications avoidance in lines 22-24 by verifying that an inserted point represents a new time interval in lines 20-21 or uniting the time intervals of an existing point with the time interval of the new occurrence in lines 22-24).
2. Adding the time stamp of its arrival $t - \alpha^*$ to its occurrences preserving data structure τ_u in order to enable the check of gap borders validity of a future edge before reporting in lines 3 and 11.

Also, at time t each vertex $u \in L$ that arrived $\beta^* + M + 1$ times units before time t and was inserted to $\mathcal{L}_{\beta^*}$ in order to ensure its treatment only within gap borders, is treated again in lines 26-34 of the algorithm. Since the maximum gap bound is reached for these vertices, they are removed from the active window list $\mathcal{L}_{\beta^*}$ as well as from their occurrence preserving data structure τ. They are also deleted from all S_v data structures of their responsible neighbors v, updating the point representing u, if additional occurrences of the vertex exist.

When a subpattern $v \in L$ is detected, all newly recognized dictionary patterns that have not been output before are reported as follows. In lines 2-8 new edges (u, v) for which v is an assigned neighbor are reported and immediately

```
Non-Uniformly Bounded Gaps DROG Algorithm for Sparse Graphs
When a vertex x arrives at time t (due to a text character arrival) do
1   if x is v ∈ R with InDeg(v) ≠ 0 or OutDeg(v) ≠ 0
2       if InDeg(v) ≠ 0
3           Perform a range query [0, t] × [t, ∞] over S_v
4           for every edge e = (u, v) represented by an output point do
5               Report e
6               Delete e from E
7               Delete every point associated with e from S_v
8               Decrease OutDeg(u) and InDeg(v) by one
9       if OutDeg(v) ≠ 0
10          Scan edges (u, v) for which v is a responsible-neighbor
11              if τ_u.first ≠ null and τ_u.first ≥ t − m_v − β_e − 1
12                  Report edge (u, v)
13                  Delete (u, v) from E
14                  Decrease InDeg(u) and OutDeg(v) by one

15  if x is u ∈ L with InDeg(v) ≠ 0 or OutDeg(v) ≠ 0
16      Insert u into L_{β*}

17  for vertices u ∈ L in L_{β*} arriving exactly α* time units before time t
            with InDeg(v) ≠ 0 or OutDeg(v) ≠ 0 do
18      if OutDeg(v) ≠ 0
19          for each assigned-neighbor v such that e = (u, v) ∈ E do
20              if τ_u = null or t − α* + α_e + m_v > τ_u.first + β_e + m_v
21                  Insert (t − α* + α_e + m_v, t − α* + β_e + m_v) to S_v
22              else let (x, y) be the last point associated with e in S_v
23                  Delete (x, y) from S_v
24                  Insert (x, t − α* + β_e + m_v) to S_v
25      τ_u.first ← t − α*

26  for vertices u ∈ L in L_{β*} arriving exactly β* + M + 1 time units
            before time t do
27      Remove u from L_{β*}
28      Remove the time stamp t − β* − M − 1 from the end of τ_u
29      for each assigned-neighbor v, such that e = (u, v) ∈ E do
30          if τ_u = null or t − β* − M − 1 + β_e + m_v < τ_u.last + α_e + m_v
31              Delete from S_v the point
                    (t − β* − M − 1 + α_e + m_v, t − β* − M − 1 + β_e + m_v)
32          else let (x′, y′) be the oldest point associated with e in S_v
33              Delete (x′, y′) from S_v
34              Insert (τ_u.last + α_e + m_v, y′) to S_v
```

Fig. 4. The DROG algorithm for dealing with non-uniformly bounded gaps dictionary represented by a sparse graph.

deleted to block future reports of these edges. The strategy of inserting a point $(x, y) = (t' + \alpha_e + 1, t' + \beta_e + 1)$ to S_v when u appears at time t' and $e = (u, v)$, implies that when v appears at time t, the range query $[0, t] \times [t, \infty]$ is performed (line 3 of the pseudo-code) giving all points (x, y) where $x \leq t$ and $t \leq y$. Thus, all points representing overlapping or adjacent time intervals associated with edge e that include the time t are reported. Our method of uniting all these points representing time intervals to a single point representing a single time interval, ensures a single report of e due to an occurrence of v at time t. Note that only points representing overlapping or adjacent identical time intervals related to the same edge are united to a single point, since several possible gaps between the same vertices u and v define distinct edges. Hence, when uniting points representing overlapping time intervals, it is only necessary to consider the start point of a new time interval with the end point of the last time interval included in S_v (line 24 of the pseudo-code). The points representing the time intervals are united according to start point that is less than or equal to the end point of the last time interval of the current edge, as we unite time intervals representing the same edge, thus sharing the same gap boundaries. Saving all the relevant time stamps of vertex u enables updating the time interval boundaries which define the point, when one occurrence of u becomes irrelevant to the gap boundaries.

Additional procedure ensuring a unique report of each edge is the deletion of all existing points representing several time intervals of e in S_v due to later occurrences of u, when e is reported, in order to avoid redundant report of e in future arrivals of v.

In lines 9-14 new edges (u, v) for which v is a responsible neighbor are reported and their immediate deletion aims at preventing a future report of such edges.

Time complexity: The preprocessing includes linear time on the dictionary size for the construction of the *AC* automaton and the graph G_D as well as its orientation exactly as in [6], and is, therefore, the same.

As for the query time: at each time unit t, there are at most *lsc* vertices for which lines 1, 15, 17 and 26 are executed. In lines 2-8 of the algorithm only new previously unreported edges are reported, therefore, the total cost of these lines execution for all vertices arriving at time t is op^* plus the cost of the range query on S_v in line 3. The time complexity of each insertion, deletion and range query on S_v depends on the size of $|S_v|$ in [31], which we use. The range query requires $O(\frac{\log|S_v|}{\log\log|S_v|} + op)$ time, where op is the size of the output of the query, which in our case, is at most d in all queries. Since the logarithmic factor of the range query cost is only implicitly written (in the $\tilde{O}$ notation), the cost is as claimed. In lines 9-14 of the algorithm the cost is $O(1)$ for each edge, for a total of at most $\delta(G_D)$ edges per vertex. The same holds for the work done at lines 18-24, where the insertion and deletion operations on S_v in lines 21, 23, 24 take logarithmic time and finding last point associated with an edge in line 22 and line 32 can easily be done in $O(1)$ time by adding a pointer from u to each of its oldest/last occurrences in the S_v data structures of its assigned neighbors when these occurrences are added to them at lines 21, 24 and 34. Thus, the total time complexity at each time unit t is $\tilde{O}(lsc \cdot \delta(G_D) + op^*)$, as claimed.

Space complexity: Note that in the worst case, where all time intervals represented by the points in the S_v data structures are non-overlapping and no edge is reported, the space complexity is the same as that of the DMOG solution. The space complexity then follows from the one stated in Section 2.1. □

3.2. DROG for dense graphs

In the case of dense graphs where $\delta(G_D) = \Omega(\sqrt{d})$, we follow the suggested *threshold* orientation by [5,6], where a vertex in G_D is defined as *heavy* if it has more than $\sqrt{d/lsc}$ neighbors, and *light* otherwise. Hence, the number of heavy vertices is less than $\sqrt{lsc \cdot d}$. An edge where at least one of its endpoints is light is oriented to leave the light vertex. For such edges the algorithms from Subsection 3.1 are applied.

Edges between two heavy vertices have no directions in the sub-graph of G_D containing only heavy vertices of L and R. The solutions for heavy vertices use a batch scan of all heavy vertices of R when a vertex $u \in L$ arrives, where we treat only one such vertex u at each time unit - the longest, and all its suffixes are treated implicitly at a given time unit. We, therefore, use a single counter $Deg(v)$ for each heavy vertex v. Reporting edges between two heavy vertices is done as follows.

Uniformly bounded gaps. We follow the framework of Subsection 2.2 of constructing the A_u arrays, and inserting $A_u[i]$ to $\mathcal{L}_{v_i}$ when u arrives, where $e = (u, v_i) \in E$. After reporting an edge, it is deleted from G_D, yet, as in the previous subsection, this deletion is not sufficient for preventing repetitious report of the edge e, since additional occurrences of v_i or of u', where the subpattern u is a suffix of the subpattern u', can yield additional reports of e. Nevertheless, by analyzing carefully the problematic scenarios this difficulty can be overcome. The modified algorithm appears in Fig. 6.

Consider the dictionary:

$$\begin{aligned} D = \{&a\{5,20\}a, a\{5,20\}ac, a\{5,20\}bc, aa\{5,20\}bc, aa\{5,20\}d, aa\{5,20\}e,\\ &aab\{5,20\}a, aab\{5,20\}ac, aab\{5,20\}cc, aab\{5,20\}c, bb\{5,20\}ac,\\ &bb\{5,20\}bc, bb\{5,20\}cc, bb\{5,20\}c, b\{5,20\}a, b\{5,20\}ac,\\ &b\{5,20\}cc, b\{5,20\}c, c\{5,20\}bc, c\{5,20\}d, c\{5,20\}e, d\{5,20\}d,\\ &d\{5,20\}e, d\{5,20\}f, e\{5,20\}f, f\{5,20\}f, f\{5,20\}g\} \end{aligned}$$

We have that $d = 27$ and $lsc = 3$. Thus, only the vertices in G_D with degree *more* than $\sqrt{27/3} = 3$ are heavy vertices. The heavy vertices graph, including all relevant data structures and their update after $v \in R$ is recognized, appears in Fig. 5.

This gives Theorem 6.

Theorem 6. *Online DROG for dictionary D containing d gapped patterns with uniform gap borders can be solved in $O(\mathfrak{D})$ preprocessing time, $O(lsc + \sqrt{lsc \cdot d} + op^*)$ time per text character, where op^* is the number of distinct unreported dictionary patterns reported due to the character arriving, and $O(\mathfrak{D} + lsc(\beta + M))$ space.*

Proof. *Correctness*: The *AC* automaton built for the dictionary subpatterns in the preprocessing stage detects every subpattern $u \in L$ or $v \in R$ recognized at time t, where at each time t at most *lsc* subpatterns that form a suffix chain can be detected. Since edges with at least one light vertex are taken care of by the algorithms of Subsection 3.1, that are proven to be correct by Theorem 4, we need only show the correctness of the algorithm in Fig. 6 for taking care of edges with heavy vertices on both endpoints. In this algorithm, when a (a heavy vertex) subpattern $u \in L$ is detected, only the vertex of the longest subpattern arriving at time t that still has edges touching it (dictionary patterns that were not reported) is inserted to $\mathcal{L}_\beta$ in lines 11-12.

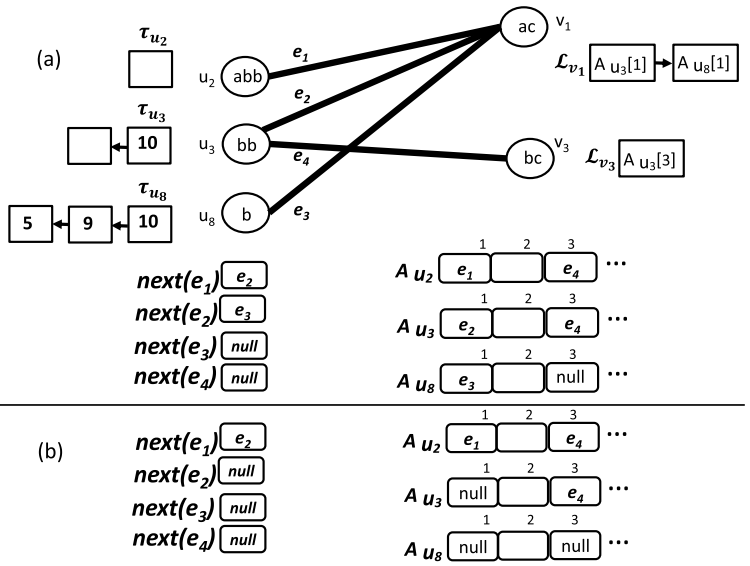

Fig. 5. Consider the text $T = aaaabaaabbaaaaaac$. (a) Depicts the heavy vertices graph and all relevant data structures, when the subpattern bb was recognized at time unit 10 and subpattern b was recognized at time units 5, 9, $\alpha = 5$, $\beta = 20$. (b) Depicts the updated *next* and $A_u[i]$ arrays after subpattern ac was recognized at time unit 18.

UNIFORMLY BOUNDED GAPS DROG ALGORITHM FOR DENSE GRAPHS
When a vertex x arrives at time t (due to a text character arrival) **do**
1 **if** x is $v_i \in R$ with $Deg(v_i) \neq 0$
2 **for** every $A_u[i] \in \mathcal{L}_{v_i}$ where $A_u[i]$ contains a pointer to edge e **do**
3 **if** $\tau_u.first \geq t - m_{v_i} - \beta - 1$
4 **while** $e = (u, v_i) \neq null$ **do**
5 Report edge e
6 Delete e from E
7 $A_u[i] \leftarrow null$
8 Delete $A_u[i]$ from $\mathcal{L}_{v_i}$
9 Decrease $Deg(u)$ and $Deg(v_i)$ by one
10 $e \leftarrow next(e)$

11 **if** x is the longest arriving subpattern vertex $u \in L$ with $Deg(u) \neq 0$
12 Insert u into $\mathcal{L}_\beta$

13 **for** vertex $u \in L$ in $\mathcal{L}_\beta$ arriving exactly α time units before time t with $Deg(u) \neq 0$ **do**
14 **for** each v_i, where $A_u[i] \neq null$ **do**
15 $\mathcal{L}_{v_i}.first \leftarrow A_u[i]$
16 **if** $A_u[i]$ was already in $\mathcal{L}_{v_i}$
17 Remove the older copy of $A_u[i]$ from $\mathcal{L}_{v_i}$
18 **for** every vertex u' a suffix of the subpattern u with $Deg(u') \neq 0$
19 $\tau_{u'}.first \leftarrow t - \alpha$

20 **for** vertex $u \in L$ in $\mathcal{L}_\beta$ arriving $\beta + M + 1$ units before time t with $Deg(u) \neq 0$ **do**
21 Remove u from $\mathcal{L}_\beta$
22 Remove the time stamp $t - \beta - M - 1$ from τ_u
23 **if** $\tau_u = null$
24 **for** every v_i, where $A_u[i] \neq null$ **do**
25 Delete $A_u[i]$ from $\mathcal{L}_{v_i}$
26 **for** every vertex u' a suffix of the subpattern u with $Deg(u') \neq 0$ **do**
27 Remove the time stamp $t - \beta - M - 1$ from $\tau_{u'}$

Fig. 6. The DROG algorithm for dealing with heavy vertices in a uniformly bounded gaps dictionary represented by a dense graph.

At time t the only heavy vertex $u \in L$ that arrived α times units before time t and was inserted to $\mathcal{L}_\beta$ in order to delay its treatment until the minimum gap bound is reached, is treated in lines 13-19 of the algorithm, if it exists. This treatment is only executed if the vertex still has edges touching it, and includes: (1) a batch scan of all heavy vertices of R to enable a future report of the edges between u and all its suffixes u' and its heavy neighbors in lines 14-17, in case such edges are left, by adding $A_u[i]$ to the $\mathcal{L}_{v_i}$ data structures of each of its heavy neighbors v_i (with duplications avoidance in lines 16-17), and (2) adding the time stamp of its arrival $t - \alpha - 1$ to its occurrences preserving data structure τ_u as well as to each $\tau_{u'}$ data structures of its suffixes u' that arrived together with u, in order to enable the check of gap borders validity of a future edge before reporting in line 3.

In order to ensure its treatment within gap borders, at time t the only vertex $u \in L$ that arrived $\beta + M + 1$ times units before time t and was inserted to $\mathcal{L}_\beta$ (if exists), is treated again in lines 20-27 of the algorithm. Since the maximum gap bound is reached for this vertex, it is removed from the active window list $\mathcal{L}_\beta$ (line 21) as well as from its occurrence preserving data structure τ_u (line 22) and, in case it is the only occurrence of u, its $A_u[i]$ is also removed from all $\mathcal{L}_{v_i}$ data structures of its heavy neighbors v_i using a batch scan of all heavy vertices in R in lines 23-25. Its arrival time stamp is explicitly removed from the occurrence preserving data structures $\tau_{u'}$ for each of its suffixes u' that arrived together with u in lines 26-27.

When a subpattern $v_i \in R$ is detected, all and only new previously unreported, recognized dictionary patterns are reported as follows. In lines 2-10 new edges (u, v_i) and (u', v_i) for each suffix u' of u are reported and immediately deleted to block future reports of such edges. The report of an edge (u, v_i) is done using the $A_u[i]$ data structure containing this edge, that was explicitly inserted to $\mathcal{L}_{v_i}$ for u the longest arriving subpattern at a certain time. The report of any edge (u', v_i) is done using the *next* data structure in the while loop in lines 4-10. Note that lines 15-17 of the algorithm ensure that only one copy of $A_u[i]$ appears in the $\mathcal{L}_{v_i}$ data structure, where u is a heavy vertex that arrived within the gap bounds (which is verified in line 3 of the pseudo-code) in order to prevent double report of the same edge in lines 2-10. The deletion of $A_u[i]$ from $\mathcal{L}_{v_i}$ after reporting $e = (u, v_i)$ prevents another report of e upon additional occurrences of v_i.

By adding the requirement of overriding $A_u[i]$ by *null* in line 7, we prevent additional report of e, either by another occurrence of u or by an occurrence of u', where the subpattern u is a suffix of the subpattern u'. In the latter case, if the subpattern u' occurred after the subpattern u in the text, $\mathcal{L}_{v_i}$ contains $A_{u'}[i]$. Thus, $e = (u, v_i)$ is included in the list derived from $A_{u'}[i]$ (Recall that by the definition of the data structures $A_{u'}[i]$ and *next*, the list of all ancestors of u' in T that have edges touching v_i is obtained through the $next(\cdot)$ pointers in the while loop in lines 4-10 of the algorithm). When v_i occurs within a proper gap from the occurrence of u', all the edges in the list derived from $A_{u'}[i]$ are reported. Nevertheless, in our algorithm the array $A_u[i]$ associated with e is checked to be non *null* before reporting edge e. If $A_u[i] = null$ the report of the edges derived from the occurrence of u' is terminated, as a *null* implies that all edges of u and its suffixes with v_i were already reported, if they existed. In case u' occurred before u, the edge $e = (u, v_i)$ is reported by the list derived from $A_{u'}[i]$, yet our algorithm implies deletion of $A_u[i]$ from $\mathcal{L}_{v_i}$ for every reported edge $e = (u, v_i)$ and clearing it (by assigning *null* to the entry), so a successive occurrence of u is not checked again with regard to v_i as $A_u[i] = null$. Hence, only a single occurrence of (u, v_i) is reported and every new yet unreported edge will be reported.

It is important to note that, the A_u data structures are not fully constructed during preprocessing stage, as mentioned in Subsection 2.2, in order to reduce the space complexity, using the definition of *special* L vertices which are the only vertices constructed in preprocessing and the rest of the A_u arrays are constructed during query time. Nevertheless, the fact that we clear A_u entries preserve the correctness of these structures construction and content validity. This is because the A_u construction is done according to a vertex-to-root path in the tree structure T of the suffix relations between the L subpatterns, where the order of such path is that a parent is a suffix of its children. Since we only clear entries of reported edges and at each such report all edges of suffixes must also be reported and deleted, we will have no future need of these entries for the construction of other entries.

Time complexity: First, note that our data structures are the same as those of [5,6] and only our query algorithm is different. Therefore, the preprocessing time (and space) is the same and can be taken as a black box from Subsection 2.2.

Regarding the query algorithm time: The treatment of each of the *lsc* vertices $v_i \in R$ arriving at time t in lines 1-10 takes $O(\sqrt{lsc \cdot d} + op^*)$, where op^* is the number of new distinct dictionary patterns reported due to the character arriving, as follows. Since $\mathcal{L}_{v_i}$ contains at most one copy $A_u[i]$ for each heavy vertex $u \in L$ (this is ensured in lines 16-17), the for-loop in line 2 takes $O(\sqrt{d/lsc})$ (as this is the maximum number of heavy vertices). In each iteration the check in line 3 takes $O(1)$ time and the while-loop in lines 4-10 is executed only for new unreported yet edges, where each while iteration takes $O(1)$ time, for a total of $O(op^*)$. Thus, the total time for treatment of vertices in R arriving at time t is: $O(\sqrt{lsc \cdot d} + op^*)$, as claimed.

In addition, the treatment of the single longest subpattern vertex $u \in L$ that may arrive at time t takes $O(1)$ time for the insertion to $\mathcal{L}_\beta$ in lines 11-12. The handling of the active window $\mathcal{L}_\beta$ in lines 13-19 and 17-20 takes $O(lsc + \sqrt{lsc \cdot d})$, as follows. The batch scan of all heavy vertices in R in lines 14-17 and lines 24-25 is bounded by the number of such vertices, which is $\sqrt{d/lsc}$. Note that, retrieving $A_u[i]$ in $\mathcal{L}_{v_i}$ can be done in constant time if we keep a pointer to each single copy of $A_u[i]$ in τ_u when it is inserted to $\mathcal{L}_{v_i}$ in line 15. Executing lines 18-19 and 26-27 takes $O(lsc)$ time, and the rest of the operations take constant time. Therefore, the total query time is as claimed.

Space complexity: The space complexity includes the space of all data structures used, which are the same as in [5,6]. Therefore, the space complexity can be taken as a black box from Subsection 2.2. Also note, that the time and space

complexity for dealing with light vertices is given by Theorem 4 by replacing the term $\delta(G_D)$ with the bound on the degree of light vertices, which is $\sqrt{d/lsc}$, and the fact that $lsc \cdot \sqrt{d/lsc}$ is $\sqrt{lsc \cdot d}$. □

Non-uniformly bounded gaps. For the case of non uniformly bounded edges we use the data structures $next_{u,v_i}$, $W_{u,i}$, AW_i and $\mathcal{L}_{\alpha^*}$ described in Subsection 2.2. Following our core idea, after reporting an edge $e = (u, v_i)$ it is deleted from G_D and $W_{u,i}[j]$ should be assigned *null* for $\alpha_e \leq j \leq \beta_e$, so that additional occurrence of u will not be inserted to the active window of v_i, AW_i. In addition, other occurrences of u already in AW_i may derive a repetitious report of e due to an additional occurrence of v_i. In the case of uniformly bounded gaps, we handled the problem by deleting all appearances of $e = (u, v_i)$ from the data structure $\mathcal{L}_{v_i}$ of located neighbors of v_i. However, dealing with non-uniformly bounded gaps is more complicated, since when edge (u, v_i) is reported, some edges $e' = (u', v_i)$, where u' is a subpattern that is a suffix of the subpattern u, are reported too while other such e's are not reported due to different gap boundaries. Consequently, we cannot automatically delete all appearances of u from $W_{u,i}[j]$ and from the data structure AW_i of located neighbors of v_i, since it may cause misses of occurrences of some suffixes of u, as a vertex occurrence implies all the suffixes of the subpattern vertex occurred as well, therefore deleting the occurrence of u causes the suffixes of u to have no indication of appearance.

We therefore suggest a delicate deletion, which after reporting edge $e = (u, v)$ with gap of size j, for some j such that $\alpha_e \leq j \leq \beta_e$, assigns $W_{u,i}[j]$ to *null* and replaces for $f \neq j$ the content of $W_{u,i}[f]$ by the longest suffix of the subpattern u, u', where $e' = (u', v_i) \in E$, e' appears in the list emanating from $next_{u,v_i}[f]$ and e' has not been reported yet. The detailed algorithms appears in Fig. 7.

Fig. 8 depicts a heavy vertices graph, including all relevant data structures. Fig. 9 depicts only the modified data structures of a heavy $u \in L$ data structure after a corresponding heavy $v \in R$ was recognized.

Theorem 7 follows.

Theorem 7. *Online DROG for dictionary D containing d gapped patterns with non-uniform gap borders can be solved in $O(\mathfrak{D} + d(\beta^* - \alpha^*))$ preprocessing time, $O(lsc + \sqrt{lsc \cdot d}(\beta^* - \alpha^*) + op^* \cdot (\beta^* - \alpha^*))$ time per query text character, where op^* is the number of new distinct dictionary patterns reported due to the character arriving, and $O(\mathfrak{D} + d(\beta^* - \alpha^*) + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + \alpha^*)$ space.*

Proof. *Correctness:* The *AC* automaton built for the dictionary subpatterns in the preprocessing stage detects every subpattern $u \in L$ or $v \in R$ recognized at time t, where at each time t at most *lsc* subpatterns that form a suffix chain can be detected. Since edges with at least one light vertex are taken care of by the algorithms of Subsection 3.1, that are proven to be correct by Theorem 5, we need only show the correctness of the algorithm in Fig. 7 for taking care of edges with heavy vertices on both endpoints. In this algorithm, when a (a heavy vertex) subpattern $u \in L$ is detected, only the vertex of the longest subpattern arriving at time t that still has edges touching it (dictionary patterns that were not reported) is inserted to $\mathcal{L}_{\alpha^*}$ in lines 6-7.

At time t the only heavy vertex $u \in L$ that arrived α^* times units before time t and was inserted to $\mathcal{L}_{\alpha^*}$ in order to delay its treatment until the minimum gap bound is reached, is treated in lines 8-13 of the algorithm, if it exists. This treatment is only executed if the vertex still has edges touching it, and includes: (1) removing it from the $\mathcal{L}_{\alpha^*}$ list in line 9, since from now on it will be treated in each of the heavy vertices in R separate active windows, and (2) a batch scan of all heavy vertices of R to enable a future report of the edges between u and all its suffixes u' and its heavy neighbors in lines 10-13, in case such edges are left, by adding for each possible value of time units j between the minimum gap bound α^* and the maximum gap bound β^* only non-*null* $W_{u,i}[j - \alpha^*]$ entries to the lists of the active windows $AW_i[j + m_{v_i} - \alpha^*]$ data structures of each of its heavy neighbors v_i, where a non-*null* $W_{u,i}[j - \alpha^*]$ entry points to an edge (and the edges derived from it through the *next* arrays pointers) that will be relevant for report in $j + m_{v_i} - \alpha^*$ time units from time t if v_i occurs (recall that m_{v_i} is the length of the subpattern v_i).

Note that, in order to ensure the treatment of u (and all its suffixes) only within gap borders, at time t the only vertex $u \in L$ that arrived $\beta^* + m_{v_i} + 1$ times units before time t and was inserted to the active windows AW_i for each v_i (if exists), is removed from all of them by clearing each $AW_i[\beta^* - \alpha^* + m_{v_i} + 1]$ in line 16 of the algorithm (where we normalize the index to the size of the active windows by subtracting α^*). Line 15 ensures that the active windows are cyclically shifted by one position at each time unit, so that the occurrences that become relevant at time t are in the entries $AW_i[1]$ of the active windows of each v_i.

For each of the at most *lsc* vertices $v_i \in R$ arriving at time unit t, the list of the first entry of the active window AW_i is scanned in line 2, if v_i still has edges touching it, and all newly recognized dictionary patterns that have not been output before are output as follows. Note that, since G may be a multi-graph, we identify edges by a quadruplet specifying the endpoints as well as the gap bounds defining each edge. In lines 2-5 new edges $e = (u, v_i, \alpha_e, \beta_e)$ and $e' = (u', v_i, \alpha_{e'}, \beta_{e'})$ for each suffix u' of u are reported are reported and immediately deleted to block future reports. The report of such edges $e = (u, v_i, \alpha_e, \beta_e)$ is done using the pairs $(W_{u,i}, j - \alpha^* + 1)$ that were explicitly inserted to the list in $AW_i[1]$. The first value of each such pair contains $W_{u,i}[j - \alpha^* + 1]$ data structures entry pointing to these edges, and the actual report is performed using an explicit call to the procedure Report&Update in line 5 for each such non-*null* entry. It is verified in line 3 that $W_{u,i}[j - \alpha^* + 1]$ is not *null*, since *null* value means this edge was already reported by time t (even though it was non-*null* during its insertion to $AW_i[j + m_{v_i} - \alpha^* + 1]$ $j + m_{v_i} - \alpha^* + 1$ time units before time t). The report of any edge

```
Non-Uniformly Bounded Gaps DROG Algorithm for Dense Graphs
When a vertex x arrives at time t (due to a text character arrival) do
1   if x is v_i ∈ R with Deg(v_i) ≠ 0
2       for every (W_{u,i}, j) ∈ AW_i[1] do
3           if W_{u,i}[j] ≠ null
4               e ← (u, v_i, α_e, β_e) ← W_{u,i}[j]
5               Report& Update(e, j)

6   if x is the longest arriving subpattern vertex u ∈ L with Deg(u) ≠ 0
7       Insert u into L_{α*}

8   for vertex u ∈ L in L_{α*} arriving exactly α* time units before time t
            with Deg(u) ≠ 0 do
9       Remove u from L_{α*}
10      for each v_i ∈ R with Deg(v_i) ≠ 0 do
11          for j ← α* to β* do
12              if W_{u,i}[j − α*] ≠ null
13                  Add (W_{u,i}[j − α*], j − α*)
                        to list AW_i[j + m_{v_i} − α*]

14  for each v_i ∈ R with Deg(v_i) ≠ 0 do
15      Shift AW_i cyclically one position by incrementing its starting position
16      Clear AW_i[β* − α* + m_{v_i} + 1]

Report&Update(e, j)
17  Report edge e
18  Delete e from E
19  W_{u,i}[j] ← null
20  Decrease Deg(u) and Deg(v_i) by one
21  e' ← next_e[j + α* − α_e]
22  if e' ≠ null and e' ∉ E
23          e' ← next_{e'}[j + α* − α_{e'}]
24  if e' ≠ null
25      e' ← (u', v_i, α_{e'}, β_{e'})
26      if W_{u',i}[j] ≠ null
27          e'' ← (u', v_i, α_{e''}, β_{e''}) ← W_{u',i}[j]
28              Report&Update(e'', j)
29  for ℓ ← α_e to β_e do
30      if ℓ − α* + 1 ≠ j
31          e' ← next_e[ℓ − α_e + 1]
32          if e' ≠ null and e' ∉ E and α_{e'} ≤ ℓ ≤ β_{e'}
33              next_e[ℓ − α_e + 1] ← next_{e'}[ℓ − α_{e'} + 1]
34          if W_{u,i}[ℓ − α* + 1] = e
35                  W_{u,i}[ℓ − α* + 1] ← next_e[ℓ − α_e + 1]
```

Fig. 7. The DROG algorithm for dealing with heavy vertices in a non-uniformly bounded gaps dictionary represented by a dense graph.

$e' = (u', v_i, \alpha_{e'}, \beta_{e'})$ that is also relevant at this time unit considering its own gap boundaries, is done using the *next* data structure during recursive executions of the Report&Update procedure described in lines 17-35.

The Report&Update procedure of an edge $e = (u, v_i, \alpha_e, \beta_e)$ ensures reporting all relevant edges and avoidance of repetitious reports, as follows. After e is reported due to a gap of size $j + \alpha^*$ in line 17 (note that the value of j passed to the procedure is already normalized by subtracting α^*), it is deleted both from G_D (line 18) and from $W_{u,i}[j]$ (line 19). Note that $W_{u,i}[j] = null$ only if all edges $e' = (u', v_i, \alpha_{e'}, \beta_{e'})$, u' a suffix of u and $\alpha_{e'} \leq (\alpha^* + j) \leq \beta_{e'}$, were reported, since lines 21-28 use the $next_e[j]$ entry in line 21 to reveal e' and ensure that every unreported yet (verified by the condition in line 26) such edge e' is reported in the recursive calls for Report&Update in line 28. Note that, since G_D is a multi-graph, the entry $W_{u',i}[j]$ may contain another edge $e'' = (u', v_i, \alpha_{e''}, \beta_{e''})$ (line 27) with the same endpoint vertices as e' but other gap borders (see for example the edges e_1 and e_5 in Fig. 8). In such case, the report reaches e' during the recursive calls and the $next_{e''}[j + \alpha^* - \alpha_{e''}]$ pointer (note that, while the normalization of all W data structures is identical, the normalization of the *next* arrays is by α_e for each edge e, which somewhat complicates the pseudo-code).

Also note, that a reported edge e' may be still reached through *next* pointer of another unreported edge $\hat{e} = (\hat{u}, v_i, \alpha_{\hat{e}}, \beta_{\hat{e}})$, where u' is a suffix of $\hat{u}$, that may occur later (see for example the edges e_1 and e_3 in Fig. 8). We avoid repetitious report of e' during a future report of $\hat{e}$ in line 22-23, in which we verify that e' is a non-reported yet edge checking that it is not already deleted from E, otherwise, we skip to the edge pointed to by $next_{e'}$ that must be a non-reported yet edge due to the update of the *next* array of every reported edge that is performed upon return from the recursive call in lines 29-35.

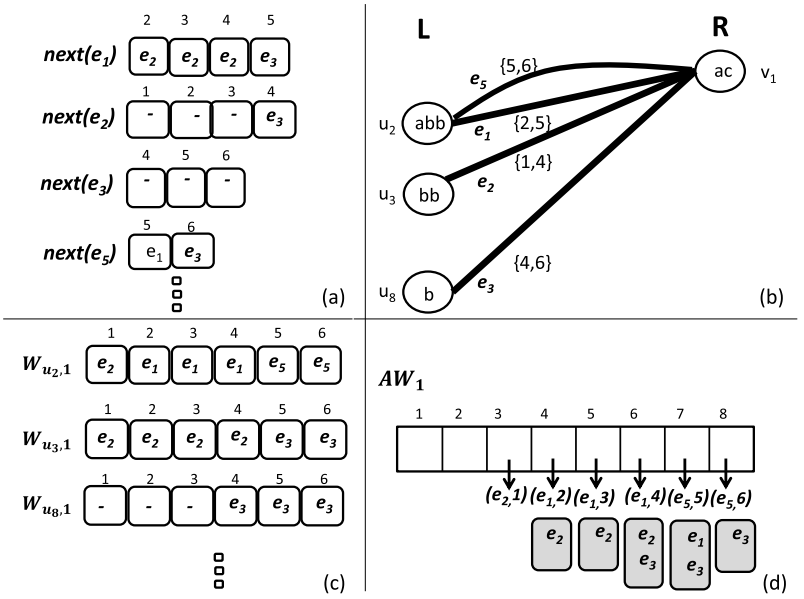

Fig. 8. The *next* arrays shown in (a) are for the dictionary that appears in (b), where for simplicity, the indices in the *next* arrays are non-normalized. The corresponding $W_{u,i}$ arrays appear in (c). The AW_1 data structure after $u_2 = abb$ was found in time 1 appears in (d), and the implicit lists that are saved in AW_1 appear in the gray blocks beneath AW_1. Note, that for the simplicity of the explanation, we focus merely on the occurrence of *abb*, including the implicit occurrence of all subpatterns that are suffixes of *abb*, yet ignoring the occurrence of *b* in the previous time unit.

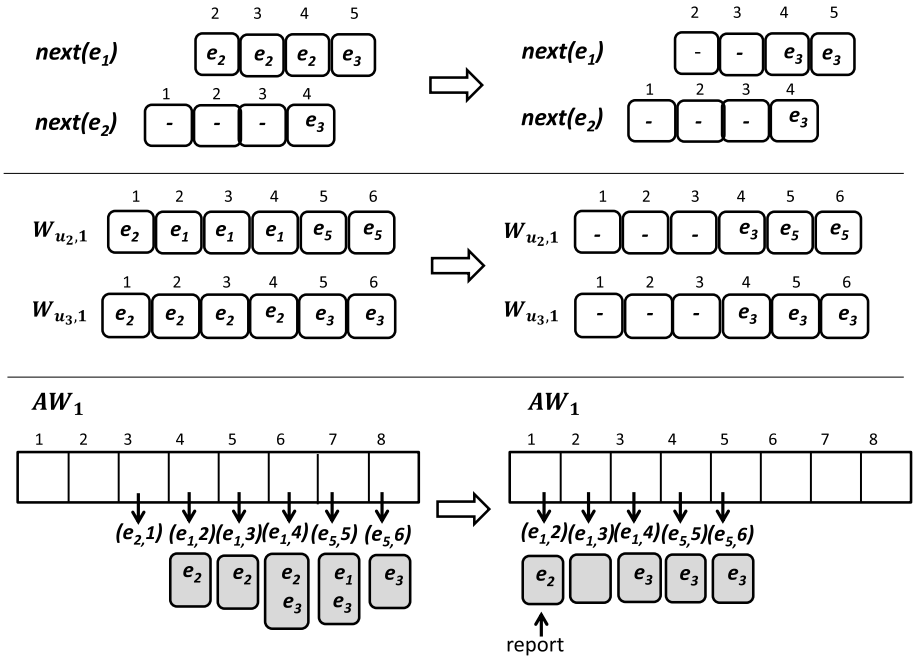

Fig. 9. The modifications required to some of the data structures of Fig. 8 after the occurrence of *ac* 4 time units from the update of AW_1 due to the occurrence of *abb*.

This update is done for every position $\ell - \alpha^* + 1$, $\alpha_e \leq \ell \leq \beta_e$ (except for the jth position that becomes irrelevant since no future report can reach it due to line 19 and the fact that any call to Report&Update is conditioned by a non-*null* value of the appropriate W-entry at position j) in the *next* and W data structures of the reported edge e, and includes: (1) an update of the $next_e$ to skip an already reported edge e', and (2) an update of the $W_{u,i}$ entry to skip the reported edge e. It is important to note that, since the Report&Update procedure is called recursively on the edges reached through the *next* pointer, and these *next* arrays are updated upon return from a the recursive call, the update of both *next* (which is also

recursively defined and built in the preprocessing) and W arrays is correctly performed with top-down use and bottom-up update.

It is also important to note, that the $W_{u,i}$ data structures are not fully constructed during preprocessing stage in order to reduce the space complexity. This is done using the definition of *special* L vertices, which are the only vertices for which the data structures are constructed in preprocessing and the rest of the $W_{u,i}[j]$s are constructed during query time, as mentioned in Subsection 2.2. Nevertheless, the fact that we clear $W_{u,i}[j]$ entries preserves the correctness of these structures' construction and content validity. This is because the $W_{u,i}$ construction is done according to a vertex-to-root path in the tree structure T of the suffix relations between the L subpatterns, where the order of such path is that a parent is a suffix of its children. Since we only clear entries of reported edges and at each such report all edges of suffixes must also be reported and deleted, we will have no future need of these entries for the construction of other entries.

Time complexity: First, note that our data structures are the same as those of [5,6] and only our query algorithm is different. Therefore, the preprocessing time (and space) is the same and can be taken as a black box from Subsection 2.2.

Regarding the query algorithm time: The treatment of each of the *lsc* vertices $v_i \in R$ arriving at time t in lines 1-5 takes $O(\sqrt{lsc \cdot d} + op^* \cdot (\beta^* - \alpha^*))$ time, where op^* is the number of new distinct dictionary patterns reported due to the character arriving, as follows. The worst-case size of the list that $AW_i[1]$ contains is $O(\sqrt{d/lsc})$, since at most one copy of $W_{u,i}$ is inserted in lines 10-13 to one entry of AW_i for each j from α^* to β^* for at most $\sqrt{d/lsc}$ heavy vertices $u \in L$ (a single one each time) arriving in a sequence of $\beta^* - \alpha^*$ time units preceding the $\alpha^* + 1$ time unit before t. Therefore, the for-loop in line 2 takes $O(\sqrt{lsc \cdot d})$ time for all *lsc* vertices $v_i \in R$ at time t, not including the edges report. In each iteration the check in line 3 takes $O(1)$ time.

The Report&Update procedure in lines 17-35 is executed only for new unreported yet edges, where each report in lines 17-28 takes $O(1)$ time and additional $O(\beta^* - \alpha^*)$ time for data structures update in lines 29-35, for a total of $O(op^* \cdot (\beta^* - \alpha^*))$. Thus, the total time for treatment of vertices in R arriving at time t is: $O(\sqrt{lsc \cdot d} + op^* \cdot (\beta^* - \alpha^*))$, as claimed.

The rest of the operations of the query algorithm take $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$ time, as follows. Lines 6-7 take $O(1)$ time. Lines 8-13 take $O(\sqrt{d/lsc}(\beta^* - \alpha^*)$ time not including the construction of the $W_{u,i}$ data structures, if needed. Recall that in order to save space, the construction of part of these structures is postponed to query time in [6], where it is assured that the total construction time of the needed data structures during query time is $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$. Lines 14-16 take $O(\sqrt{d/lsc})$ time. Thus, the total query time is as claimed.

Space complexity: The space complexity includes the space of all data structures used, which are the same as in [5,6]. Therefore, the space complexity can be taken as a black box from Subsection 2.2. Also note, that the time and space complexity for dealing with light vertices is given by Theorem 5 by replacing the term $\delta(G_D)$ with the bound on the degree of light vertices, which is $\sqrt{d/lsc}$, and the fact that $lsc \cdot \sqrt{d/lsc}$ is $\sqrt{lsc \cdot d}$. □

4. 3SUM conditional lower bounds

In this section we show lower bounds for the (offline) unbounded DROG problem conditioned on the 3SUM conjecture. The different versions of DROG, namely, unbounded, uniformly and non-uniformly bounded gaps, form a hierarchy of generalizations, where the non-uniformly bounded is the most general as each dictionary pattern P_i has its own gap boundaries α_i and β_i, the next step in the hierarchy is the bounded case, where $\alpha_i = \alpha$ and $\beta_i = \beta$ for every i, and the most restricted version in the hierarchy is the unbounded case, where $\alpha_i = \alpha = 0$ and $\beta_i = \beta = \infty$ for every i. Setting β or β_i to ∞ actually means setting it to be of size $|T|$ in the offline version or to the (dynamically changing) current overall text size in the online version. This may affect the efficiency but not the correctness of the algorithms. Thus, more general DROG versions can solve (possibly less efficiently) the unbounded DROG version by setting the values of the gap bounds as above accordingly, therefore, the lower bounds hold for the generalized problems as well.

The state of the art technique for specifying formal limitations for data structures problems is to prove *conditional lower bounds* (CLBs) based on the *conjectured* hardness of some problem. A popular conjecture for CLBs is that the 3SUM problem (given n integers determine if any three sum to zero) cannot be solved in truly subquadratic time, where truly subquadratic time is $O(n^{2-\Omega(1)})$ time. The same is conjectured even if the algorithm is allowed to use randomization (see e.g. [36,1,24,17]). We observe that the 3SUM problem can be reduced to DROG similarly to the reduction to DMOG [5,6], which emphasizes that the difficulty of the DMOG problem is inherent and does not stem from the output size.

The path for the reduction is as follows. Kopelowitz, Pettie, and Porat [24] provided efficient reductions of 3SUM to enumerating triangles in a tripartite graph (originally shown by Pătraşcu [36]), thereby showing that many known triangle enumeration algorithms ([22,14,12,23]) are essentially and conditionally optimal, up to subpolynomial factors. The following indexing version of the triangle enumeration problem is a natural extension of the offline problem. In the *vertex-triangles* problem the goal is to preprocess a graph so that given a query vertex u we may list all triangles that contain u. This version solves the triangle enumeration problem, which gives a lower bound conditioned on the 3SUM conjecture. [5,6] use the vertex-triangles problem in order to show that the simplest DMOG version of (offline) unbounded setting is 3SUM-hard.

Our conditional lower bound is stated by the following theorem. Note that, since the lower bounds apply also for randomised algorithms the statements use expected time, where the expectation is taken over the choice of random bits used by the algorithm. We use the term *reporting time* to describe the time spent on output in each time unit. Thus, to understand the statements of the following theorems, when the total query time per time unit of an algorithm can be formulated as

$O(t_q + op \cdot t_r)$ time, we say that t_q is the *query time* per time unit and t_r is the reporting time per time unit. Since we want to prove a lower bound on the time spent on each time unit during the query even if the output is small, it is important that the reporting time is assumed to be sub-polynomial in d (i.e., $o(d)$).

Theorem 8. *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. For any algorithm that solves the DROG problem on a dictionary D with d patterns, if the amortized expected preprocessing time is $O(\mathfrak{D} \cdot \delta(G_D)^{1-\Omega(1)})$ and the amortized expected reporting time is sub-polynomial, then the amortized expected query time must be $\Omega((\delta(G_D))^{1-o(1)})$.*

Proof. The proof begins from the CLB for triangle enumeration stated in [5,6].

Corollary 9. [5,6] *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. Then for any constant $0 < x < 1/2$, any algorithm for enumerating all triangles in a graph G with m edges, $n = \Theta(m^{1-x})$ vertices, and $\hat{d} = \delta(G) = \Theta(m^x)$, where $\hat{d}$ is the average degree of a vertex in G, must spend $\Omega(m \cdot \delta(G)^{1-o(1)})$ expected time.*

The important observation is that the number of triangles in the constructed graph G in the proof of Corollary 9 (see [5,6]) is actually polynomially smaller than $m \cdot \delta(G)$, which means that the lower bound is *not* due to the time required to report the output. [5,6] prove based on Corollary 9, that assuming 3SUM requires $\Omega(n^{2-o(1)})$ expected time, any algorithm that solves the vertex-triangles problem on a graph G with n vertices and m edges, if the amortized expected preprocessing time is $O(m \cdot \delta(G)^{1-\Omega(1)})$ and the amortized expected reporting time is sub-polynomial, then the amortized expected query time must be at least $\Omega((\hat{d} \cdot \delta(G))^{1-o(1)})$, where $\hat{d}$ is the average degree of the queried vertex. [5,6] use a reduction from vertex-triangles problem to a special case of DMOG in which every dictionary subpattern is a single character with unbounded gaps between them. Since the input of the DROG and DMOG problems is identical, we may use the same reduction from vertex triangles to get the special case of DROG with unbounded gaps dictionary, where the subpatterns of each gapped pattern in the dictionary D are single characters. We refer to this special case of DROG as DROG*.

We reduce the vertex-triangles problem to DROG* problem as follows. We convert the input graph $G = (V, E)$ of the vertex-triangle problem to a bipartite graph $G_D = (L \cup R, E_D)$, which uniquely defines the input dictionary D for the DROG* problem, as follows. Every vertex of $u \in V$ has two corresponding vertices in G_D, $u_1 \in L$ and $u_2 \in R$, and every edge $(u, v) \in E$ has two corresponding copies $(u_1, v_2), (v_1, u_2) \in E_D$. We preprocess the graph G_D. Now, to answer a vertex-triangle query on some vertex $u \in V$ of G, we input as the next characters in the query text for DROG* all the neighbors v of u in G, which correspond to a single character representing both $v_1 \in L$ and $v_2 \in R$ of G_D. Thus, there is a one-to-one correspondence between the edges reported by the DROG* algorithm in the output for the query characters representing neighbors vertices $v \in G$ (which are vertices $v_1 \in L, v_2 \in R$ in G_D) of a vertex $u \in G$ and the triangles in the output of the vertex-triangles query for the graph G. Note that we may assume that each vertex is queried once for vertex-triangle queries as the answer for a single vertex does not change over the sequence of queries. Therefore, all required triangles are reported through the reduction. Since each vertex-triangle query costs $\Omega(\hat{d} \cdot \delta(G)^{1-o(1)})$ amortized expected time then the amortized expected time spent for each of the neighbors of u is $\Omega(\delta(G)^{1-o(1)})$, since the average number of neighbors is $\hat{d}$ and the measure is the *amortized* expected time.

The theorem then follows from the CLB for vertex triangles of [5,6], and the fact that it holds no matter what the output size is. □

We also generalize a lower bound of [5,6] stated for uniformly bounded gaps to non-uniform gap borders as follows.

Theorem 10. *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. For any algorithm that solves the non-uniformly bounded DROG/DMOG problems on a dictionary D with d patterns, if the amortized expected preprocessing time is $O(\mathfrak{D} \cdot \delta(G_D)^{1-\Omega(1)})$ and the amortized expected reporting time is sub-polynomial, then the amortized expected time spent on each text character must be at least $\Omega((\beta^* - \alpha^*)^{1-o(1)})$.*

Proof. As in [5,6], we convert the input graph G of the vertex-triangles problem to a tripartite graph G_T by creating three copies of the vertices V_1, V_2, V_3 (denote the copy of vertex v of G in V_i by v_i, for $i = 1, 2, 3$) and for each edge (u, v) in G we add the following 6 edges to G_T: (u_1, v_2), (u_1, v_3), (u_2, v_1), (u_2, v_3), (u_3, v_1), (u_3, v_2). We also add a dummy vertex to G_T with degree 0. Each triangle in G corresponds to a constant number of triangles in G_T. Let α^* be any positive integer and let $\beta^* = \alpha^* + 2\hat{d}$, where $\hat{d}$ is the average degree of vertices in G. In order to solve vertex-triangles queries we use a special case of DROG/DMOG, in which every dictionary subpattern is a single character with unbounded gaps between them (denoted by DROG*/DMOG*). Therefore, each vertex in G_T represents a different character in the dictionary alphabet (where vertices from different vertices sets can have the same character), and each edge in G_T forms an unbounded gapped pattern in the dictionary D.

The query text of the DROG*/DMOG* is built as follows. For every query vertex $u \in G$ in the instance of the vertex-triangles problem, we only ask queries on the characters representing neighbors of the vertex $u_1 \in V_1$ that corresponds to u in a specially tailored way. For such vertex $u_1 \in V_1$, we first list as the next query characters of the query text of

DMOG*/DROG* the characters corresponding to neighbors of u_1 from V_2, followed by α^* copies of the character representing the dummy vertex, and then list the characters corresponding to the neighbors of u_1 from V_3 as text query characters. Since the input of the DROG* and DMOG* problems is identical, the same reduction is used for both problems. Note that for the reduction to DROG* we may assume that each vertex is queried once for vertex-triangle queries as the answer for a single vertex does not change over the sequence of queries. Thus, there is a one-to-one correspondence (up to constant-time removal of duplicate output triangles) between the gapped patterns reported by the DROG*/DMOG* algorithms for the described sequence of characters in the query text and the output triangles that should be reported for a given vertex in the vertex-triangle problem.

From the construction of the tripartite graph and the input to the DROG* or DMOG* algorithm, two characters representing vertices of an edge in G_T that represents a gapped pattern the output of the DROG*/DMOG* algorithm must be separated in the query text list by at least α^* characters/vertices, and by at most the length of the list which is β^*. Thus, by the CLB for DMOG [5,6], the amortized expected time spent on a character/vertex must be at least $\Omega(\delta(G)^{1-o(1)})$. Since the proof of Corollary 9 ensures the existence of a graph for which $\hat{d} = \delta(G) = \Theta(m^x)$, we have that the amortized expected time spent on a vertex must be at least $\Omega(\delta(G)^{1-o(1)}) = \Omega((m^x)^{1-o(1)}) = \Omega((\beta^* - \alpha^*)^{1-o(1)})$. The theorem then follows from the reduction from vertex triangles to DROG*/DMOG* and the CLB for vertex triangles together with the fact that this CLB holds no matter what the output size is. □

5. Conclusion and open problems

In this paper we give the first formalization of the Dictionary Recognition with One Gap (DROG) problem and give solutions for this problem in the online setting required for NIDS applications. An important feature of our algorithms in that their complexity depends on parameters that are small in some practical inputs, especially in NIDS. Some open problems are:

- Can some of the factors in these solutions be reduced?
- Can these solutions be adapted to take care of a dictionary with subpatterns having more than one gap?

Since the DROG problem is a crucial bottleneck procedure in NIDS applications these open problems should be addressed in the future.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Abboud, V.V. Williams, Popular conjectures imply strong lower bounds for dynamic problems, in: Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS), 2014, pp. 434–443.
[2] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.
[3] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, A.A. Schäffer, Improved dynamic dictionary matching, Inf. Comput. 119 (2) (1995) 258–282.
[4] A. Amir, D. Keselman, G.M. Landau, M. Lewenstein, N. Lewenstein, M. Rodeh, Text indexing and dictionary matching with one error, J. Algorithms 37 (2) (2000) 309–325.
[5] A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, 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), 2016, pp. 12:1–12:12.
[6] A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, B.R. Shalom, Mind the gap! Online dictionary matching with one gap, Algorithmica 81 (6) (2019) 2123–2157.
[7] A. Amir, A. Levy, E. Porat, B.R. Shalom, Dictionary matching with one gap, in: Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM), 2014, pp. 11–20.
[8] A. Amir, A. Levy, E. Porat, B.R. Shalom, Dictionary matching with a few gaps, Theor. Comput. Sci. 589 (2015) 34–46.
[9] J.L. Bentley, Decomposable searching problems, Inf. Process. Lett. 8 (5) (1979) 244–251.
[10] P. Bille, I.L. Gørtz, H.W. Vildhøj, D.K. Wind, String matching with variable length gaps, Theor. Comput. Sci. 443 (2012) 25–34.
[11] P. Bille, M. Thorup, Regular expression matching with multi-strings and intervals, in: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 1297–1308.
[12] A. Bjørklund, R. Pagh, V.V. Williams, U. Zwick, Listing triangles, in: Proceedings 41st International Colloquium on Automata, Languages, and Programming (ICALP (I)), 2014, pp. 223–234.
[13] G.S. Brodal, L. Gasieniec, Approximate dictionary queries, in: Proceeding of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM), 1996, pp. 65–74.
[14] N. Chiba, T. Nishizeki, Arboricity and subgraph listing algorithms, SIAM J. Comput. 14 (1) (1985) 210–223.
[15] R. Cole, L. Gottlieb, M. Lewenstein, Dictionary matching and indexing with errors and don't cares, in: Proceedings of the 36th Annual ACM Symposium on Theory on Computing (STOC), 2004, pp. 91–100.
[16] K. Fredriksson, S. Grabowski, Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance, Inf. Retr. 11 (4) (2008) 335–357.
[17] A. Grønlund, S. Pettie, Threesomes, degenerates, and love triangles, J. ACM 65 (4) (2018).
[18] T. Haapasalo, P. Silvasti, S. Sippu, E. Soisalon-Soininen, Online dictionary matching with variable-length gaps, in: Symposium on Experimental Algorithms (SEA), 2011, pp. 76–87.

[19] K. Hofmann, P. Bucher, L. Falquet, A. Bairoch, The PROSITE database, its status in 1999, Nucleic Acids Res. 27 (1) (1999) 215–219.
[20] W. Hon, T. Lam, R. Shah, S.V. Thankachan, H. Ting, Y. Yang, Dictionary matching with uneven gaps, in: Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM), 2015, pp. 247–260.
[21] W. Hon, T. Lam, R. Shah, S.V. Thankachan, H. Ting, Y. Yang, Dictionary matching with a bounded gap in pattern or in text, Algorithmica 80 (2) (2018) 698–713.
[22] A. Itai, M. Rodeh, Finding a minimum circuit in a graph, SIAM J. Comput. 7 (4) (1978) 413–423.
[23] T. Kopelowitz, S. Pettie, E. Porat, Dynamic set intersection, in: Proceedings of the 14th Annual Symposium on Algorithms and Data Structures (WADS), 2015, pp. 470–481.
[24] T. Kopelowitz, S. Pettie, E. Porat, Higher lower bounds from the 3sum conjecture, in: Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2016, pp. 1272–1287.
[25] G. Kucherov, M. Rusinowitch, Matching a set of strings with variable length don't cares, Theor. Comput. Sci. 178 (1–2) (1997) 129–154.
[26] D.T. Lee, C.K. Wong, Quintary trees: a file structure for multidimensional database systems, ACM Trans. Database Syst. 5 (3) (1980) 339.
[27] N. Liu, F. Xie, X. Wu, Multi-pattern matching with variable-length wildcards using suffix tree, Pattern Anal. Appl. 21 (4) (2018) 1151–1165.
[28] G.S. Lueker, A data structure for orthogonal range queries, in: Proceedings of the 19th Annual Symposium on Foundations of Computer Science, 1978, pp. 28–34.
[29] K. Mehlhorn, S. Näher, Dynamic fractional cascading, Algorithmica 5 (2) (1990) 215–241.
[30] M. Morgante, A. Policriti, N. Vitacolonna, A. Zuccolo, Structured motifs search, J. Comput. Biol. 12 (8) (2005) 1065–1082.
[31] C.W. Mortensen, Fully dynamic orthogonal range reporting on RAM, SIAM J. Comput. 35 (6) (2006) 1494–1525.
[32] E.W. Myers, A four Russians algorithm for regular expression pattern matching, J. ACM 39 (2) (1992) 430–448.
[33] G. Myers, G. Mehldau, A system for pattern matching applications on biosequences, Comput. Appl. Biosci. 9 (3) (1993) 299–314.
[34] G. Navarro, M. Raffinot, Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching, J. Comput. Biol. 10 (6) (2003) 903–923.
[35] Y. Nekrich, Space efficient dynamic orthogonal range reporting, Algorithmica 49 (2007) 94–108.
[36] M. Pătraşcu, Towards polynomial lower bounds for dynamic problems, in: Proceedings of the 42nd Annual Symposium on Theory of Computing (STOC), 2010, pp. 603–610.
[37] Verint Systems. Personal communication, 2013. Addres: Maskit St. 33 Herzliya Israel.
[38] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\theta(n)$, Inf. Process. Lett. 17 (2) (1983) 81–84.
[39] M. Zhang, Y. Zhang, L. Hu, A faster algorithm for matching a set of patterns with variable length don't cares, Inf. Process. Lett. 110 (6) (2010) 216–220.