Algorithmica (2019) 81:2123–2157
https://doi.org/10.1007/s00453-018-0526-2

Mind the Gap!

Online Dictionary Matching with One Gap

Amihood Amir[1,2] · **Tsvi Kopelowitz**[1,3] · **Avivit Levy**[4] · **Seth Pettie**[3] · **Ely Porat**[1] · **B. Riva Shalom**[4]

Received: 15 November 2016 / Accepted: 8 November 2018 / Published online: 13 November 2018

Abstract
We examine the complexity of the online Dictionary Matching with One Gap Problem (DMOG) which is the following. Preprocess a dictionary D of d patterns, where each pattern contains a special *gap* symbol that can match any string, so that given a text that arrives online, a character at a time, we can report all of the patterns from D that are suffixes of the text that has arrived so far, before the next character arrives. In more general versions the gap symbols are associated with *bounds* determining the possible lengths of matching strings. Online DMOG captures the difficulty in a bottleneck procedure for cyber-security, as many digital signatures of viruses manifest themselves as patterns with a single gap. In this paper, we demonstrate that the difficulty in obtaining efficient solutions for the DMOG problem, even in the offline setting, can be traced back to the infamous 3SUM conjecture. We show a conditional lower bound of $\Omega(\delta(G_D) + op)$ time per text character, where G_D is a bipartite graph that captures the structure of D, $\delta(G_D)$ is the *degeneracy* of this graph, and op is the output size. Moreover, we show a conditional lower bound in terms of the magnitude of gaps for the bounded case, thereby showing that some known offline upper bounds are essentially optimal. We also provide upper-bounds in terms of the degeneracy for the online DMOG problem. In particular, we introduce algorithms whose time cost depends linearly on $\delta(G_D)$. Our algorithms make use of graph orientations, together with some additional techniques. These algorithms are of interest for practical cases in which $\delta(G_D)$ is a small constant. Since $\delta(G_D)$ can in general be as large as $\sqrt{d}$, and even larger if G_D is a multi-graph, we also obtain other solutions adequate for such dense cases.

Keywords Pattern matching · Dictionary matching · 3SUM · Triangle reporting

Partially supported by ISF Grant 571/14, BSF Grant 2014028, and NSF Grants CCF-1217338, CNS-1318294, and CCF-1514383. This work is supported in part by ISF Grant 1278/16. This project has received funding from the European Research Council (ERC) under the European Union Horizon 2020 research and innovation programme (Grant Agreement No. 683064). A partial version of this paper appeared in ISAAC 2016.

Extended author information available on the last page of the article

1 Introduction

Understanding the computational limitations of algorithmic problems often leads to algorithms that are efficient for inputs that are seen in practice. This paper, which stemmed from an industrial-academic connection [32], is an example of such a case. We focus on an aspect of Cyber-security which is a critical modern challenge. Network intrusion detection systems (NIDS) perform protocol analysis, content searching and content matching, 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 [32]. 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 (see Table 1). 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 [20,29] and has been revisited many times in the intervening years (e.g. [8,9,16,26,28,30,33]), in this paper we study what is apparently a mild generalization of the problem that has nonetheless resisted attempts at finding a definitive efficient solution.

The set of d patterns to be detected, called a *dictionary*, could be quite large. While dictionary matching is well studied (see, e.g. [2,4,5,11,14]), NIDS applications motivate the *dictionary matching with one gap* problem, defined formally as follows.

Definition 1 *The Dictionary Matching with One Gap Problem (DMOG)*, is:

Input: A text T of length $|T|$ over alphabet Σ, and a dictionary D of d gapped patterns $P_1, \ldots, P_d$ over alphabet Σ where each pattern has at most one gap.
Output: All locations in T where a pattern $P_i \in D$, $1 \le i \le d$, ends.

In the offline DMOG problem T and D are presented all at once. We study the more practical *online* DMOG 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 the subset of patterns with a match ending at this character should be reported before the next character arrives. Three cost measures are of interest: a preprocessing time, a time per character, and a time per match reported. Online DMOG is a serious bottleneck for NIDS, though it has received attention from both the industry and the academic community.

1.1 Previous Work

Finding efficient solutions for DMOG has proven to be a difficult algorithmic challenge as, unfortunately, little progress has been obtained on this problem even though researchers both in the pattern matching community and the industry have tackled it.

Table 1 The table presents SNORT rules statistics: number of gaps in each pattern, number of such patterns, total number of subpatterns in such patterns, maximum and average subpattern length, number of subpatterns that are not suffix or prefix of any other subpattern, number of unbounded gaps, gap bounds 1–10, gap bounds 11–100 and gaps of bound more than 100

Gaps no.	Pat. no.	Subp. no.	Max len.	Aver. len.	Non-suf. Subp.	Un-bounded	Bound 1–10	Bound 11–100	Bound > 100
0	7927	7927	347	20.39	1842	0	0	0	0
1	2478	4956	184	13.13	1251	1796	621	54	7
2	1767	5301	138	9.65	1224	2343	995	171	25
3	681	2724	88	9.11	577	1331	575	107	30
4	396	1980	138	9.11	292	1160	263	132	29
5	136	816	97	9	86	520	85	55	20
6	91	637	49	10.05	53	418	49	69	10
7	38	304	45	8.48	20	230	12	24	0
8	17	153	34	7.16	11	130	5	1	0
9	12	120	39	8.35	0	78	8	8	14
10	4	44	37	7.41	0	40	0	0	0
11	6	72	31	7.51	0	66	0	0	0
12	4	52	47	6.37	13	38	9	1	0
13	0	0	0	0	0	0	0	0	0
14	1	15	7	1.4	5	1	12	1	0

Most of the gaps are unbounded and bounded gaps usually have range less than 100

There are only 20 gaps (less than 0.004%) with range greater than 1000. The total number of patterns reflecting the SNORT rules is 13,556, where the total number of subpatterns is 25,101, of which 16,246 are unique (i.e., appear only once) and 2995 appear in more than one pattern

Table 2 describes a summary and comparison of previous work. It illustrates that previous formalizations of the problem, 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. Table 2 also shows a comparison of the upper and lower bounds and demonstrates the situations in which our solutions are essentially optimal (assuming some popular conjectures, as described in Sect. 2).

1.2 New Results

The DMOG problem has several natural parameters, e.g., $\mathfrak{D}$, which is the total size of the dictionary D, d, which is the number of patterns in the dictionary, and the magnitude of the gap. We establish upper and lower bounds for the cases of unbounded gaps ($\alpha = 0$, $\beta = \infty$), uniformly bounded gaps where all patterns have the same bounds, α and β, on their gap, and the most general non-uniform gaps version, where each pattern $P_i \in D$ has its own gap bounds, α_i and β_i. Note that, in NIDS applications the total size of the query text maybe huge as it arrives continuously in online manner, but other parameters have fixed values. A dictionary D for SNORT patterns with one gap has: $\mathfrak{D} = 65073$ and $d = 2478$ (see Table 1). We show that the complexity of DMOG depends also on a "hidden" parameter that is a function of the *structure* of the gapped patterns. This structure is exposed by using a graph representation of the dictionary D.

The Bipartite Graph G_D The dictionary D can be represented as a graph G_D, as follows. The set of vertices V correspond to first or second subpatterns and the set of edges E correspond to patterns, i.e., connects a vertex representing a first subpattern of a dictionary pattern to the vertex representing its corresponding second subpattern. The resulting graph $G = (V, E)$ is converted to a bipartite graph by creating two copies of V called L (the left vertices) and R (the right vertices). For every edge $(u, v) \in E$ there is an edge in the bipartite graph between $u_L \in L$ and $v_R \in R$, where u_L is a copy of u and v_R is a copy of v (see examples in Fig. 1). The occurrence of a subpattern, then corresponds to an arrival of a vertex, where each vertex in V that arrives during query time is replaced by its two copies, first the copy from R and then the copy from L. This ordering guarantees that a self loop in G is not mistakenly reported the first time its single vertex arrives. Note that G_D is a multi-graph in the non-uniformly bounded gaps case, since in this case there may be several edges between the same pair of vertices due to different gap bounds representing different possibilities of time limitations where occurrences of the vertices result in reporting dictionary patterns occurrences (see Fig. 1b).

We use the notion of graph *degeneracy* $\delta(G_D)$ which is defined as follow. The degeneracy of a graph $G = (V, E)$ is $\delta(G) = \max_{U \subseteq V} \min_{u \in U} d_{G_U}(u)$, where d_{G_U} is the degree of u in the subgraph of G induced by U. 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. Notice that the degeneracy of G is unchanged, up to constant factors, due to the transformation to the bipartite graph G_D.

Table 2 Comparison of previous work and some new results

References	Preprocess time	Total query time	Algorithm type	Remark
[25]	None	$\tilde{O}(\|T\| + \mathfrak{D})$	Online	Reports only first occurrence
[33]	$O(\mathfrak{D})$	$\tilde{O}(\|T\| + d)$	Online	Reports only first occurrence
[18]	$O(\mathfrak{D})$	$O(\|T\| \cdot lsc + socc)$	Online	Reports one occurrence per pattern and location
[6]	$\tilde{O}(\mathfrak{D})$	$\tilde{O}(\|T\|(\beta - \alpha) + op)$	Offline	DMOG
[21]	$O(\mathfrak{D})$	$\tilde{O}(\|T\|(\beta^* - \alpha^*) + op)$	Offline	DMOG
This paper	$O(\mathfrak{D})$	$\tilde{O}(\|T\| \cdot \delta(G_D) \cdot lsc + op)$	Online	DMOG
This	$O(\mathfrak{D})$	$\Omega(\|T\| \cdot \delta(G_D)^{1-o(1)} + op)$	Online	DMOG
paper	$O(\mathfrak{D})$	$\Omega(\|T\| \cdot (\beta - \alpha)^{1-o(1)} + op)$	Or offline	

The parameters: $\mathfrak{D}$ is the total size of the dictionary D, lsc is the longest suffix chain of subpatterns in D, $socc$ 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

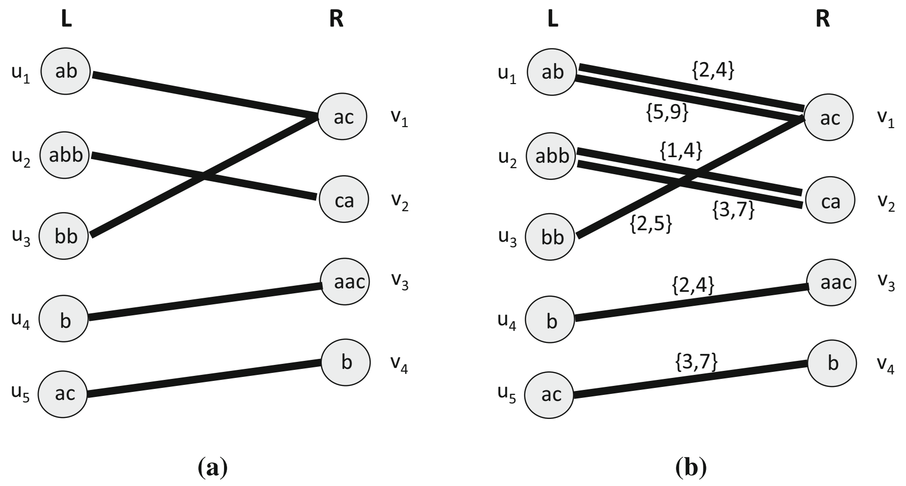


Fig. 1 **a** The bipartite graph G_D for the uniformly bounded gaps dictionary $D = \{ab\{2,4\}ac, abb\{2,4\}ca, bb\{2,4\}ac, b\{2,4\}aac, ac\{2,4\}b\}$. In this dictionary $\alpha = 2$, $\beta = 4$. **b** The bipartite multi-graph G_D for the non-uniformly bounded gaps dictionary $D = \{ab\{2,4\}ac, ab\{5,9\}ac, abb\{1,4\}ca, abb\{3,7\}ca, bb\{2,5\}ac, b\{2,4\}aac, ac\{3,7\}b\}$. In this dictionary $\alpha^* = 1$, $\beta^* = 9$

Vertex-Triangle Queries A key component in understanding both the upper and lower bounds for DMOG is the *vertex-triangles* problem, where the goal is to preprocess a graph so that given a query vertex u we may list all triangles that contain u. The vertex-triangles problem, besides being a natural graph problem, is of particular interest here since, as will be demonstrated in Sect. 2, it is reducible to DMOG. Our reduction demonstrates that the complexity of the DMOG problem already emerges when all patterns are of the form of two characters separated by an unbounded gap. This simplified online DMOG problem (denoted as DMOG*) is equivalent to the *Induced Subgraph* (ISG) problem, formally defined as follows.

Definition 2 *The Induced Subgraph (ISG)* problem is:

Input: A graph $G = (V, E)$ and a sequence of vertices from V,
Output: For each vertex v arriving at time t, report all edges $(u, v) \in E$ such that u arrived at time t', where $t' < t$.

We consider the online setting of ISG, where we preprocess the graph $G = (V, E)$ in advance, then the sequence of vertices from V arrive one by one, and all occurrences of edges due to the arrival of a vertex are reported before the next vertex arrives. Notice that answering consecutive queries is done independently. Thus, characters and gapped patterns in DMOG* correspond to vertices and edges in ISG, respectively. This equivalence is summarized in Observation 1.

Observation 1 *The ISG and DMOG* problems are equivalent.*

We show that vertex-triangles queries are reducible to ISG. Equation 1 summarizes the path for proving the lower bound for the DMOG problem, where VT denotes the vertex-triangle problem. The reductions (shown in Sect. 2: the first is shown in Theorem 4 and the second in Theorem 1) are denoted as $\preceq$ and the equivalence between ISG and DMOG* (observed here) is denoted as $\cong$.

$$\mathsf{3SUM} \preceq VT \preceq ISG \cong DMOG^* \tag{1}$$

Lower Bounds Leading to New Upper Bounds After trying to tackle the DMOG problem from the upper bound perspective, we suspected that a lower bound could be proven, and indeed were successful in showing a connection to the 3SUM conjecture. The *conditional lower bound* (CLB) proof provides insight for the inherent difficulty in solving DMOG, but is also unfortunate news for those attempting to find efficient upper bounds. Fortunately, a careful examination of the reduction from 3SUM to DMOG reveals that the CLB from the 3SUM conjecture can be phrased in terms of $\delta(G_D)$, which 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 [32]. This lead to designing algorithms whose runtime can be expressed in terms of $\delta(G_D)$, and can therefore be helpful in such practical settings.

Thus, our reduction serves two purposes. First, in Sect. 2 we prove a CLB for DMOG based on the 3SUM conjecture by combining a reduction from triangle enumeration to the vertex-triangles problem with our new reduction from the vertex-triangles problem to DMOG. Our lower bound states that any online DMOG algorithm with low preprocessing and reporting costs must spend $\Omega(\delta(G_D)^{1-o(1)})$ per character, assuming the 3SUM conjecture. Interestingly, the path for proving this CLB deviates from the common conceptual paradigms for proving lower bounds conditioned on the 3SUM conjecture, and is of independent interest. In particular, the common paradigm considers set-disjointness or set-intersection type problems, which correspond to edge triangle queries, while here we consider vertex-triangle queries. Moreover, our CLB holds for the *offline* case as well, and can be rephrased in terms of other parameters. For example, in the DMOG problem with uniform gaps $\{\alpha, \beta\}$, we prove that the cost per character of scanning T must be $\Omega((\beta - \alpha)^{1-o(1)})$. This gives some indication that some recent algorithms for the offline version of DMOG problem are essentially optimal ([6,21]).

Second, in Sect. 3 we provide optimal solutions (under the 3SUM conjecture), up to subpolynomial factors, for ISG and, therefore, also for vertex-triangles queries, with $O(|E|)$ preprocessing time and $O(\delta(G) + op)$ time per each vertex, where op is the size of the output due to the vertex arrival. The connection between ISG and DMOG led us to extend the techniques used to solve ISG, combine them with additional ideas and techniques, thereby introduce several new online DMOG algorithms in Sect. 4. The statement of our DMOG 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].

While the parameter lsc could theoretically be as large as d, in many practical situations it is very small. In an analysis of the graph created using SNORT software rules lsc is not greater than 5 [32]. Note that, in natural languages dictionaries such as the English dictionary lsc is also a small constant. While it is possible to find a suffix chain of English words with length 7, it is difficult (if possible) to find such a chain with greater length. Nevertheless, we also present 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 $\sqrt{lsc \cdot d}$. Notice that while in the uniformly bounded case we have $\delta(G_D) = O(\sqrt{d})$, in the non-uniform case $\delta(G_D)$ could be much higher and so these new algorithms become a vast improvement.

Table 3 summarizes our upper-bounds for DMOG. Note that, since subpatterns may be long, we must accommodate a delay in the time a vertex corresponding to a second subpattern is treated as if it has arrived, thus inducing a minor additive space usage. For simplicity of exposition, we assume that $|\Sigma|$ is constant. We stress that it is easy to formally reduce DMOG over any alphabet Σ to DMOG over alphabet $\{0, 1\}$ by encoding any symbol in binary in a way that makes the start and end of an encoding clear, e.g. the b-th element of Σ, where $b = b[0]b[1] \dots b[k]$ in binary, is mapped to $110b[0]0b[1] \dots 0b[k]10$. Such standard binary encoding technique can be adapted for the Aho-Corasic automaton [2] we use as a subpattern detection mechanism, so that the only change in our upper-bounds for an unbounded size alphabet is an additional cost of $O(\log |\Sigma|)$ worst-case time per character.

Paper Contributions The main contributions of this paper are:

- Obtaining algorithms for DMOG that are asymptotically fast for some practical inputs.
- Proving conditional lower bounds from the 3SUM conjecture, which in particular deviate from the common paradigm of such proofs.
- Formalizing the ISG problem. This problem serves in this paper for supplying a deeper understanding of the DMOG problem, but is also of independent interest.

Paper Organization Sect. 2 describes our conditional lower bounds. In Sect. 3 we introduce a solution for ISG, which is then extended to simplified versions of the unbounded, uniformly and non-uniformly bounded DMOG* problems in Sects. 3.1, 3.2 and 3.3 . In Sect. 4.1, the ISG algorithms are extended to solutions for the various DMOG versions. Finally, in Sect. 4.2 we obtain upper bounds adequate for denser instances of DMOG.

2 3SUM: Conditional Lower Bounds

In this section we prove that conditioned on the 3SUM conjecture we can prove lower bounds for the vertex-triangles problem, the ISG problem, and the (offline) unbounded DMOG problem. Note that the different versions of DMOG, namely, unbounded, uniformly and non-uniformly bounded gaps, form a hierarchy of generalisations, 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. Note, that setting β or β_i to ∞ actually means setting it to be of size $|T|$ in the offline version

Table 3 A summary of upper bounds for DMOG described in this paper

Gaps type	Preprocess time	Query time per text character	Space
Uniform (sparse graph)	$O(\mathfrak{D})$	$O(\delta(G_D) \cdot lsc + op)$	$O(\mathfrak{D} + lsc(\beta + M))$
Non-uniform (sparse graph)	$O(\mathfrak{D})$	$\tilde{O}(\delta(G_D) \cdot lsc + op)$	$\tilde{O}(\mathfrak{D} + lsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + lsc \cdot \alpha^*)$
Uniform (dense graph)	$O(\mathfrak{D})$	$O(lsc + \sqrt{lsc \cdot d} + op)$	$O(\mathfrak{D} + lsc(\beta + M))$
Non-uniform (dense graph)	$O(\mathfrak{D} + d(\beta^* - \alpha^*))$	$\tilde{O}(\sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + op)$	$\tilde{O}(\mathfrak{D} + d(\beta^* - \alpha^*) + \sqrt{lsc \cdot d}(\beta^* + M))$

Uniform and non-uniform refer to the type of gap bounds under consideration. M is the maximal length of a subpattern in the dictionary D

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, the more general DMOG versions can solve (possibly less efficiently) the unbounded DMOG version by setting the values of the gap bounds as above accordingly, therefore, the lower bounds hold for the generalised problems as well.

Background Polynomial (unconditional) lower bounds for data structure problems are considered beyond the reach of current techniques. Thus, it has recently become popular to prove CLBs based on the *conjectured* hardness of some problem. One of the most popular conjectures 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. This conjecture holds even if the algorithm is allowed to use randomization (see e.g. [1,17,24,31]). In this section we show that the infamous 3SUM problem can be reduced to DMOG, which sheds some light on the difficulty of the DMOG problem. Interestingly, our reduction does not follow the common paradigm for proving CLBs based on the 3SUM conjecture, providing a new approach for reductions from 3SUM. This approach is of independent interest, and is described next.

Triangles Pătraşcu [31] showed that 3SUM can be reduced to enumerating triangles in a tripartite graph. Kopelowitz, Pettie, and Porat [24] provided more efficient reductions, thereby showing that many known triangle enumeration algorithms ([10,12,22,23]) are essentially and conditionally optimal, up to subpolynomial factors. Hence, the offline version of triangle enumeration is well understood. The following two indexing versions of the triangle enumeration problem are a natural extension of the offline problem. In the *edge-triangles* problem the goal is to preprocess a graph so that given a query edge e all triangles containing e are listed. The vertex-triangles problem is defined above. Clearly, both these versions solve the triangle enumeration problem, which immediately gives lower bounds conditioned on the 3SUM conjecture.

The edge-triangles problem on a tripartite graph corresponds to preprocessing a family F of sets over a universe U in order to support set intersection queries in which given two sets $S, S' \in F$ the goal is to enumerate the elements in $S \cap S'$ (see [24]). Indeed, the task of preprocessing F to support set-intersection enumeration queries, and hence edge-triangles, is well studied [13,23]. Furthermore, the set intersection problem has been used extensively as a tool for proving that many algorithmic problems are as hard as solving 3SUM [1,24,31]. However, the vertex-triangles problem has yet to be considered directly.[1]

The Lower Bounds We use the vertex-triangles problem in order to show that the ISG problem is hard, and thus the simplest DMOG version of (offline) unbounded setting is 3SUM-hard. The most significant conditional lower bounds that we prove are stated by the following theorems. To understand the statements of the following theorems, when the total query time of an algorithm can be formulated as $O(t_q + op \cdot t_r)$ time, we say that t_q is the query time and t_r is the reporting time.

[1] The closely related problem of deciding whether a given vertex is contained by any triangle (a decision version) has been addressed [7].

Theorem 1 *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. For any constant $0 < x < 1/2$, any algorithm that solves the ISG problem on a graph G with m edges, $n = \Theta(m^{1-x})$ vertices, 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 $\Omega((\delta(G))^{1-o(1)})$.*

Theorem 2 *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. For any constant $0 < x < 1/2$, any algorithm that solves the DMOG problem on a dictionary D with d patterns and $\Theta(d^{1-x})$ subpatterns, if the amortized expected preprocessing time is $O(d \cdot \delta(G_D)^{1-\Omega(1)})$ and the amortized expected reporting time is sub-polynomial, then the amortized expected query time must be $\Omega((\delta(G_D))^{1-o(1)})$.*

Our proof begins from the conditional lower bounds for triangle enumeration introduced in [24]. The actual statement in [24] refers to the *arboricity* of G instead of the degeneracy of G. The arboricity of an undirected graph is the minimum number of forests into which its edges can be partitioned. We note that, both the terms degeneracy and arboricity are quantitatively the same, up to a factor of 2. If a graph G is oriented acyclically with outdegree k, then its edges may be partitioned into k forests by choosing one forest for each outgoing edge of each node. Thus, the arboricity of G is at most equal to its degeneracy. In the other direction, an n-vertex graph that can be partitioned into k forests has at most $k(n-1)$ edges and therefore has a vertex of degree at most $2k-1$. Thus, the degeneracy is less than twice the arboricity.

Theorem 3 [24] *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. Then for any constants $0 < x \leq 1$, $0 < y \leq 1$, such that $x \leq 2y$, there exists a graph G with n vertices, m edges, and arboricity $\gamma(G) = \Theta(n^x) = \Theta(m^y)$, with $t = O(m^{1-\Omega(1)})$ triangles, such that listing all triangles requires $\Omega(m \cdot \gamma(G)^{1-o(1)})$ expected time.*

Corollary 1 *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 $\hat{d} = \delta(G)$, where $\hat{d}$ is the average degree of a vertex in G, must spend $\Omega(m \cdot \delta(G)^{1-o(1)})$ expected time independent of the output size.*

Proof Theorem 3 assures that assuming 3SUM requires $\Omega(n^{2-o(1)})$ expected time, then for any constants $0 < x' \leq 1$, $0 < y' \leq 1$, such that $x' \leq 2y'$, there exists a family of graphs G with n vertices, m edges, and arboricity $\gamma(G) = \Theta(n^{x'}) = \Theta(m^{y'})$, with $t = O(m^{1-\Omega(1)})$ triangles, such that listing all triangles requires $\Omega(m \cdot \gamma(G)^{1-o(1)})$ expected time. At the end of their proof to Theorem 3, [24] show that the parameters of the graph G they constructed can be expressed in terms of a constant $0 < x < 1/2$, as follows: G has m edges, $n = \Theta(m^{1-x})$ vertices, and $\gamma(G) = \Theta(m^x)$. It is well-known that the arboricity of any graph is at least the average degree of its vertices. During their construction of G, [24] also show that the average degree of its vertices is also upper bounded by $\gamma(G)$. Therefore, $\hat{d} = \gamma(G) = \Theta(m^x)$. To conclude the proof we need only note that the degeneracy of G is asymptotically the same as the arboricity. Also note that the number of triangles in the constructed graph G is actually polynomially smaller than the $m \cdot \delta(G)$, but this only strengthen the stated lower bound, as it means that the lower bound in *not* due to the time required to report the output. □

We also need the following theorem.

Theorem 4 *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. For any constant $0 < x < 1/2$, any algorithm that solves the vertex-triangles problem on a graph G with m edges and $n = \Theta(m^{1-x})$ vertices, 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 vertices.*

Proof We reduce the triangle enumeration problem considered in Corollary 1 to the vertex-triangles problem. We preprocess G and then answer vertex-triangles queries on each of the m^{1-x} vertices thereby enumerating all of the triangles in G. If we assume a sub-polynomial reporting time, then by Corollary 1 either the preprocessing takes $\Omega(m \cdot \delta(G)^{1-o(1)})$ time or each query must cost at least $\Omega(\frac{m \cdot \delta(G)^{1-o(1)}}{m^{1-x}}) = \Omega((m^x \delta(G))^{1-o(1)}) = \Omega((\hat{d} \cdot \delta(G))^{1-o(1)})$ amortized expected time. □

We are now ready to prove Theorems 1 and 2.

Proof (Proof of Theorems 1 and 2) We reduce the vertex-triangles problem considered in Theorem 4 to ISG as follows. We preprocess the graph G for ISG queries. Now, to answer a vertex-triangle query on some vertex u, we input all of the neighbors of u into the ISG algorithm. Thus, there is a one-to-one correspondence between the edges reported by the ISG algorithm and the triangles in the output of the vertex-triangles query. 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. Since ISG is equivalent to a special case of DMOG in which every dictionary subpattern is a single character with unbounded gaps between them (denoted as DMOG*), and given Theorem 1, the proof of Theorem 2 follows. □

Another lower bound that we get is the following.

Theorem 5 *Assume 3SUM requires $\Omega(n^{2-o(1)})$ expected time. For any constant $0 < x < 1/2$, any algorithm that solves the uniformly bounded DMOG problem on a dictionary D with d patterns and $\Theta(d^{1-x})$ subpatterns, if the amortized expected preprocessing time is $O(d \cdot \delta(G_D)^{1-\Omega(1)})$ and the amortized expected reporting time is sub-polynomial, then the amortized expected time spent on each text character is $\Omega((\beta - \alpha)^{1-o(1)})$, where $\beta - \alpha = \Omega(d^x)$.*

Proof The proof is similar to the proofs of Theorems 1 and 2. First, 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 and for each edge (u, v) in G we add 6 edges to G_T between all possible copies of u and v. 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. We use ISG to solve vertex-triangles queries in Theorem 1, but we only ask queries on the neighbors of vertices in V_1 in a specially tailored way as follows. We first list the neighbors of u from V_2, followed by α copies of the dummy vertex, and then list the neighbors from V_3. From the construction of the tripartite graph and the input to the ISG algorithm, two vertices of an edge (u, v) that is part of the output of the ISG algorithm must be separated in the input list by at least α vertices, and by at most $\alpha + d(u) + d(v)$, where $d(u)$, $d(v)$ are the degree of u and v, respectively. Note, that the average over all possible vertices of the length of this list is β. Thus, by Theorem 1 the amortized expected time spent on a vertex is $\Omega(\delta(G)^{1-o(1)})$. Since the proof of Corollary 1 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 is $\Omega(\delta(G)^{1-o(1)}) = \Omega((m^x)^{1-o(1)}) = \Omega((\beta - \alpha)^{1-o(1)})$. Since ISG is equivalent to a special case of DMOG in which every dictionary subpattern is a single character with unbounded gaps between them (denoted as DMOG*), the theorem follows. □

A Note on Triangle Reporting Problems and Other Popular Conjectures Many CLBs based on other popular conjectures, such as the Boolean Matrix Multiplication conjecture or the Online Matrix Vector Multiplication conjecture, use reductions from set-disjointness and hence from edge-triangles queries (see [1,19]). However, it is not clear how to obtain meaningful lower bounds for vertex-triangles queries based on these conjectures. These difficulties are discussed below.

Since edge-triangle queries can be used to solve set disjointness, another natural candidate for a conjecture with which we can prove that edge-triangle queries are hard is the Boolean Matrix Multiplication (BMM) conjecture, which states that no $O(n^{3-\Omega(1)})$ combinatorial algorithm[2] exists for BMM on two $n \times n$ matrices. This is because the answer to a set disjointness query corresponds to the inner product of the characteristic vectors of the two sets, and each entry in the output of BMM is the inner product of one row and one vector from the input matrices. Notice that this approach derives a conditional lower bound from the BMM conjecture for the decision version of edge-triangle queries (does there exist a triangle containing the query edge), which can be solved via the reporting version considered above if we stop the query process after the first triangle is reported, thereby obtaining a conditional lower bound for the reporting version itself.

Just like the decision version of edge-triangle queries corresponds to the inner product of two boolean vectors, we can show that vertex-triangles queries correspond to the outer product of two vectors. However, outer products are too weak of a tool for proving conditional lower bounds from BMM, since the output of the outer product of two vectors of length n is n^2, and in order to solve BMM using outer products we need to consider n pairs of vectors and their outer products, resulting in $\Omega(n^3)$ information which is already too much.

Another candidate for proving the hardness of edge-triangle queries is the recent Online Matrix Vector (OMV) multiplication conjecture which states that there is no $O(n^{3-\Omega(1)})$ algorithm for multiplying an $n \times n$ matrix with n vectors of length n each, where the vectors arrive online and the output of the ith multiplication must be given prior to the arrival of the $(i + 1)$th vector. Since multiplying a matrix with a vector can be solved via inner products, the connection to edge-triangle queries is clear. However, it is not clear how to use the OMV conjecture to prove some hardness

[2] There is no clear definition of a *combinatorial algorithm*, and the notion that is accepted by the algorithmic community is that the way to establish if an algorithm is combinatorial or not is done by just looking at it.

on outer products, and so it is not clear if this conjecture can be used to prove the hardness of vertex-triangles queries, ISG, and DMOG.

3 The Induced Subgraph Problem

An Upper Bound via Graph Orientations In graph orientations the goal is to *orient* the graph edges while providing some guarantee on the out-degrees of the vertices. Formally, an orientation of an undirected graph $G = (V, E)$ is called a *c-orientation* if every vertex has out-degree at most $c \geq 1$. The notion of graph *degeneracy* is closely related to graph orientations [3]. Chiba and Nishizeki [12] linear time greedy algorithm assigns a $\delta(G)$-orientation of G. We preprocess G_D using this algorithm, thereby obtaining a c-orientation with $c = \delta(G_D)$, and use it for solving ISG problem as follows. First, we view an orientation 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, we say that u is *responsible* for e, and that e is *assigned* to u. Furthermore, u is a *responsible-neighbor* of v and v is an *assigned-neighbor* of u.

3.1 Unbounded Edge Occurrences

Each vertex $v \in R$ maintains a *reporting list* $\mathcal{L}_v$, which is a linked list containing links to responsible-neighbors of v that have already appeared during the current query. When a vertex $v \in R$ arrives at query time t, the elements in the reporting list $\mathcal{L}_v$ are scanned and their edges are reported. In addition, the edges for which v is their responsible-neighbour are scanned, and those for which the assigned-neighbour u is marked as arrived are reported. If the arrived vertex is $u \in L$, u is marked as arrived and is added to the reporting lists of its assigned-neighbours, deleting a previous appearance of u in those lists, if existed. If we want to report all the times an edge appeared we also need to maintain and update the time stamps of u in the list τ_u as in Sect. 3.2. However, as the length of these lists is unbounded in this case, it incurs additional linear space. We, therefore, only report a single occurrence of each edge per query vertex and do not maintain the τ_u lists at all. We denote this relaxation of the ISG problem by ISG*. Since the number of assigned-neighbors is bounded by $O(\delta(G))$, we have proven Theorem 6.

Theorem 6 *The ISG* problem on a graph G with m edges and n vertices can be solved online with* $O(m+n)$ *preprocessing time,* $O(\delta(G)+op)$ *time per query vertex, where op is the number of edges reported at vertex arrival, and* $O(m)$ *space.*

3.2 Uniformly Bounded Edge Occurrences

In this case, the ISG problem is restricted with two positive integer parameters α and β so an edge (u, v) can only be reported if $\alpha < t' - t \leq \beta + 1$, where t and t' are arrival times of u and v, respectively. The interval between β time units ago and α time units ago is called the *active window*.

The data structures used in this case are:

1. For each vertex $v \in R$, **a reporting list** $\mathcal{L}_v$ maintaining all responsible-neighbours of v, $u \in L$, that arrived at least α and at most β time units ago, without repetitions.
2. For each vertex $u \in L$, **an ordered list** τ_u of the time stamps u arrived within the the current active window.
3. The **list** $\mathcal{L}_\beta$ of the last β vertices $u \in L$. They are delayed for α time units before they are considered.

When a vertex arrives at time t, the data structures of the vertices are updated accordingly, as follows.

1. If the arrived vertex is $v \in R$,
   (a) The elements of $\mathcal{L}_v$ are scanned and their edges (u, v) are reported according to τ_u.
   (b) The edges for which v is their responsible-neighbour are scanned, and for every assigned-neighbour u that has a non empty τ_u, edge (u, v) is reported.
2. If the arrived vertex is $u \in L$, u is inserted into $\mathcal{L}_\beta$.

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

1. For every $v \in R$ that is an assigned neighbour of u, u is added to the beginning $\mathcal{L}_v$.
2. If τ_u is not empty, the previous appearance of u is removed from every $\mathcal{L}_v$, where v is an assigned neighbour of u.
3. $t - \alpha - 1$ is added to τ_u.

For vertices $u \in L$, arriving exactly $\beta + 1$ time units before time t,

1. u is removed from $\mathcal{L}_\beta$.
2. The time stamp $t - \beta - 1$ is removed from τ_u.
3. In case τ_u becomes empty, u is removed from every $\mathcal{L}_v$ where v is an assigned neighbour of u.

Theorem 7 *The Induced Subgraph problem with uniformly bounded edge occurrences on a graph G with m edges and n vertices can be solved with $O(m+n)$ preprocessing time, $O(\delta(G) + op)$ time per query vertex, where op is the number of edges reported at vertex arrival, and $O(m + \beta)$ space.*

Proof Updating $\mathcal{L}_\beta$, as well as treating a vertex arrived either to R or to L, requires time linear in the number of assigned-neighbours of a vertex. Thus the time cost per vertex is $O(\delta(G) + op)$. Regarding space, since each responsible neighbor appears only once, the space consumption of all the $\mathcal{L}_v$ lists is $O(m)$. The additional space usage is another $O(\beta)$ words for $\mathcal{L}_\beta$ and for the τ_u lists. □

3.3 Non-uniformly Bounded Edge Occurrences

In non-uniformly bounded edge occurrences each edge $e = (u, v)$ has its own boundaries $[\alpha_e, \beta_e]$ and can only be reported if $\alpha_e < t' - t < \beta_e + 1$, where t and t' are arrival times of u and v, respectively. Notice that in this case the input is a multi-graph as (u, v) with boundaries $[2, 4]$ is a distinct edge from (u, v) with boundaries $[5, 9]$, as can be seen in Fig. 1, therefore requires a distinct report if both appear in the input. The active window for this ISG version is the time window between $\beta^* = \max_{e \in E}\{\beta_e\}$ and $\alpha^* = \min_{e \in E}\{\alpha_e\}$ time units ago.

The case of non-uniformly bounded edge occurrences combined with the approach of a general active window, used in the previous subsection, introduces a new challenge. Suppose, we stored all responsible-neighbours of v, $u \in L$, that arrived during the active window, in $\mathcal{L}_v$ as in the uniformly bounded edge occurrences case. If τ_u includes all the appearances of u within the active window, when a vertex $v \in R$ arrives, the information in $\mathcal{L}_v$ cannot be automatically reported, as some of the appearances of nodes $u \in \mathcal{L}_v$ are not within the gaps of edge (u, v), thus only part of their τ_u list needs to be reported. A naive filtering considers for each $u \in \mathcal{L}_v$ a scan of τ_u and reports only time stamps t where $\alpha_e < t - t' \leq \beta_e + 1$, where t and t' are arrival times of u and v, which sums up to $\beta^* - \alpha^*$ time per query vertex. To avoid an overhead in query time, our filtering mechanism checks all appearances of all responsible-neighbours of v in a batched query, where each responsible-neighbour appearance is filtered according to the edge's gaps. This is achieved by maintaining for each vertex $v \in R$ a fully dynamic data structure S_v for supporting 4-sided 2-dimensional orthogonal range reporting queries instead of $\mathcal{L}_v$.[3] Given an $[x_0, y_0] \times [x_1, y_1]$-range, it returns the points of S_v that have (x, y) coordinates in the given range. For each responsible-neighbor $u \in L$ of v, that arrived in the active window in time t, where $e = (u, v)$, the point $(t + \alpha_e + 1, t + \beta_e + 1)$ is inserted into S_v, yielding the occurrences in S_v are from the "point of view" of v.

The data structures used in this case are:

1. For each vertex $v \in R$, **a data structure** S_v maintaining points representing all occurrences of responsible-neighbours of v, $u \in L$, that arrived at least α^* and at most β^* time units ago. To implement S_v, we use Mortensen's data structure [27] that supports the set of $|S_v|$ points from $\mathbb{R}^2$ with $O(|S_v| \log^{7/8+\epsilon} |S_v|)$ words of space, insertion and deletion time of $O(\log^{7/8+\epsilon} |S_v|)$ and $O(\frac{\log |S_v|}{\log\log |S_v|} + op)$ time for range reporting queries on S_v, where op is the size of the output.
2. For each vertex $u \in L$, **an ordered list** τ_u of time stamps of the times u arrived within the the current active window.
3. The **list** $\mathcal{L}_{\beta^*}$ of the last β^* vertices $u \in L$. They are delayed for at least α^* time units before they are considered.

When a vertex arrives at query time t, the data structures of the vertices are updated accordingly, as follows.

1. If the arrived vertex is $v \in R$,

[3] Since our final running time has a log-factor, the sub-logarithmic operations costs don't transfer to the final asymptotic bound. Thus, we can also use interval trees [15].

(a) A range query of $[0, t] \times [t, \infty]$ is performed over S_v. The edges representing the range output are reported.
(b) The edges for which v is their responsible-neighbour are scanned, and every assigned-neighbour u that has a non empty τ_u, where τ_u contains a time stamp t' such that $\alpha_e < t - t' \leq \beta_e + 1$, the edge (u, v) is reported.

2. If the arrived vertex is $u \in L$,

   (a) u is inserted into $\mathcal{L}_{\beta^*}$.
   (b) For each assigned-neighbour v, such that $e = (u, v)$, the point $(t + \alpha_e + 1, t + \beta_e + 1)$ is inserted to S_v.

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

1. $t - \alpha^* - 1$ is added to τ_u.

For vertices $u \in L$, arriving exactly $\beta^* + 1$ time units before time t,

1. u is removed from $\mathcal{L}_{\beta^*}$.
2. The time stamp $t - \beta_* - 1$ is removed from τ_u.
3. For each assigned-neighbour v, such that $e = (u, v)$, the point $(t - \beta^* + \alpha_e, t - \beta^* + \beta_e)$ is removed from S_v.

Theorem 8 *The Induced Subgraph problem with non-uniformly bounded edge occurrences on a graph G with m edges and n vertices can be solved with $O(m + n)$ preprocessing time, $\tilde{O}(\delta(G)\log(\beta^* - \alpha^*) + op)$ time per query vertex, where op is the number of edges reported due to the vertex arriving, and $\tilde{O}(m + \delta(G)(\beta^* - \alpha^*) + \alpha^*)$ space.*

Proof According to the algorithm, there are different procedures for vertices $v \in R$ and $u \in L$ arriving at time t, t' respectively.

For vertices $v \in R$, S_v is queried, yielding all of the points whose first coordinate is at most t and whose second coordinate is at least t. Each such reported point (x, y) is due to some vertex u with edge $e = (u, v)$, arriving at time t', for which $x = t' + \alpha_e < t \leq t' + \beta_e + 1 = y$. This guarantees that the range reported query provides a desired output. In addition, the assigned-neighbours of v are filtered according to their arrival time in at most $O(\log(\beta^* - \alpha^*) + op)$ where op is the size of the output. The cost of such a range query, in addition to the size of the current output, is $O(\frac{\log(\beta^* - \alpha^*)}{\log\log(\beta^* - \alpha^*)})$ time. In addition, the $O(\delta(G))$ assigned-neighbours of v are filtered according to their arrival time in at most $O(\log(\beta^* - \alpha^*) + op)$ where op is the size of the output.

Regarding vertices $u \in L$, their arrival time can require adding or deleting of at most $\delta(G)$ points from all of the orthogonal reporting data-structures at the assigned-neighbors of u at the cost of $O(\log^{7/8+\epsilon}(\beta^* - \alpha^*)$ for each update of a relevant S_v structure.

Regarding space: The space usage for the algorithm is $O(\beta^*)$ due to the vertices maintained in $\mathcal{L}_{\beta^*}$ and the lists of time stamps. Another $O(\delta(G)(\beta^* -$

$\alpha^*) \log^{7/8+\epsilon}(\delta(G)(\beta^* - \alpha^*)))$ space is required for all of the orthogonal range reporting data structures. □

4 Solving DMOG

4.1 DMOG via Graph Orientations

When extending ISG to online DMOG, the longer subpatterns introduce new challenges that need to be addressed. It is helpful to still consider the bipartite graph presentation of the DMOG instance, where vertices correspond to subpatterns and edges correspond to patterns. The algorithms from Sect. 3 are used as basic building blocks in our algorithms for DMOG by treating a subpattern arriving as the vertex arriving in the appropriate graph, while addressing the difficulties that arise from subpatterns being arbitrarily long strings.

Subpatterns Detection Mechanism First, a mechanism for determining when a subpattern arrives is needed. One way of doing this is by using the the Aho–Corasick (AC) Automaton [2], using a standard binary encoding technique so that each character costs $O(\log|\Sigma|)$ worst-case time. For simplicity we assume that $|\Sigma|$ is constant. However, while in the ISG problem each character corresponds to the arrival of at most one subpattern, in the DMOG each arriving character may correspond to several subpatterns which all arrive at once, since a subpattern could be a proper suffix of another subpattern. We, therefore, phrase the complexities of our algorithms in terms of lsc, which is the maximum number of vertices in the bipartite graph that arrive due to a character arrival. This induces a multiplicative overhead of at most lsc in the query time per text character relative to the time used by the ISG algorithms.

Finally, there is an issue arising from subpatterns no longer being of length one, which for simplicity we first discuss this in the unbounded case. When $u \in L$ arrives and it has an assigned vertex $v \in R$ where m_v is the length of the subpattern associated with v, then we do not want to report the edge (u, v) until at least $m_v - 1$ time units have passed from the arrival of u, since the appearance of the subpattern of v should not overlap with the appearance of the subpattern of u. Similarly, in the bounded case, we must delay the removal of u from $\mathcal{L}_v$ by at least $m_v - 1$ time units. Notice that if we remove u 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. We, therefore, delay the removal of u by $M - 1$ time units, where M is the length of the longest subpattern that corresponds to a vertex in R. This solves the issue of synchronization, however, some of the reporting lists now have elements that should not be reported. Nevertheless, the reporting time remains asymptotical to the size of the output as the elements in $\mathcal{L}_v$ are ordered by decreasing occurrence time, thus the first element u, found in $\mathcal{L}_v$ with occurrence time larger than β, implies the termination of the report.

Combining these ideas with the algorithms in Sect. 3 gives the following algorithms.

Uniformly Bounded Gaps The data structures used in this case are:

1. For each vertex $v \in R$, **an ordered reporting list** $\mathcal{L}_v$ maintaining all responsible-neighbours of v, $u \in L$, that arrived at least α and at most $\beta + M$ time units ago, without repetitions.
2. For each vertex $u \in L$, **an ordered list** τ_u of the time stamps u arrived within the the current active window.
3. The **list** $\mathcal{L}_\beta$ of the last $\beta + M$ vertices $u \in L$. They are delayed for α time units before they are considered.

When the AC-automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a subpattern that is a suffix of the subpattern represented by state s,

1. If the arrived vertex is $v \in R$,
   (a) Let $u = \mathcal{L}_v.first$
   (b) while $\tau_u.first \geq t - m_v - \beta - 1$
      i. Report edges (u, v).
      ii. Proceed to the next u element in $\mathcal{L}_v$.
   (c) The edges for which v is their responsible-neighbour are scanned, and every assigned-neighbour u that has a non empty τ_u, where the first element in τ_u is the time stamp t' such that $t - m_v - \beta - 1 \leq t'$, the edge (u, v) is reported.
2. If the arrived vertex is $u \in L$, u is inserted into $\mathcal{L}_\beta$.

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

1. For every $v \in R$ that is an assigned neighbour of u, u is added to the beginning $\mathcal{L}_v$.
2. If τ_u is not empty, the previous appearance of u is removed from every $\mathcal{L}_v$, where v is an assigned neighbour of u.
3. $t - \alpha - 1$ is added to the beginning of τ_u.

For vertices $u \in L$, arriving exactly $\beta + M + 1$ time units before time t,

1. u is removed from $\mathcal{L}_\beta$.
2. The time stamp $t - \beta - M - 1$ is removed from the end of τ_u.
3. In case τ_u becomes empty, u is removed from every $\mathcal{L}_v$, where v is an assigned neighbour of u.

Theorem 9 *The DMOG problem with uniformly bounded gap borders can be solved such that dictionary patterns are reported online 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.*

Proof The algorithm consider a linear time traversal over the text using the AC-automaton. At each location t, we consider $O(lsc)$ vertices, representing the subpattern recognized by the AC-automaton at the current query time, and all its possible $O(lsc)$

suffixes that are subpatterns in the dictionary. Every vertex requires $O(\delta(G_D)$ operations besides the output report as a vertex $v \in R$ considers its assigned-neighbors and a vertex $u \in L$ is inserted int all the reporting lists of its assigned neighbors. The scanning of elements in $\mathcal{L}_v$ is considered as the reported occurrences, until locating the first element of a responsible vertex u, where the gap between the newest occurrence time of u and $t - m_v + 1$ is larger than β, where the scan is terminated, as the rest of the elements of $\mathcal{L}_v$ are older. The time of deleting a vertex from $\mathcal{L}_\beta$ or from $\mathcal{L}_v$ can be accounted for by their insertion time. In the preprocessing, the AC automaton is built in time linear in the size of the dictionary $\mathfrak{D}$.

Regarding space: Each responsible neighbor appears only once in $\mathcal{L}_v$, thus the space consumption of all the $\mathcal{L}_v$ lists is $O(d)$, where d is the number of gapped patterns in the dictionary. The AC automaton requires linear space in the size of the dictionary $\mathfrak{D}$. The additional space usage is required for the $O(lsc)$ vertices maintained for $O(\beta - \alpha + M)$ time units by the τ_u lists, and additional $O(lsc \cdot \alpha)$ is required for the $u \in L$ vertices maintained by $\mathcal{L}_\beta$ for α time units, until they can be considered a arrived. □

Non-Uniformly Bounded Gaps The data structures used in this case are:

1. For each vertex $v \in R$, **a data structure** S_v maintaining points representing all occurrences of responsible-neighbours of v, $u \in L$, that arrived at least α^* and at most β^* time units ago. To implement S_v, we use Mortensen's data structure [27].
2. For each vertex $u \in L$, **an ordered list** τ_u of time stamps of the times u arrived within the the current active window.
3. The **list** $\mathcal{L}_{\beta^*}$ of the last $\beta^* + M$ vertices $u \in L$. They are delayed for at least α^* time units before they are considered.

When the AC-automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a subpattern that is a suffix of the subpattern represented by state s,

1. If the arrived vertex is $v \in R$,
   (a) A range query of $[0, t - m_v + 1] \times [t - m_v + 1, \infty]$ is performed over S_v. The edges representing the range output are reported.
   (b) The edges for which v is their responsible-neighbour are scanned, and every assigned-neighbour u that has a non empty τ_u, where τ_u contains a time stamp t' such that $\alpha_e < t - m_v - 1 - t' \leq \beta_e + 1$, the edge (u, v) is reported.
2. If the arrived vertex is $u \in L$, u is inserted into $\mathcal{L}_{\beta^*}$.

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

1. $t - \alpha^* - 1$ is added to the beginning of τ_u.
2. For each assigned-neighbour v, such that $e = (u, v)$, the point $(t - \alpha^* + \alpha_e, t - \alpha^* + \beta_e)$ is inserted to S_v.

For vertices $u \in L$, arriving exactly $\beta^* + M + 1$ time units before time t,

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

Theorem 10 *The DMOG problem with non-uniformly bounded gap borders can be solved such that dictionary patterns are reported online in: $O(\mathfrak{D})$ preprocessing time, $\tilde{O}(lsc \cdot \delta(G_D) + op)$ time per text character, where op is the number of patterns that are reported due to the character arriving, and $\tilde{O}(\mathfrak{D} + lsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + lsc \cdot \alpha^*)$ space.*

Proof The algorithm considers a linear time traversal over the text using the AC-automaton. At each location t, we consider $O(lsc)$ vertices, representing the subpattern recognized by the AC-automaton at the current query time, and all its possible $O(lsc)$ suffixes that are subpatterns in the dictionary. Each vertex $v \in R$ requires a range query applied to S_v that contains at most $lsc(\beta^* - \alpha^* + M)$ points, thus requires $O(\frac{\log(lsc(\beta^*-\alpha^*+M))}{\log\log(lsc(\beta^*-\alpha^*+M))})$ per range query. Additional $O(\delta(G)\log(\beta^* - \alpha^* + M))$ time is required for scanning all of the assigned-neighbors of v and reporting those arrived within the gaps boundaries. Each vertex $u \in L$ requires updating $\mathcal{L}_{\beta^*}$, and inserting or deleting a node from S_v, of every assigned neighbor of u, v thus requires $O(\delta(G_D)\log^{7/8+\epsilon}(lsc(\beta^* - \alpha^* + M)))$. Thus, the time required per query time is $O(lsc \cdot \delta(G_D)\log(lsc(\beta^* - \alpha^* + M)) + op)$ when considering the maximal possible number of subpatterns recognized by the AC automaton at each query time. In the preprocessing, the AC automaton is built in time linear in the size of the dictionary $\mathfrak{D}$.

Regarding space: Since each S_v contains points only from its responsible neighbor, each lsc vertices that were located at each of the last $\beta^* - \alpha^* + M$ locations in the text could have been inserted to $\delta(G_D)$ structures S_v, yielding the space consumption of all the S_v lists is $lsc \cdot \delta(G_D)(\beta^* - \alpha^* + M)$. The AC automaton requires linear in the size of the dictionary $\mathfrak{D}$. The additional space usage is required for the $O(lsc)$ vertices maintained for $O(\alpha^*)$ time units by $\mathcal{L}_{\beta^*}$ until they can be considered as arrived. □

4.2 DMOG via Threshold Orientations

Sections 3 and 4.1 focus on orientations whose out-degree is bounded by $\delta(G_D)$. Thus, when $\delta(G_D) = \sqrt{d}$ the DMOG algorithms take $O(lsc \cdot \sqrt{d})$ time. This is exacerbated in the non-uniform case where the degeneracy can be much larger, since the same subpatterns can represent different gapped patterns if they have different gaps boundaries, thus two vertices can be connected by more than one edge. In this section we show how in such dense inputs we can reduce the factor depending on lsc from $lsc \cdot \delta(G_D)$ to $\sqrt{lsc \cdot d}$, by using a different method for orienting the graph edges which we refer to as a *threshold* orientation.

Definition 3 A vertex in G_D is *heavy* if it has more than $\sqrt{d/lsc}$ neighbors, and *light* otherwise.

Our algorithms use two key properties. The first is that light vertices have at most $\sqrt{d/lsc}$ neighbors, and the second is that the number of heavy vertices is less than $\sqrt{lsc \cdot d}$.

We orient all edges that touch a light vertex to leave that vertex, breaking ties arbitrarily if both vertices are light. Thus, every edge e connecting a light vertex with a light/heavy vertex, the light vertex is the responsible-neighbor, and the heavy vertex, if exists in e, is the assigned-neighbor. We handle differently edges with at most one heavy vertex as an endpoint and edges connecting two heavy vertices. The algorithms for the former edges appears in Sect. 4.2.1 while the algorithms for the latter edges are described in Sect. 4.2.2. The combination of all algorithms is the total solution for DMOG via threshold orientation.

4.2.1 Edges Connecting at Most One Heavy Vertex

Uniformly Bounded Gaps The data structures used by the algorithm dealing with edges where at most one of its endpoints is heavy, when considering uniformly bounded gaps, are the same as detailed in Sect. 4.1 for the case of uniformly bounded gaps. (ordered reporting lists $\mathcal{L}_v$ for each vertex $v \in R$, ordered lists τ_u of the time stamps for each $u \in L$ and the list $\mathcal{L}_\beta$ of the last $\beta + M$ vertices $u \in L$).

When the AC-automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a subpattern that is a suffix of the subpattern represented by state s,

1. If the arrived vertex is $v \in R$,
   (a) Let $u = \mathcal{L}_v.first$
   (b) while $\tau_u.first \geq t - m_v - \beta - 1$
      i. Report edges (u, v).
      ii. Proceed to the next u element in $\mathcal{L}_v$.
   (c) If v is a light vertex, the edges for which v is their responsible-neighbour are scanned, and every assigned-neighbour u that has a non empty τ_u, where the first element in τ_u is the time stamp t' such that $t - m_v - \beta - 1 \leq t'$, the edge (u, v) is reported.
2. If the arrived vertex is $u \in L$, u is inserted into $\mathcal{L}_\beta$.

In addition, the *active window* is maintained by updating $\mathcal{L}_\beta$ and acknowledging arrived nodes $u \in L$ that become relevant.
For vertices $u \in L$, arriving exactly $\alpha + 1$ time units before time t,

1. If u is a light vertex,
   (a) For every $v \in R$ that is an assigned neighbour of u, u is added to the beginning $\mathcal{L}_v$.
   (b) If τ_u is not empty, the previous appearance of u is removed from every $\mathcal{L}_v$, where v is an assigned neighbour of u.
2. $t - \alpha - 1$ is added to the beginning of τ_u.

For vertices $u \in L$, arriving exactly $\beta + M + 1$ time units before time t,

1. u is removed from $\mathcal{L}_\beta$.

2. The time stamp $t - \beta - M - 1$ is removed from the end of τ_u.
3. In case τ_u is empty and u is a light vertex, u is removed from every $\mathcal{L}_v$, where v is an assigned neighbour of u.

Lemma 1 *The DMOG problem with uniform gap borders can be solved for edges with at most a single heavy endpoint, 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.*

Proof An arriving vertex can be either light or heavy and either from L or from R. In case the vertex is heavy, it merely updates its time stamps and is inserted to $\mathcal{L}_\beta$, if it is $u \in L$, or we report relevant information from its $\mathcal{L}_v$ list, in time proportional to the output size, if it is $v \in R$. There is no necessity to go over the assigned-neighbour of v as the threshold orientation set every heavy vertex to be assigned to a light vertex.

Additional operations required in case the vertex is light, are insertion of the vertex to the data structures of all its assigned neighbors, if the vertex is $u \in L$ and reporting the assigned neighbors of v that arrived within the gap restrictions if the vertex is $v \in R$.

Since there are at most lsc vertices arriving at a time, and each light vertex has at most $O(\sqrt{d/lsc})$ assigned neighbors, solving the DMOG for edges with at most one heavy endpoint costs at most $\tilde{O}(lsc + \sqrt{lsc \cdot d})$ time. The preprocessing and space complexity are the same as those of the Sect. 4.1. □

Non-Uniformly Bounded Gaps The data structures used by the algorithm dealing with edges where at most one of their endpoints is heavy, when considering non-uniformly bounded gaps, are the same as detailed in Sect. 4.1 for the case of non-uniformly bounded gaps. (Range query data structures S_v for each vertex $v \in R$, ordered lists τ_u of the time stamps for each $u \in L$ and the list $\mathcal{L}_{\beta^*}$ of the last $\beta + M$ vertices $u \in L$).

When the AC-automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a subpattern that is a suffix of the subpattern represented by state s,

1. If the arrived vertex is $v \in R$,
   (a) A range query of $[0, t - m_v + 1] \times [t - m_v + 1, \infty]$ is performed over S_v. The edges representing the range output are reported.
   (b) If v is a light vertex, the edges for which v is their responsible-neighbour are scanned, and every assigned-neighbour u that has a non empty τ_u, where τ_u contains a time stamp t' such that $\alpha_e < t - m_v - 1 - t' \leq \beta_e + 1$, the edge (u, v) is reported.
2. If the arrived vertex is $u \in L$, u is inserted into $\mathcal{L}_{\beta^*}$.

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

1. $t - \alpha^* - 1$ is added to the beginning of τ_u.

2. If u is a light vertex, for each assigned-neighbour v, such that $e = (u, v)$, the point $(t - \alpha^* + \alpha_e, t - \alpha^* + \beta_e)$ is inserted to S_v.

For vertices $u \in L$, arriving exactly $\beta^* + M + 1$ time units before time t,

1. u is removed from $\mathcal{L}_{\beta^*}$.
2. The time stamp $t - \beta^* - M - 1$ is removed from the end of τ_u.
3. If u is a light vertex, for each assigned-neighbour v, such that $e = (u, v)$, the point $(t - \beta^* - M + \alpha_e, t - \beta^* - M + \beta_e)$ is removed from S_v.

Lemma 2 *The DMOG problem with non-uniform gap borders can be solved for edges with at most a single heavy endpoint, with $O(\mathfrak{D} + d(\beta^* - \alpha^*))$ preprocessing time, $\tilde{O}(lsc + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + op)$ time per query text character, and $\tilde{O}(\mathfrak{D} + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + lsc \cdot \alpha^*)$ space.*

Proof The proof is similar to that of Lemma 1 and Theorem 10. □

4.2.2 Edges Connecting two Heavy Vertices

The remaining task is reporting edges connecting two heavy vertices. From a very high level, we will leverage the fact that the number of heavy vertices is less than $\sqrt{lsc \cdot d}$, and so even if 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 vertices in R is still less than $\sqrt{lsc \cdot d}$. So using a batched scan on all of R will keep the time cost low. We show that after some preprocessing such a scan can produce the desired result. In addition, at each time unit, we handle only a single vertex $u \in L$ currently arriving, the one representing the longest subpattern found by the Aho Corasick automaton at that time, which is the subpattern associated with the current accepting state. Other subpatterns, which are suffixes of that subpattern are handled implicitly, without increasing the time complexity unless they are reported. The preprocessing that enables us this procedure uses a tree-like structure.

The Tree Structure We construct a tree T among the subpatterns associated with heavy vertices from L, where a vertex u' is an ancestor of a vertex u if and only if the subpattern associated with u' is a suffix of the subpattern associated with u. An additional vertex corresponding to the empty string is added as the root of T, since it is a suffix of every subpattern. The tree T can be constructed in linear time from the AC automaton of D. An example of such a tree appears in Fig. 2. The graph vertices arriving due to a text character arrival correspond to the vertices on some path from the root of T to some vertex u, not including the root. We emphasize that the AC automaton mechanics allow one to report only one vertex u such that u and all of its ancestors exactly correspond to the subpatterns that have arrived (implicitly).

In the following paragraphs we provide algorithms for solving the DMOG problem for edges connecting two heavy nodes for the cases of uniformly and non-uniformly bounded gaps.

Uniformly Bounded Gaps Let $R = \{v_1, v_2, \ldots, v_{|R|}\}$. Recall that $|R| = O(\sqrt{lsc \cdot d})$ since we only deal with heavy vertices. For the case of reporting edges between two

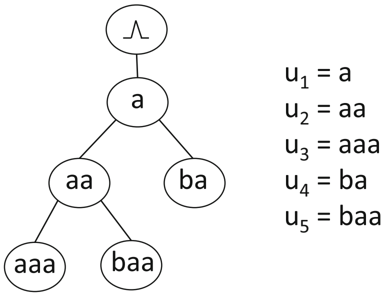


Fig. 2 The tree structure for $\{u_1, \ldots u_5\} \in L$

heavy vertices, where the dictionary has uniformly bounded gaps, the data structures used are:

1. For every edge $e = (u, v_i) \in E_D$, **a pointer** $next(e)$ points to an edge $e' = (u', v_i)$ where u' is the lowest proper ancestor of u in T such that $(u', v_i) \in E_D$. If no such vertex u' exists then $next(e) = null$.
2. For every vertex $u \in L$, **an array** $A_u[]$ of size $|R|$ is built, where $A_u[i]$ contains a pointer to a list of all edges $(u', v_i) \in E_D$ for all u' ancestors of u in T (which may be u). If $e = (u, v_i) \in E_D$, then $A_u[i]$ points to $e = (u, v_i)$. Otherwise, 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 $(u', v_i) \in E_D$ and if no such edge exists then $A_u[i] = null$. The list of all u's ancestors of u in T that have edges $(u', v_i) \in E_D$ is obtained through the edge pointed to by $A_u[i]$ and its $next(\cdot)$ pointers.
3. For each vertex $v \in R$, **an ordered reporting list** $\mathcal{L}_v$ maintaining $A_u[v]$ entries, implying pointers to edges (u, v), where $u \in L$, is a responsible-neighbour of v, that arrived at least α and at most $\beta + M$ time units ago, without repetition.
4. For each vertex $u \in L$, **an ordered list** τ_u of the time stamps u arrived, when u was associated with the longest subpattern recognized at the time. The time stamps are within the current active window.
5. The **list** $\mathcal{L}_\beta$ of the last $\beta + M$ vertices $u \in L$ that were associated with the longest subpattern recognized at their recognition time. They are delayed for α time units before they are considered.
6. The New **array** of size $|L|$, where $New[u]$ is the newest occurrence of the subpattern associated with vertex u, that was delayed α time units.

For example, consider the subpatterns from Fig. 2, $u_1 = a, u_2 = aa, u_3 = aaa, u_4 = ba, u_5 = baa$. Figure 3 depicts their $A_u[]$ arrays, based on edges between the nodes represented by these subpatterns. According to these arrays, when baa arrives, we add the implicit linked list of baa, aa, a to the beginning of $\mathcal{L}_{ba}$ as these subpatterns have an edge connected to ba.

Constructing A_u Arrays. The A_us arrays can be constructed in BFS order over T. Each A_u array can be constructed by filling the entries corresponding to edges $e = (u, v_i)$ with a pointer to e and filling the rest of the entries by $A_u[i] = A_{u'}[i]$, where u' is the closest ancestor of u in T. Yet, such a construction consumes $O(lsc \cdot d)$ preprocessing time and space. In order to decrease the space requirement we maintain arrays only for specially chosen $O(\sqrt{\frac{d}{lsc}})$ vertices in T, such that during the query

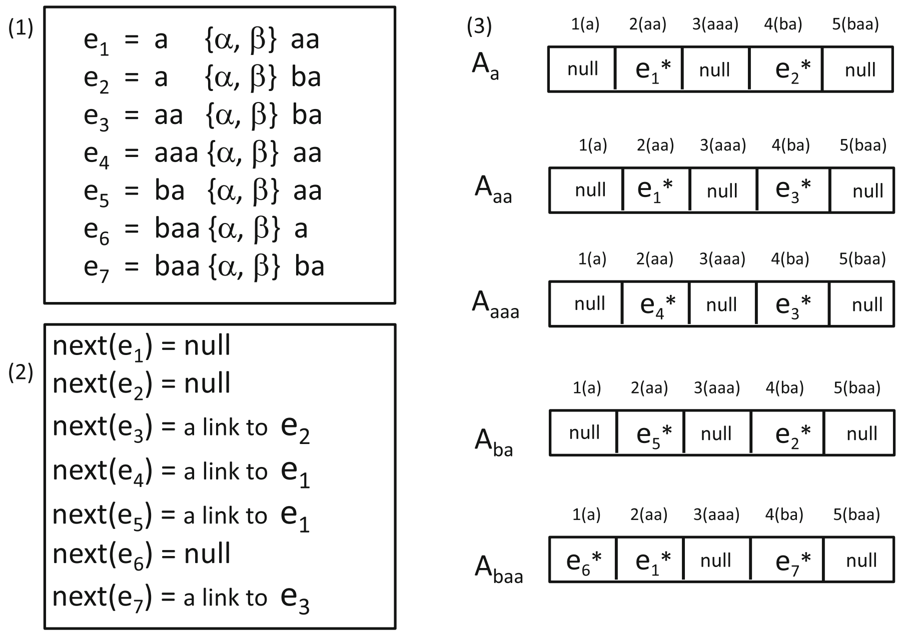


Fig. 3 An example of $A_u[]$ arrays. (1) Some edges emanating from $\{u_1 = a, u_2 = aa, u_3 = aaa, u_4 = ba, u_5 = baa\} \in L$. (2) The $next(e)$ pointers of the edges in (1). (3) The arrays of $u_1, \dots u_5$, where e_i^* refers to a link to edge e_i

phase whenever we need array $A_u[]$ for some vertex u, where u is not special, we construct it during the query in $O(\sqrt{lsc \cdot d})$ time. The construction of the A_u array, given only the arrays of the special vertices uses the following procedure.

1. Initialize all $A_u[i]$ entries to $null$.
2. Let $u' = u$
3. While u' is not a spacial vertex do
   (a) For every edge $e' = (u', v_i) \in E_D$, if $A_u[i] = null$, then set $A_u[i] = (e')^*$, where $(e')^*$ is a pointer to e'.
   (b) $u' \leftarrow$ the closest ancestor of u' in T.
4. If u' is a special vertex, for every entry where $A_u[i] = null$, set $A_u[i] = A_{u'}[i]$.

Choosing Special Vertices Let the weight of a node $u \in L$, denoted by $weight(u)$, be the degree of u in G_D. Notice that the weight of any vertex is at least $\sqrt{\frac{d}{lsc}}$ since all of the vertices are assumed to be heavy. In the preprocessing, we partition T into $O(\sqrt{\frac{d}{lsc}})$ small subtrees such that each subtree has total weight $\Theta(\sqrt{lsc \cdot d})$, except for possibly the subtree containing the root of T. The special vertices are the roots of these small subtrees. The partitioning is obtained by (greedily) peeling small subtrees in the bottom of T. Specifically, let T_u be the subtree of T rooted at u, and let $weight(T_u)$ be the total weight of vertices in T_u. Then, we iteratively peel a subtree T_u such that $weight(T_u)$ is at least $\sqrt{lsc \cdot d}$ but the total weight of each subtree of a child of u in T

is (separately) less than $\sqrt{lsc \cdot d}$. This peeling continues until no such subtree exists, so the remaining subtree must have total weight less than $\sqrt{lsc \cdot d}$ and is the last small subtree (also containing the root). The partitioning can be implemented in linear time using a post-order traversal. Notice that by the construction method, for any vertex u that is not the root of T, the total weight of vertices on the path from u to its closest proper ancestor that is a special vertex is $O(\sqrt{lsc \cdot d})$, since this path requires passing two subsequent special vertices if u itself is also a special vertex, otherwise, we need only find the special vertex which is the ancestor of the subtree u belongs to. This ancestor, by construction, has total weight $O(\sqrt{lsc \cdot d})$.

When the AC-automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a subpattern that is a suffix of the subpattern represented by state s,

1. If the arrived vertex is $v_i \in R$,
   (a) Let $e^* = A_u[i] = \mathcal{L}_{v_i}.first$, where e^* is a pointer to edge $e = (u, v_i)$.
   (b) While $A_u[i] \neq null$ and $\tau_u.first \geq t - m_v - \beta - 1$ for the currently considered (u, v_i),
      i. Let $U = u$, the vertex referred to by the current $A_u[i]$.
      ii. While $e \neq null$ do
         A. Report edge (u, v) according to the time stamps in τ_U.
         B. $e = next(e)$.
      iii. Proceed to the next $e^* = A_u[i]$ in $\mathcal{L}_v$ and return to step (b).
   (c) The edges for which v is their responsible-neighbour are scanned, and every assigned-neighbour u that has $New[u]$ equals the time stamp t' such that $t - m_v - \beta - 1 \leq t'$, the edge (u, v) is reported according to the time stamps in every $\tau_{u''}$, where u'' is a successor of u in T, including u itself.
2. If the arrived vertex is $u \in L$, and u is the vertex associated with the longest subpattern represented by state s at time t, then u is inserted into $\mathcal{L}_\beta$.

In addition, the *active window* is maintained by updating $\mathcal{L}_\beta$ and acknowledging arrived nodes $u \in L$ that become relevant.
For vertices $u \in L$, arriving exactly $\alpha + 1$ time units before time t, if u is the vertex associated with the longest subpattern represented by state s at time $t - \alpha - 1$,

1. If u is not a special vertex, array A_u is constructed.
2. For each v_i, where $A_u[i] \neq null$
   (a) $A_u[i]$ is added to the beginning of $\mathcal{L}_{v_i}$.
   (b) If τ_u is not empty, then the previous appearance of $A_u[i]$ is removed from $\mathcal{L}_{v_i}$.
3. $t - \alpha - 1$ is added to the beginning of τ_u.
4. If u is not a special vertex, array A_u is deleted.
5. For every u' that is an ancestor of u in the tree, $New[u'] = t - \alpha - 1$.

For vertices $u \in L$, arriving exactly $\beta + M + 1$ time units before time t, where u is the vertex associated with the longest subpattern represented by state s, at time $t - \beta - M - 1$,

1. u is removed from $\mathcal{L}_\beta$.
2. The time stamp $t - \beta - M - 1$ is removed from the end of τ_u.
3. In case τ_u becomes empty,
   (a) If u is not a special vertex, array A_u is constructed.
   (b) For each v_i, where $A_u[i] \neq null$, $A_u[i]$ is deleted from $\mathcal{L}_{v_i}$.
   (c) If u is not a special vertex, array A_u is deleted.

Lemma 3 *The DMOG problem with uniform gap borders for edges where both endpoints are heavy, can be solved with $O(\mathfrak{D})$ preprocessing time, $O(lsc+\sqrt{lsc \cdot d}+op)$ time per text character, and $O(\mathfrak{D} + (\beta + M))$ space.*

Proof In the preprocessing, the AC automaton is built in time linear in the size of the dictionary $\mathfrak{D}$. The $next(e)$ pointers are calculated by going over the tree T from the root down and assigning $next(e) = e'$ where $e = (u, v_i), e' = (u', v_i) \in E_d$ and u' is the closest proper ancestor of u, in total $O(d)$ time. The computation of the A_u arrays for all the special vertices in the preprocessing, is done in a top-down approach by first constructing the array for the root of T and then constructing each $A_u[]$ for a special vertex u only after the array for the closest proper special ancestor u' of u was constructed, using the construction procedure of constructing an array during a query. Due to the property that the total weights of vertices on the path from u to u' is $O(\sqrt{lsc \cdot d})$ the time to construct the array for each special vertex is $O(\sqrt{lsc \cdot d})$. Since the number of special vertices is $O(\sqrt{\frac{d}{lsc}})$ the total preprocessing arrays construction costs $O(d)$ time.

At query time t, in case the arrived vertex is $u \in L$, and it is associated with the longest subpattern recognised at time t by the AC automaton, constant time operations of updating τ_u, the insertion of u into $\mathcal{L}_\beta$ are performed. In addition all the lsc vertices, that are ancestors of u in T are updating their New value. For the insertion into L_{v_i}, again, we consider merely the longest subpattern that was recognized at time t, represented by vertex u and $A_u[i] \neq null$ is inserted into the beginning of $\mathcal{L}_{v_i}$ (or deleted from it after the appearances of u become not relevant). In case u is not a special vertex, the $A_u[]$ array is constructed. The time cost of this process is the total number of edges of vertices on the path from u to its closest ancestor that is a special vertex, and another $O(\sqrt{lsc \cdot d})$ time for initializing A_u and scanning $A_{u'}$. Since the total degree of vertices on the path from u to its closest special ancestor to be $O(\sqrt{lsc \cdot d})$, due to the selection of special vertices, the total time cost for constructing A_u is $O(\sqrt{lsc \cdot d})$.

The rest of the subpatterns recognized at time t are suffixes of the subpattern represented by u, thus their edges with v_i are included in the implicit $next$ list starting at $A_u[i]$. Since the number of heavy nodes is bounded by $\sqrt{lsc \cdot d}$, we get that handling a node of the form $u \in L$ at query time t requires $O(lsc + \sqrt{lsc \cdot d})$ time.

For every arrived vertex $v_i \in R$ we scan $\mathcal{L}_{v_i}$ and report the edges represented by each of the $A_u[i] \in \mathcal{L}_{v_i}$ by following their $next$ links, until we reach $A_u[i]$ where its $\tau_u.first$ is not within the appropriate time frame. Recall, that $A_u[i]$ is inserted into the beginning of L_{v_i} and previous appearances of $A_u[i]$ is removed from it, yielding L_{v_i} contains appearances of u and its ancestors in decreasing order of occurrence time. Thus, when going over L_{v_i}, the first appearance of $A_u[i]$ that is not within the appropriate time frame, implies the rest of the list contains older appearances that are not in the appropriate time frame as well, therefore the scan is terminated and the reporting time remains asymptotic to the output size. We report edges according to the time stamps appearing in τ_U, where U is the longest successor of the current responsible vertex u, that u is contained in the implicit list of $A_U[i]$, as the time stamps are updated only for the longest subpattern recognized at every location. In case u has additional appearances, there is an additional $A_{U'}[i]$ in $\mathcal{L}_{v_i}$, containing u in its implicit list, and including different time stamps in its $\tau_{U'}$.

In addition, we go over all the $\sqrt{lsc \cdot d}$, $u \in L$ assigned neighbors of v and report them, if their $New[u]$ is within the required gap. We report the appearance of the edges according to the τ_u list of the successors of u, as the time stamps are updated only for the longest subpattern recognized at every location, here again, the time stamps lists are maintained in decreasing ordered, thus reporting takes linear time in the size of the output. The union of the times required for the different cases of vertices concludes the proof of the query complexity.

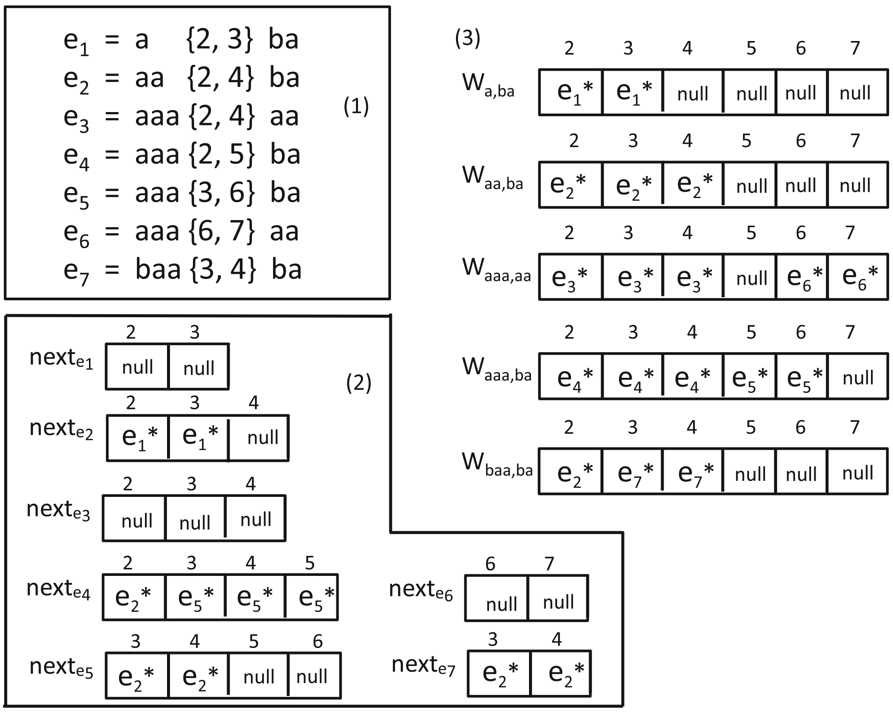
Regarding space: The AC automaton requires linear space in the size of the dictionary $O(\mathfrak{D})$. Since for each vertex $u \in L$, $A_u[i]$ appears only once in $\mathcal{L}_{v_i}$, the space consumption of all the $\mathcal{L}_v$ lists is $O(d)$. Each of the $O(\sqrt{\frac{d}{lsc}})$ special vertices maintains an A_u array of size $\sqrt{lsc \cdot d}$, thus the space consumption of the arrays is $O(d)$. The New array has $\sqrt{lsc \cdot d}$ entries, yet they are included in the size of the dictionary $O(\mathfrak{D})$. The additional space usage is required for the vertices maintained for $O(\beta - \alpha + M)$ time units by the τ_u lists, and additional $O(\alpha)$ is required for the $u \in L$ vertices maintained by $\mathcal{L}_\beta$ for α time units, until they can be considered a arrived. □

Lemmas 1 and 3 yield Theorem 11.

Theorem 11 *The DMOG problem with 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 As in the uniform case, let $R = \{v_1, v_2, \ldots, v_{|R|}\}$ where $|R| = O(\sqrt{lsc \cdot d})$. We leverage again the tree structure of T, where a vertex that arrives implies that all its ancestors arrived as well. Nevertheless, due to the implications of the non-uniform gaps, where the gap boundaries of an edge e are denoted by α_e and β_e, the suggested algorithm differs from from that of the $DMOG$ with uniformly bounded gaps, by focusing on updating a data structure related to vertex $v_i \in R$ by the occurrences of $u \in L$ where $e = (u, v_i) \in E_D$, according to the gap boundaries. Hence, different data structures are used in this case.

1. For each $e = (u, v_i) \in E_D$, **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 e' such that u' is the lowest ancestor of u in T (possibly u itself) where $e' = (u', v_i) \in E_D$ and $\alpha_{e'} \leq j \leq \beta_{e'}$. Additional requirement is that the pointers $next_e[j]$ do not form a loop. If no such edge exists then $next_e[j]$ is assigned $null$. (For simplicity, the indices of the array are treated as starting from α_e and ending at β_e).
   In order to guarantee that there is no loop, edges connecting the same vertices but having different bounds are lexicographically ordered by their bounds, having $next_e[j]$ point to the next appropriate edge in this ordering, if such an edge exists.



e1 = a {2, 3} ba
e2 = aa {2, 4} ba
e3 = aaa {2, 4} aa
e4 = aaa {2, 5} ba
e5 = aaa {3, 6} ba
e6 = aaa {6, 7} aa
e7 = baa {3, 4} ba
(1)
(2)
(3)
next_e1: 2 null, 3 null
next_e2: 2 e1*, 3 e1*, 4 null
next_e3: 2 null, 3 null, 4 null
next_e4: 2 e2*, 3 e5*, 4 e5*, 5 e5*
next_e5: 3 e2*, 4 e2*, 5 null, 6 null
next_e6: 6 null, 7 null
next_e7: 3 e2*, 4 e2*
W_a,ba: 2 e1*, 3 e1*, 4 null, 5 null, 6 null, 7 null
W_aa,ba: 2 e2*, 3 e2*, 4 e2*, 5 null, 6 null, 7 null
W_aaa,aa: 2 e3*, 3 e3*, 4 e3*, 5 null, 6 e6*, 7 e6*
W_aaa,ba: 2 e4*, 3 e4*, 4 e4*, 5 e5*, 6 e5*, 7 null
W_baa,ba: 2 e2*, 3 e7*, 4 e7*, 5 null, 6 null, 7 null

Fig. 4 An example of $A_u[]$ arrays. (1) Some edges of the dictionary, with non uniformly bounded gaps. (2) The $next_{e_i}$ arrays of the edges in (1). (3) $W_{u,v}$ arrays of some edges, where e_i^* refers to a link to edge e_i

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. $W_{u,i}[j]$ contains a pointer to $e' = (u', v_i)$ such that u' is the lowest ancestor of u in T (possibly u itself), where $e' = (u', v_i) \in E_d$ and $\alpha_{e'} \leq j \leq \beta_{e'}$. (thus, $W_{u,i}[j]$ contains a pointer to $e = (u, v_i) \in E_D$ if $\alpha_e \leq j \leq \beta_e$. If $\alpha_e > j$ or $j > \beta_e$ then $W_{u,i}[j]$ contains a pointer to e' for $e' = next_e[j]$.) If no such an edge exists, then $W_{u,i}[j]$ is assigned $null$. Hence, $W_{u,i}[j]$ gives access to a list of edges in E_D that: (a)touch v_i,(b) touch an ancestor of u in T, and (c) their boundaries contain j.
3. For each vertex $v_i \in R$, **a cyclic *active window* array** $Active_i$ of size $\beta^* - \alpha^* + M + 1$ is maintained, where $Active_i[j]$ is a pointer to a list of lists of edges that need to be reported if v_i appears in $j - 1$ time units from now.

An example of the $next_e$ arrays can be seen in Fig. 4, using the tree structure of Fig. 2. Note that $next_{e_4}[3]$ contains a link to e_5 while $next_{e_4}[2]$ contains a link to e_2, since the gap of two locations is legal to both e_4 and e_2 but not to e_5, while the gap of three locations is legal to e_5 as well, and the first subpattern of e_5, aaa, is the lowest common ancestor of the first subpattern of e_4, aaa, that has an edge to node ba.
The construction of the $W_{u,i}$ arrays is similar to the construction of A_u arrays described in the previous paragraph. In order to reduce space we maintain arrays only for specially chosen $O(\sqrt{\frac{d}{lsc}})$ vertices in T, such that whenever we need to construct an array $W_{u,i}$

for some vertices u, v_i during the query phase, where u is not special, we can construct $W_{u,i}$ in $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$ time.
The construction of the $W_{u,i}$ array, given only the $W_{u,i}$ arrays of the special vertices uses the following procedure.

1. Initialize all $W_{u,i}$ entries to $null$.
2. Let $u' = u$
3. While u' is not a spacial vertex
    (a) For every edge $e = (u', v_i) \in E_D$, and every $\alpha_e \leq j \leq \beta_e$ if $W_{u,i}[j] = null$, then set $W_{u,i}[j] = e^*$, where e^* is a pointer to e.
    (b) $u' \leftarrow$ the closest ancestor of u' in T.
4. If u' is a special vertex, for every entry such that $W_{u,i}[j] = null$, set $W_{u,i}[j] = W_{u',i}[j]$.

Choosing Special Vertices Let the weight of a node $u \in L$, denoted by $weight(u)$, be the number of entries $W_{u,i}[j] \neq null$. Notice that the total weight of all of the vertices in T is $O(d(\beta^* - \alpha^*))$ (as opposed to $O(d)$ in the A_u arrays of the uniform case). In the preprocessing, we partition T into $O(\sqrt{\frac{d}{lsc}})$ small subtrees such that each subtree has total weight $\Theta(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$, except for possibly the subtree containing the root of T. The special vertices are the roots of these small subtrees. The partitioning is obtained by (greedily) peeling small subtrees in the bottom of T, as was explained regarding the A_u arrays construction. The partitioning of the tree costs linear time using a post-order traversal. Also, the total weight of vertices on the path from u to its closest proper ancestor that is a special vertex is $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$, since this path requires passing two subsequent special vertices if u itself is also a special vertex, otherwise, we need only find the special vertex which is the ancestor of the subtree u belongs to. This ancestor, by construction, has total weight of $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$.

When the AC-automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows. For each $v_i \in R$,

1. The active window array $Active_i$ is shifted by one position, by incrementing its starting position in a cyclic manner.
2. $Active_i[\beta^* - \alpha^* + M + 1]$ is cleared (It could have been reported in the previous query when it was $Active_i[1]$).

For every vertex v associated with a subpattern that is a suffix of the subpattern represented by state s,

```
1. For every arrived vertex of the form $v_i \in R$,
    (a) Let $(W_{u,i}[j], j) = Active_i[1].first$
    (b) While $W_{u,i}[j] \neq null$ do
        i. Let $W_{u,i}[j]$ be a pointer to edge $e = (u', v_i)$ and $j$ the current gap.
        ii. while $e \neq null$,
            A. Report $e$ ending at $t$ (where its $u'$ ends at $t - m_{v_i} - j - 1$).
            B. $e = next_e[j]$
        iii. Proceed to the next $(W_{u,i}[j], j)$ in $Active_i[1]$ and return to step (b).
```

2. If the longest recognized subpattern at time t is represented by the arrived vertex $u \in L$,
   (a) If u is not a special vertex, array $W_{u,i}$ is constructed.
   (b) For each $v_i \in R$ and for each j, where $W_{u,i}[j] \neq null$,
      i. The pair $(W_{u,i}[j], j)$ is inserted to $Active_i[j + m_{v_i} + 1]$, where m_{v_i} is the length of the subpattern corresponding to v_i, (since in $j + m_{v_i}$ time units from now, if $v_i \in R$ arrives we need to report the edges pointed to by $W_{u,i}[j]$).
   (c) If u is not a special vertex, array $W_{u,i}$ is deleted.

Lemma 4 *The DMOG problem with non-uniform gap borders, for edges where both endpoints are heavy, 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 $O(\mathfrak{D} + d(\beta^* - \alpha^*) + \sqrt{lsc \cdot d}(\beta^* + M))$ space.*

Proof In the preprocessing, the AC automaton is built in time linear in the size of the dictionary $\mathfrak{D}$. The $next_e$ arrays are calculated by going over the tree T from the root to the bottom and assigning $next_e[j] = e'$ or $next_e[j] = next_{e'}[j]$ where $e = (u, v_i)$, $e' = (u', v_i) \in E_D$ and u' is the closest proper special ancestor of u. This procedure require $O(d(\beta^* - \alpha^*))$ time. Computing $W_{u',i}$ for all special vertices u' and all i is executed using a top-down approach, similar to the procedure detailed for the construction of such an array during a query. The time to construct the $W_{u,i}$ arrays for a special vertex and every v_i is $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$. Since the number of special vertices is $O(\sqrt{\frac{d}{lsc}})$ the total preprocessing phase costs $O(d(\beta^* - \alpha^*))$ time.

At query time t each of the lsc vertices recognized by the AC automaton are handled in case the vertex is $v_i \in R$. We scan $Active_i[1]$ and report all the edges represented by each of the $W_{u,i}[j]$ in it, by following their $next_e[j]$ links. Note that by the $next_e$ pointer arrays construction, for a given vertex $u \in L$ all the multi-edges $e = (u, v_i)$ and all edges (u', v_i), where u' is an ancestor of u in T, whose boundaries contain j, form an implicit list, by considering the pointers $next_e[j]$ for each such edge e. The elements in $Active_i[1]$ require no filtering, as they were inserted to a specific entry ($Active_i[j + m_{v_i} + 1]$), where the occurrence of u will form a legal edge with v_i in accordance with the gap boundaries of $e = (u, v_i)$. Hence, reporting the content of $Active_i[1]$ requires time linear in the size of the output.

In case the vertex is $u \in L$, we consider merely the longest subpattern that was recognized at time t, represented by vertex u. In case u is not a special vertex, the $W_{u,i}$ arrays are constructed. The time cost of this process is the total weight of vertices on the path, from u to its closest special ancestor, and another $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$ time for initializing $W_{u,i}$ for every v_i. As the total weight of vertices on the path from u to its closest special ancestor is $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$, the total time cost for constructing $W_{u,i}$ during a query is $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$. Given array $W_{u,i}$, for every $v_i \in R$ and every $\alpha^* \leq j \leq \beta^*$, $W_{u,i}[j] \neq null$ is inserted into the suitable entry of $Active_i$. Since the number of heavy vertices is bounded by $\sqrt{lsc \cdot d}$, such insertions require $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$ time. The rest of the subpatterns recognized at time t are suffixes of the subpattern represented by u, thus their edges with v_i are included in the implicit $next_e[j]$ pointers list starting at $W_{u,i}[j]$. Hence, handling a vertex of the form $u \in L$ at query time t requires $\tilde{O}(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$ time. The union of the times required for the different cases of vertices concludes the proof of the query complexity.

Regarding space: The AC automaton requires linear space in the size of the dictionary $\mathfrak{D}$. The space requirement for the $next_e$ pointer arrays is $\sum_{e \in E_D}(\beta_e - \alpha_e + 1) \leq d(\beta^* - \alpha^*)$. Every $Active_i$ array is of size $\beta^* - \alpha^* + M$ and we have $O(\sqrt{\frac{d}{lsc}})$ such arrays. In addition, there are $O(\sqrt{\frac{d}{lsc}})$ special vertices, for which we save their $O(\sqrt{lsc \cdot d})$ $W_{u,i}[j]$ arrays of size $O(\beta^* - \alpha^*)$ each, hence the space consumption of all the saved $W_{u,i}[j]$ arrays is $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$. The total space required is, therefore, as stated. □

Lemmas 2 and 4 yield Theorem 12.

Theorem 12 *The DMOG problem with non-uniform gap borders can be solved with* $O(\mathfrak{D} + d(\beta^* - \alpha^*))$ *preprocessing time,* $\tilde{O}(lsc + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + op)$ *time per query text character, and* $\tilde{O}(\mathfrak{D} + d(\beta^* - \alpha^*) + \sqrt{lsc \cdot d}(\beta^* + M))$ *space.*

References

1. Abboud, A., Williams, V.V.: Popular conjectures imply strong lower bounds for dynamic problems. In: Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS) (2014)
2. Alfred, V.A., Corasick, J.C.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
3. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. Assoc. Comput. Mach. (JACM) **42**(4), 844–856 (1995)
4. Amir, A., Farach, M., Idury, R.M., La Poutré, J.A., Schäffer, A.A.: Improved dynamic dictionary matching. Inf. Comput. **119**(2), 258–282 (1995)
5. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Text indexing and dictionary matching with one error. J. Algorithms **37**(2), 309–325 (2000)
6. Amir, A., Levy, A., Porat, E., Shalom, B.R.: Dictionary matching with one gap. In: Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 11–20 (2014)
7. Bansal, N., Williams, R.: Regularity lemmas and combinatorial algorithms. Theory Comput. **8**(1), 69–94 (2012)
8. Bille, P., Gørtz, I.L., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. Theor. Comput. Sci. **443**, 25–34 (2012)
9. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1297–1308 (2010)
10. Bjørklund, A., Pagh, R., Williams, V.V., Zwick, U.: Listing triangles. In: Proceedings of of 41st International Colloquium on Automata, Languages, and Programming (ICALP (I)), pp. 223–234 (2014)
11. Brodal, G.S., Gasieniec, L.: Approximate dictionary queries. In: Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 65–74 (1996)
12. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. (SICOMP) **14**(1), 210–223 (1985)
13. Cohen, H., Porat, E.: Fast set intersection and two-patterns matching. Theor. Comput. Sci. **411**(40–42), 3795–3800 (2010)
14. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of the 36 Annual Symposium on Theory of Computing (STOC), pp. 91–100 (2004)
15. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry, 2 reised edn, ch. Section 10.1: Interval Trees (ed.), p. 212217. Springer, Berlin (2000)

16. Fredriksson, K., Grabowski, S.: Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. Inf. Retr. **11**(4), 335–357 (2008)
17. Grønlund, A., Pettie, S.: Threesomes, degenerates, and love triangles. In: Proceedings of 55th IEEE Anuual Symposium on Foundation of Computer Science (FOCS), pp. 621–630 (2014)
18. Haapasalo, T., Silvasti, P., Sippu, S., Soisalon-Soininen, E.: Online dictionary matching with variable-length gaps. In: Proceedings of International Symposium on Experimental Algorithms (SEA), pp. 76–87 (2011)
19. Henzinger, M., Krinninger, S., Nanongkai, D., Saranurak, T.: Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In: Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC), pp. 21–30 (2015)
20. Hofmann, K., Bucher, P., Falquet, L., Bairoch, A.: The PROSITE database, its status in 1999. Nucl. Acids Res. **27**(1), 215–219 (1999)
21. Hon, W.-K., Lam, T.-W., Shah, R., Thankachan, S.V., Ting, H.-F., Yang, Y.: Dictionary matching with uneven gaps. In: Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 247–260 (2015)
22. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM J. Comput. (SICOMP) **7**(4), 413–423 (1978)
23. Kopelowitz, T., Pettie, S., Porat, E.: Dynamic set intersection. In: Proceedings of the 14th International Symposium on Algorithms and Data Structures (WADS) (2015)
24. Kopelowitz, T., Pettie, S., Porat, E.: Higher lower bounds from the 3-sum conjecture. In: Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (2016)
25. Kucherov, G., Rusinowitch, M.: Matching a set of strings with variable length don't cares. Theor. Comput. Sci. **178**(1–2), 129–154 (1997)
26. Morgante, M., Policriti, A., Vitacolonna, N., Zuccolo, A.: Structured motifs search. J. Comput. Biol. **12**(8), 1065–1082 (2005)
27. Mortensen, C.W.: Fully dynamic orthogonal range reporting on RAM. SIAM J. Comput. **35**(6), 1494–1525 (2006)
28. Eugene, W., Myers, G.: A four russians algorithm for regular expression pattern matching. J. ACM **39**(2), 430–448 (1992)
29. Myers, G., Mehldau, G.: A system for pattern matching applications on biosequences. CABIOS **9**(3), 299–314 (1993)
30. Navarro, G., Raffinot, M.: Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. J. Comput. Biol. **10**(6), 903–923 (2003)
31. Pătraşcu, M.: Towards polynomial lower bounds for dynamic problems. In: Proceedings of 42nd ACM Symposium on Theory of Computing (STOC), pp. 603–610 (2010)
32. VerInt.: Personal communication (2013)
33. Zhang, M., Zhang, Y., Liang, H.: A faster algorithm for matching a set of patterns with variable length don't cares. Inf. Process. Lett. **110**(6), 216–220 (2010)

Affiliations

Amihood Amir[1,2] · Tsvi Kopelowitz[1,3] · Avivit Levy[4] · Seth Pettie[3] · Ely Porat[1] · B. Riva Shalom[4]

✉ Avivit Levy
avivitlevy@shenkar.ac.il

Amihood Amir
amir@cs.biu.ac.il

Tsvi Kopelowitz
kopelot@gmail.com

Seth Pettie
ettie@umich.edu

Ely Porat
porately@cs.biu.ac.il

B. Riva Shalom
rivash@shenkar.ac.il

[1] Bar-Ilan University, Ramat Gan, Israel

[2] Johns Hopkins University, Baltimore, USA

[3] University of Michigan, Ann Arbor, USA

[4] Shenkar College, Ramat Gan, Israel