Dynamic Dictionary Matching in the Online Model[⋆]

Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat

Bar-Ilan University, Ramat-Gan, Israel.
golansh1@cs.biu.ac.il, kociumaka@mimuw.edu.pl,
kopelot@gmail.com, poratelу@cs.biu.ac.il

Abstract. In the classic *dictionary matching problem*, the input is a dictionary of patterns $\mathcal{D} = \{P_1, P_2, \ldots, P_k\}$ and a text T, and the goal is to report all the occurrences in T of every pattern from $\mathcal{D}$. In the *dynamic* version of the dictionary matching problem, patterns may be either added or removed from $\mathcal{D}$. In the *online* version of the dictionary matching problem, the characters of T arrive online, one at a time, and the goal is to establish, immediately after every new character arrival, which of the patterns in $\mathcal{D}$ are a suffix of the current text.
In this paper, we consider the *dynamic* version of the *online dictionary matching problem*. For the case where all the patterns have the same length m, we design an algorithm that adds or removes a pattern in $\mathcal{O}(m \log\log \|\mathcal{D}\|)$ time and processes a text character in $O(\log\log \|\mathcal{D}\|)$ time, where $\|\mathcal{D}\| = \sum_{P\in\mathcal{D}} |P|$. For the general case where patterns may have different lengths, the cost of adding or removing a pattern P is $\mathcal{O}(|P| \log\log \|\mathcal{D}\| + \log d/ \log\log d)$ while the cost per text character is $\mathcal{O}(\log\log \|\mathcal{D}\| + (1 + \mathit{occ}) \log d/ \log\log d)$, where $d = |\mathcal{D}|$ is the number of patterns in $\mathcal{D}$ and *occ* is the size of the output. These bounds improve on the state of the art for dynamic dictionary matching, while also providing online features. All our algorithms are Las-Vegas randomized and the time costs are in the worst-case with high probability. A by-product of our work is a solution for the *fringed colored ancestor problem*, resolving an open question of Breslauer and Italiano [J. Discrete Algorithms, 2013].

1 Introduction

In the classic *dictionary matching problem* [2,3,4,5,6,7,8,11,13,15,17,18,19,20,22], the input is a dictionary of patterns $\mathcal{D} = \{P_1, P_2, \ldots, P_{|\mathcal{D}|}\}$ and a text T, both over alphabet Σ, and the goal is to report all the occurrences of patterns from $\mathcal{D}$ in T. The dictionary matching problem is one of the most fundamental pattern matching problems and was already considered in the 70s [1]. For example, one of the crucial components of Network Intrusion Detection Systems (NIDS) is the

⋆ This research is supported by ISF grants no. 824/17 and 1278/16 and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064). The authors thank Tatiana Starikovskaya for early discussions on the online dynamic dictionary problem.

ability to detect the presence of viruses and malware in streaming data. This task is typically executed by searching for occurrences of special digital signatures which indicate the presence of harmful intent. However, while searching for one such signature is a relatively simple task, NIDS have to deal with the task of searching for many signatures in parallel, which is exactly the dictionary matching problem where the patterns are the special digital signatures. Indeed, the task of finding these signatures dominates the performance of such security tools [25], and several practical approaches have been suggested [9,24].

A common objective in many dictionary matching applications is to support *online* arrivals of the text characters. In NIDS, for example, the typical goal is to report a special digital signature as soon as it appears, before the next packet of data arrives. To this end, in the *online* version of the dictionary matching problem, the text T arrives one character at a time, and the goal is to report all of the occurrences of patterns from $\mathcal{D}$ in T as soon as they appear, before the next character arrives. Thus, if an occurrence of a pattern $P \in \mathcal{D}$ ends at the ith character of T, then P must be reported before the $(i+1)$th character arrives.

Online Dynamic Dictionary Matching Problem. Another common task in applications that utilize dictionary matching is to support changes to $\mathcal{D}$. In NIDS, for example, one might introduce new digital signatures on the fly or remove from consideration digital signatures creating too many false positives. To this end, this paper focuses on the *online dynamic dictionary matching problem*, where the goal is to solve the online dictionary problem so that patterns may be added or removed from $\mathcal{D}$ between the arrivals of text characters. In particular, the requirement now is that when the ith character $T[i]$ arrives, the algorithm must report all patterns in the *current* $\mathcal{D}$ with an occurrence that ends at the ith character of T, and this reporting must take place before $T[i+1]$ arrives.

In this work, we investigate the online dynamic dictionary matching problem with a focus on randomized Las-Vegas algorithms, with worst-case high probability[1] guarantees on the runtime. A summary of our results follows.

1.1 Previous Work and New Results.

For (offline) dictionary matching the Aho–Corasick (AC) data structure [1] provides an optimal linear time solution. However the AC data structure was not initially designed for being neither dynamic nor online.

Let $\|\mathcal{D}\| = \sum_{P\in\mathcal{D}} |P|$ be the total length of all patterns in $\mathcal{D}$, let $d = |\mathcal{D}|$ be the number of patterns in $\mathcal{D}$, and let *occ* denote the size of the output.

Dynamic Dictionary Matching. In the following we provide an overview of previous work on dynamic dictionary matching that does not support online reporting. Amir et al. [3] improved on Amir et al. [2] by designing a dynamic dictionary matching data structure that allows for adding/deleting a pattern

[1] Throughout this paper, an event happens *with high probability* (whp) if the probability of the event not happening is polynomially small in the size of the input.

P in $O(|P| \log \|\mathcal{D}\| / \log\log \|\mathcal{D}\|)$ time, and processing a text T costs $O((|T| + occ) \log \|\mathcal{D}\| / \log\log \|\mathcal{D}\|)$ time. Sahinalp and Vishkin [23] provide an optimal solution for a restricted family of dictionaries[2]. Feigenblat et al. [14] sacrifice the update time in order to obtain a faster text processing time. Remarkably, there have been no significant improvements to date over these time bounds.

Online Dictionary Matching. The AC data structure is an online dictionary matching, but the time guarantee per text character is in an amortized sense. Kopelowitz et al. [22] provide an online data structure that is based on the AC data structure, and has a worst-case time of $O(\log\log |\Sigma|)$ per text character.

In recent years there has been a renewed interest in dictionary matching in the streaming model which inherently provide online solutions. Let m be the length of the longest pattern in $\mathcal{D}$. Clifford et al. [11] introduced an algorithm that costs $O(\log\log m)$ time per character, which was improved by Golan and Porat [20] to an algorithm that costs $O(\log\log |\Sigma|)$ time per character.

Our Results: Online Dynamic Dictionary Matching. While there has been a large body of work on either online or dynamic dictionary matching, not much is known about the combination, beyond trivial solutions. In order to simplify the description of our results, we first focus on the special case in which all of the patterns in $\mathcal{D}$ have the same length.

Theorem 1. *Let $m > 0$ be a positive integer and let $\mathcal{D}$ be a dictionary where each $P \in \mathcal{D}$ has $|P| = m$. Then there exists a linear-space data structure for online dynamic dictionary matching that supports the following operations on $\mathcal{D}$ (the worst-case time costs are given in parentheses):*

1. insert(P) *(assumes $|P| = m$): Insert a pattern P into $\mathcal{D}$ ($O(m \log\log \|\mathcal{D}\|)$ with high probability).*
2. delete(P)*: Delete a pattern $P \in \mathcal{D}$ ($O(m \log\log \|\mathcal{D}\|)$ with high probability).*
3. online_search(T)*: Report all of the occurrences of patterns from P in $\mathcal{D}$ in an online fashion, processing one character at a time ($O(\log\log \|\mathcal{D}\|)$ time per character).*

Moreover, the updates to $\mathcal{D}$ can take place during an online_search *operation. If such an update takes place during an* online_search *operation, then from that time onwards the algorithm applies the update to the output.*

Notice that the time costs of our algorithm are only a $\log\log \|\mathcal{D}\|$ factor away from optimal, which is an exponential improvement over [3] and [2]. Moreover, our algorithm is online and has worst-case guarantees per text character.

For the case of a general dictionary $\mathcal{D}$, we present the following algorithm.

[2] Sahinalp and Vishkin [23] claim that their results can be extended to general dictionaries, but the details were left for a full version that was never made available. The missing details are unclear and do not seem to be straightforward.

Theorem 2. *There exists a linear-space data structure for online dynamic dictionary matching that supports the following operations on a dictionary $\mathcal{D}$ (the worst-case time costs are given in parentheses):*

1. insert(P)*: Insert a new pattern P into $\mathcal{D}$ ($O(|P| \log\log \|\mathcal{D}\| + \log d/ \log\log d)$ with high probability).*
2. delete(P)*: Delete a pattern $P \in \mathcal{D}$ ($O(|P| \log\log \|\mathcal{D}\| + \log d/ \log\log d)$ with high probability).*
3. online_search(T)*: Report all of the occurrences of patterns from P in $\mathcal{D}$ in an online fashion, processing one character at a time ($O(\log\log \|\mathcal{D}\| + (1 + occ) \log d/ \log\log d)$ time per character).*

Moreover, the updates to $\mathcal{D}$ can take place during an online_search *operation. If such an update takes place during an* online_search *operation, then from that time onwards the algorithm applies the update to the output.*

The main difference between Theorem 1 and Theorem 2 is that when the lengths of patterns are all the same then there can be at most one pattern whose occurrence ends at a given text location, while if the lengths of the patterns are different then there could be several patterns whose occurrence ends at a given location. This difference ends up costing a factor of $\log d/ \log\log d$ per output element and per text location in the general case. Nevertheless, this overhead is a significant improvement compared to the $\log \|\mathcal{D}\|/ \log\log \|\mathcal{D}\|$ overhead from [3].

Fringe Colored Ancestors. As a consequence of our data structure, we also address an open problem described in [10] which is known as the *fringe colored ancestors* problem. The problem definition and solution are given in Section 3.

2 Algorithmic Preliminaries

2.1 The Aho-Corasick Data Structure

The following provides an overview of the AC algorithm [1], which is helpful for gaining intuition for our new algorithm. The AC algorithm stores a trie of all of the patterns in $\mathcal{D}$ where each edge is marked with the corresponding character from Σ. Each node u in the trie corresponds to a string $S(u)$ that is a prefix of some pattern (possibly more than one) from $\mathcal{D}$ and is comprised of the concatenation of the characters on the unique simple path from the root to u. Thus, if r is the root, then $S(r)$ is the empty string. Moreover, each node u in the trie has at most $|\Sigma|$ edges to children of u, each edge corresponding to a different character from Σ, which are called *forward-links*. In addition, each non-root node u has a *failure-link* which points to a node u' if and only if $S(u')$ is the longest prefix of some pattern from $\mathcal{D}$ that is also a proper suffix of $S(u)$. Notice that a failure-link exists for every non-root node in the trie since the string corresponding to the root is the empty string. Finally, each node u stores a *reporting-link* to the node $\hat{u}$, if such a node exists, where $S(\hat{u})$ is the longest proper suffix of $S(u)$ such that $S(\hat{u}) \in \mathcal{D}$ (that is, $\hat{u}$ represents a pattern in $\mathcal{D}$).

In order to solve the dictionary matching problem, the AC algorithm starts from the root of the trie and scans T. Suppose that after scanning $i-1$ characters of T, the algorithm is at node u and the ith character is c. If there exists a child x of u where the edge (u, x) is marked with c, then the algorithm transitions to x. Otherwise, if u is the root then the algorithm continues to the next text character, and if u is not the root then the algorithm follows the failure-link (u, u'), sets u to be u', and attempts to transition from the new u with c (this last step is recursive which may entail following additional failure-links if needed). Once a transition is successful, then the algorithm uses reporting-links to report all of the occurrences of patterns in T that end at the ith location.

Using standard amortization arguments together with hashing, it is straightforward to show that the total time for scanning a text T is $O(|T| + occ)$ time, where *occ* is the size of the output.

Worst-Case Versions of the AC Data Structure. The main downside of the AC data structure regarding the time cost is that the *worst-case* (as opposed to amortized) time for treating a single character from T could be $\Omega(\max_{P\in\mathcal{D}}\{|P|\})$. One straightforward way of reducing this worst-case time is to store a *direct failure-link* for each pair of a node u and a character c for which there is no forward-link for u that has character c. The direct failure-link for u and c points to a node v such that $S(v)$ is the longest suffix of $S(u)c$. The direct failure-links allow for a constant worst-case time per character, but the space usage becomes $O(n|\Sigma|)$.

Challenges with the Dynamic Setting. When considering the dynamic dictionary case, one of the challenges in using an AC-like data structure is that a single update to $\mathcal{D}$ has the potential of updating a large number of failure-links (whether we are using direct failure-links or not).

2.2 Generalized Suffix Tree

The backbone of the new data structure is the *generalized suffix tree* (GST) $\mathsf{T}(\mathcal{D})$ of the patterns in $\mathcal{D}$, which is also the main component of the solution by Amir et al. [2] and its subsequent improved variants [3]. The GST $\mathsf{T}(\mathcal{D})$ is a compacted trie of the suffixes of the patterns $P \in \mathcal{D}$. That is, we construct the trie containing the suffixes of the patterns $P \in \mathcal{D}$ and compress every non-branching path with no terminal inner nodes (a path whose internal nodes have a single child and do not represent suffixes of the patterns) into a single edge. Since edges in the compressed trie may correspond to multiple characters (which happens when the edge represents a non-branching path in the uncompacted trie), the string corresponding to a given edge is stored implicitly as a pointer to a fragment of one of the patterns $P \in \mathcal{D}$. By standard arguments the size of $\mathsf{T}(\mathcal{D})$ is $O(\|\mathcal{D}\|)$.

Notice that the nodes of $\mathsf{T}(\mathcal{D})$ form a subset of the nodes of the underlying uncompacted trie. A node u of the uncompacted trie is *explicit* if u is in $\mathsf{T}(\mathcal{D})$, and otherwise u is *implicit*. As in the AC data structure, for a node u (implicit or explicit), let $S(u)$ denote the unique string corresponding to u, which is obtained by concatenating the labels of edges on the path from the root to u.

For each substring S of a pattern in $\mathcal{D}$, there exists either an implicit or explicit node u such that $S(u) = S$; we call u the *locus* of S. Since the locus might be implicit, u is represented by a pair (v, β), where v is the nearest explicit *descendant* of u and β is the *offset*—the length of the path from u to v in the uncompacted trie. Notice that if u is explicit, then u is represented as $(u, 0)$.

Suffix Links. A crucial and extremely helpful feature of suffix trees is that *suffix-links* are stored in explicit nodes. The *suffix-link* from a non-root node u is a pointer to the node v such that $S(u) = cS(v)$ for some $c \in \Sigma$. The suffix-links play a role similar to that of failure-links in the AC data structure: $S(v)$ is the longest *substring* of some pattern in $\mathcal{D}$ that is also a proper suffix of $S(u)$ [3]. By standard suffix-tree arguments, if u is explicit, then v is also explicit.

The suffix-links of $\mathsf{T}(\mathcal{D})$ span a tree $\mathsf{SLT}(\mathcal{D})$ on the set of implicit and explicit nodes of $\mathsf{T}(\mathcal{D})$. The *suffix-link path* from a node u (either implicit or explicit) in $\mathsf{SLT}(\mathcal{D})$, denoted by π_u, is the path in $\mathsf{SLT}(\mathcal{D})$ from u to the root of $\mathsf{SLT}(\mathcal{D})$. Notice that π_u may contain implicit nodes (but only if u is implicit).

GST Construction. Amir et al. [2] showed that a GST (and its suffix-links) can be maintained efficiently while undergoing insertions and deletions of strings. The implementation of [2] was designed for constant-size alphabets, but the only issue with supporting large integer alphabets is the following primitive: given a node u and a character c, retrieve the outgoing edge from u whose label starts with c. Such a primitive can be implemented in $O(1)$ worst-case time using dynamic hash tables [16], which also support updates in $O(1)$ time whp.

Lemma 3 (Dynamic GST; Amir et al. [2]). *A generalized suffix tree $\mathsf{T}(\mathcal{D})$, including the suffix-links for explicit nodes, can be maintained in linear space so that inserting a string to $\mathcal{D}$ takes $O(|P|)$ time with high probability.*

The Extended GST of the Reversed Dictionary. The *reversed dictionary* of $\mathcal{D}$ is the dictionary $\mathcal{D}^R = \{P^R : P \in \mathcal{D}\}$, where P^R is the string obtained by reversing P. Notice that $\mathsf{SLT}(\mathcal{D})$ is isomorphic to the generalized suffix *trie* of $\mathcal{D}^R$, which is the uncompacted version of $\mathsf{T}(\mathcal{D}^R)$. Specifically, a node u that is either explicit or implicit in $\mathsf{T}(\mathcal{D})$ corresponds to a unique node v (either implicit or implicit) in $\mathsf{T}(\mathcal{D}^R)$ such that $S(v) = (S(u))^R$. Due to the isomorphism, we denote $v = u^R$. In this work, $\mathsf{T}(\mathcal{D}^R)$ is represented using an extension $\mathsf{E}(\mathcal{D}^R)$ (the E stands for "extension") obtained by explicitly adding all implicit nodes of $\mathsf{T}(\mathcal{D}^R)$ which correspond to explicit nodes of $\mathsf{T}(\mathcal{D})$. We emphasize that while this is not the first time that the generalized suffix trie of $\mathcal{D}^R$ [14,22] is used for solving dictionary matching, as far as we know this is the first time that an algorithm uses $\mathsf{E}(\mathcal{D}^R)$.

[3] This is in contrast to the AC failure-links, which lead to the locus of the longest *prefix* of some pattern in $\mathcal{D}$ that is also a proper suffix of $S(u)$.

2.3 Traversing the Generalized Suffix Tree

As text characters arrive, the GST enables maintaining the *main pointer*—the locus of the longest suffix of T that is a substring of some pattern in $\mathcal{D}$ [4]. In an online solution, the locus must be updated before the next character arrives.

Suppose that the main pointer is at a node u. When the next character c arrives, the algorithm must find the node v representing the longest suffix of Tc. Notice that v is the locus of the longest suffix of $S(u)c$ that is also a substring of a pattern in $\mathcal{D}$. If u has a forward-link ℓ labeled with c, then v is the other endpoint of ℓ, and finding v costs $O(1)$ time using standard suffix tree traversal methods. Otherwise, a naïve way of finding v is to follow the path of suffix-links, starting from u, until reaching a node u' with a forward-link ℓ' labeled with c, and then to traverse ℓ'. If such a node does not exist (which is only possible if c does not occur in any $P \in \mathcal{D}$), then v is the root.

If an amortized bound for processing a character of T is sufficient, then one can traverse the suffix-link path from u (while suffix-links are not stored for implicit nodes, they can be simulated in constant amortized time). Our goal is to obtain efficient worst-case running time, so a different solution is needed. The main idea is to implement *direct failure-links*, which allow to directly reach v from u and c. Formally, given a node u and a character $c \in \Sigma$, the c-labeled direct failure-link leads from u to the locus of the longest suffix of $S(u)c$ occurring as a substring of a pattern $P \in \mathcal{D}$.

As observed above, the task of simulating a direct failure-link reduces to finding the lowest ancestor u' of u in $\mathsf{SLT}(\mathcal{D})$ that has a c-labeled forward-link. If u' exists, then, by standard suffix-tree arguments, all ancestors u'' of u' in $\mathsf{SLT}(\mathcal{D})$ must also have a c-labeled forward-link. In other words, the nodes with a c-labeled forward-link satisfy the so-called *fringe property* [10] on $\mathsf{SLT}(\mathcal{D})$: a set M of nodes in a rooted tree T satisfies the fringe property if $v \in M$ implies that either v is the root of T or the parent of v is also in M. Moreover, the ancestors of u in $\mathsf{SLT}(\mathcal{D})$ define π_u, and π_u corresponds to the path in $\mathsf{T}(\mathcal{D}^R)$ from u^R to the root. Thus, the task of finding u' can be expressed in terms of *fringe colored ancestor* queries (see Section 3 for a formal definition) in $\mathsf{T}(\mathcal{D}^R)$. Moreover, the algorithm uses $\mathsf{E}(\mathcal{D}^R)$, instead of $\mathsf{T}(\mathcal{D}^R)$, since if u does not have a c-labeled forward-link, then u' must be explicit in $\mathsf{T}(\mathcal{D})$ (see Claim 8), and therefore u'^R is explicit in $\mathsf{E}(\mathcal{D}^R)$.

To conclude, traversing the GST reduces to simulating direct failure-links from explicit and implicit nodes of $\mathsf{T}(\mathcal{D})$. A subroutine designed for this task is among our main technical contributions and is described in Section 4.

[4] That is, for each *prefix* of the text, the algorithm finds the longest *suffix* that is a substring of some pattern in $\mathcal{D}$. This is in contrast to [2] where the goal is to find, for each *suffix* of T, the longest *prefix* of the suffix that is a substring of some pattern in $\mathcal{D}$. Notice that in general the sets of these strings is not necessarily the same.

2.4 Reporting the Patterns

As in previous amortized procedures for scanning the text T [2,3], the most expensive phase is reporting the patterns. If the main pointer is to a node u, then the loci of the patterns to be reported lie on π_u. The algorithm maintains a mark on each node v in $\mathsf{SLT}(\mathcal{D})$ where $S(v) \in \mathcal{D}$, and so the reporting procedure reduces to the task of repeatedly finding lowest marked ancestors in $\mathsf{SLT}(\mathcal{D})$, starting from u. However, finding the marked ancestors is performed on $\mathsf{E}(\mathcal{D}^R)$, since all of the terminal nodes of $\mathsf{T}(\mathcal{D})$ correspond to explicit nodes in $\mathsf{E}(\mathcal{D}^R)$.

While any algorithm for reporting marked ancestors could be used, in order to derive our main results, we tweak existing marked ancestor algorithms. The proof of Theorem 1 exploits the fact that a locus of a pattern P is marked only when P is inserted, and so when marking the locus of P, one can spend time proportional to $|P|$ which is also the depth of the locus. Furthermore, due to uniform pattern lengths, there is at most one ancestor to be reported. The proof of Theorem 2 uses a new algorithm for reporting marked ancestors whose cost depends on the number of marked nodes d rather than on the tree size $\|\mathcal{D}\|$.

In Section 5 we provide more details on how to report the patterns.

2.5 Updates to the Dictionary

Insertions. When a pattern P is inserted into $\mathcal{D}$, the GST, the data structure for implementing direct failure-links, and the reporting data structure need to be updated accordingly. In addition, the main pointer may need to be updated (because a longer suffix of the current T may now occur as a substring of P). If the node u represented by the main pointer is at depth at least $|P|$, then the main pointer does not need to change. However, if the depth of u is less than $|P|$, then right after P is inserted into the GST, the algorithm traverses from the root of the GST with the last $|P|$ characters of T (the time cost of the traversal is amortized over the time cost of inserting P).

Deletions. The only immediate effect of deleting a pattern P is that P should not be reported anymore, but the suffixes of P may remain in the GST for some time. Thus, when deleting P, the only immediate consequence is that the algorithm removes P from the reporting data structure. Since we are interested in a linear space solution, the algorithm employs a lazy approach for updating the GST and the data structure for implementing direct failure-links by using the standard doubling technique: if the GST becomes too large, the algorithm initiates a background process of constructing a new GST (including the auxiliary data structures) on the non-deleted patterns. By using the doubling technique, the algorithm avoids the technical details of explicitly supporting pattern deletions.

3 Fringe Colored Ancestors

In this section, we describe a solution for the *fringe colored ancestor* problem. We begin by recallling two results from the literature that we shall use as subroutines.

Lemma 4 (Kopelowitz [21, Theorem 5.1]). *There exists a linear-space data structure maintaing an ordered list L and disjoint subsets $S_1, \ldots, S_k \subseteq L$ that supports the following operations (the runtimes are given in parentheses*[5]*):*

1. $\mathsf{insert}(u, v)$*: Insert a new item u after a given item $v \in L$ ($O(1)$ time whp).*
2. $\mathsf{set\_insert}(v, i)$ *(assumes that $v \notin \bigcup_{j=1}^{k} S_j$): Insert an element $v \in L$ into S_i ($O(\log\log |L|)$ time whp).*
3. $\mathsf{pred}(v, i)$*: Locate the predecessor of $v \in L$ in S_i ($O(\log\log |L|)$ time whp).*

Lemma 5 (Cole and Hariharan [12, Theorem 8.1]). *There exists a linear-space data structure maintaining a dynamic rooted tree T subject to the following operations, supported in $O(1)$ time:*

1. $\mathsf{insert\_leaf}(u, v)$*: Insert a new leaf u as a child of a node v.*
2. $\mathsf{insert}(u, v)$*: Insert a new node u by subdividing the edge from a non-root node v to the parent of v.*
3. $\mathsf{LCA}(u, v)$*: Return the lowest common ancestor of two nodes u and v.*

We now describe the new component for fringe colored ancestors.

Lemma 6. *Let T be a rooted tree such that each node in T is* colored *using* 0 *or more colors from a set Δ. If a node u in T has 0 colors, then u is said to be uncolored. For each color $\delta \in \Delta$, let M_δ be the nodes of T colored with color δ. Let $N = |T| + \sum_{\delta \in \Delta} |M_\delta|$. Suppose that for each color $\delta \in \Delta$, the set M_δ satisfies the fringe property. Then there exists an $O(N)$-space data structure that supports each of the following operations in $O(\log\log N)$ time with high probability:*

1. $\mathsf{insert\_leaf}(u, v)$*: Insert a new leaf u as a child of node v.*
2. $\mathsf{insert}(u, v)$ *(assumes that v is a non-root uncolored node): Insert a new node u by subdividing the edge connecting node v and the parent of v.*
3. $\mathsf{color}(v, \delta)$ *(assumes that v is either the root or the parent of v is already colored with δ): Color node v with color δ, i.e., add v to M_δ.*
4. $\mathsf{colored\_ancestor}(v, \delta)$*: Return the lowest ancestor of node v that is colored with color δ.*

Proof. The data structure uses Lemma 5 on T in order to support LCA queries. In addition, the data structure uses Lemma 4 to store the parenthesis representation of T, defined recursively as follows. Let r be the root of T and let $v_1, \ldots, v_k$ be the children of r (in an arbitrary order). The parenthesis representation of the tree is the concatenation of the representations of the subtrees rooted at $v_1, v_2, \ldots v_k$, wrapped with (at the beginning and) at the end. The two enclosing parentheses represent the root r.

We extend the parenthesis representation to represent T together with the colors of nodes in T, and denote this new representation by L. The representation L is defined as follows. For a node v with $c(v) > 0$ colors, instead of creating

[5] Theorem 5.1 in [21] is expressed in terms of expected runtimes. However, the only randomization is due to hashing, and the same time bounds hold whp.

just one pair of wrapping parentheses representing v, the parenthesis representation uses $c(v)+1$ nested pairs of parentheses, one additional pair for each color. Thus, each extra pair of parentheses is assigned a unique color from Δ, representing one of the colors of v. The algorithm maintains the invariant that the *innermost* parentheses for a node v are the uncolored parentheses. Each node $v \in T$ maintains pointers to the corresponding uncolored parentheses, and each (colored or uncolored) parenthesis stores a pointer to the corresponding node.

For each $\delta \in \Delta$, let $S_\delta \subseteq L$ be the set of parentheses in L corresponding to color δ. Notice that for different $\delta, \delta' \in \Delta$, the sets S_δ and $S_{\delta'}$ are disjoint, and thus the algorithm uses Lemma 4 to store the sets S_δ. The total space usage of all of the data structures is $O(|T|)$ machine words.

Supporting operations. To insert a new leaf u with parent v, the algorithm locates the closing uncolored parenthesis $\texttt{)}_v$ corresponding to v and adds a new pair of matching uncolored parentheses $\texttt{(}_u$ and $\texttt{)}_u$ immediately prior to $\texttt{)}_v$.

To subdivide the edge from v to its parent, the algorithm locates the uncolored parentheses corresponding to v and inserts an opening (closing) uncolored parenthesis $\texttt{(}_u$ ($\texttt{)}_u$) immediately before (after) $\texttt{(}_v$.

To add a color δ to a node v, the algorithm locates the uncolored parentheses $\texttt{(}_v$ and $\texttt{)}_v$ representing v, inserts a pair of parentheses $\texttt{(}_{v,\delta}$ and $\texttt{)}_{v,\delta}$ immediately before $\texttt{(}_v$ and after $\texttt{)}_v$, respectively, and adds both $\texttt{(}_{v,\delta}$ and $\texttt{)}_{v,\delta}$ to S_δ.

To answer a lowest colored ancestor query, the algorithm finds the node $x \in T$ corresponding to the predecessor in S_δ of the opening uncolored parenthesis $\texttt{(}_v$ representing v, and then returns $y = \mathsf{LCA}(x, v)$. The time cost is $O(\log\log N)$ time using the data structures of Lemmas 4 and 5.

The correctness of the query algorithm follows from the fringe property: Since x is colored with δ, then y is both an ancestor of v and colored with δ. Suppose by contradiction that there exists a lower ancestor u of v that is colored with δ. Then the opening δ-colored parenthesis of u appears between the parenthesis that defines x and $\texttt{(}_v$, contradicting x being the node corresponding to the predecessor of $\texttt{(}_v$ in S_δ. Thus, y is the lowest ancestor of v colored with δ. ◻

4 Simulating Direct Failure Links

We begin by proving two claims that are used in the proof of Lemma 9.

Claim 7. Let (x^R, y^R) be an edge in $\mathsf{T}(\mathcal{D}^R)$ with subsequent internal implicit nodes $z_1^R, \dots, z_k^R$. Then $z_1^R, \dots, z_k^R$ are either all explicit or all implicit in $\mathsf{E}(\mathcal{D}^R)$.

Proof. Denote $z_0^R = x^R$ and $z_{k+1}^R = y^R$. If a node z_i^R, for $1 \le i \le k$, is explicit in $\mathsf{E}(\mathcal{D}^R)$, then z_i is explicit in $\mathsf{T}(\mathcal{D})$, and the occurrences of $S(z_i)$ in $\mathcal{D}$ are followed by at least two distinct characters a, b. On the other hand, as z_i^R is implicit in $\mathsf{T}(\mathcal{D}^R)$, the occurrences of $S(z_i)$ in $\mathcal{D}$ are all preceded by the same character c. Hence, both a and b follow some occurrences of $S(z_{i+1}) = cS(z_i)$ in $\mathcal{D}$, so z_{i+1} is explicit in $\mathsf{T}(\mathcal{D})$ and z_{i+1}^R is explicit in $\mathsf{E}(\mathcal{D}^R)$. Furthermore, z_{i-1} is the target of the suffix-link from z_i, so z_{i-1} is explicit in $\mathsf{T}(\mathcal{D})$ and z_{i-1}^R is explicit in $\mathsf{E}(\mathcal{D}^R)$. By induction, $z_1^R, \dots, z_k^R$ are all explicit in $\mathsf{E}(\mathcal{D}^R)$ if at least one of them is. ◻

Claim 8. Let u be a node of $\mathsf{T}(\mathcal{D})$ with no c-labeled forward-link. If u' is the lowest ancestor of u in $\mathsf{SLT}(\mathcal{D})$ with a c-labeled forward-link, then u' is explicit in $\mathsf{T}(\mathcal{D})$.

Proof. If u is explicit in $\mathsf{T}(\mathcal{D})$, then all of the nodes in π_u are also explicit, and in particular u' is explicit. Otherwise, all of the nodes in π_u (including u) have a forward-link with another label $d \neq c$. Consequently, u' has at least two forward-links and thus u' is explicit in $\mathsf{T}(\mathcal{D})$. $\square$

Lemma 9. *There exists a data structure that augments* $\mathsf{T}(\mathcal{D})$ *with* $O(\|\mathcal{D}\|)$ *space, and supports inserting a string* P *into* $\mathcal{D}$ *in* $O(|P| \log\log \|\mathcal{D}\|)$ *time with high probability, and simulating direct failure-links in* $O(\log\log \|\mathcal{D}\|)$ *time.*

Proof. The proof begins by listing the components augmenting $\mathsf{T}(\mathcal{D})$.

Components. The first augmenting component is $\mathsf{E}(\mathcal{D}^R)$. Additionally, for every explicit node u of $\mathsf{T}(\mathcal{D})$, the data structure stores a bidirectional pointer between u and the corresponding node u^R in $\mathsf{E}(\mathcal{D}^R)$. If u has a c-labeled forward link, then u^R is colored with color c. Notice that every node on π_u is then explicit and has a c-labeled forward-link. Moreover, the nodes on π_u correspond to the ancestors of u^R in $\mathsf{E}(\mathcal{D}^R)$, and so all of the ancestors of u^R in $\mathsf{E}(\mathcal{D}^R)$ are colored with color c. Therefore, the set M_c of nodes in $\mathsf{E}(\mathcal{D}^R)$ with color c satisfies the fringe property, and so the algorithm augments $\mathsf{E}(\mathcal{D}^R)$ with the data structure of Lemma 6 to maintain the sets M_c, while supporting colored ancestor queries.

In order to support updating $\mathsf{E}(\mathcal{D}^R)$, the algorithm also maintains $\mathsf{T}(\mathcal{D}^R)$ and, for every explicit node of $\mathsf{T}(\mathcal{D}^R)$, a bidirectional pointer between the node and its counterpart in $\mathsf{E}(\mathcal{D}^R)$. Claim 7 classifies edges of $\mathsf{T}(\mathcal{D}^R)$ into two types depending on whether the internal nodes are all explicit or implicit in $\mathsf{E}(\mathcal{D}^R)$. If the internal nodes of an edge e from $\mathsf{T}(\mathcal{D}^R)$ are explicit in $\mathsf{E}(\mathcal{D}^R)$, then the data structure stores an array A_e with pointers to these explicit nodes in $\mathsf{E}(\mathcal{D}^R)$, and e maintains a pointer to A_e.

For each explicit node u of $\mathsf{T}(\mathcal{D})$, the algorithm maintains a pointer to a fragment[6] $P[\ell..r]$ of a pattern $P \in \mathcal{D}$ such that $P[\ell..r] = S(u)$. Finally, for each pattern $P \in \mathcal{D}$, and each $r \in \{1, \ldots, |P|\}$, the data strucutre stores a pointer $\lambda_{P,r}$ to the locus of $(P[1..r])^R$ in $\mathsf{E}(\mathcal{D}^R)$ (which is a terminal node).

The only non-trivial aspect of the space complexity is the size of the component of Lemma 6, which is $O(\|\mathcal{D}\|)$ since the number of colorings $\sum_{c\in\Sigma} |M_c|$ is bounded by the number of edges in the GST.

Implementing Direct Failure-links. Given a node u in $\mathsf{T}(\mathcal{D})$ and a character $c \in \Sigma$, let v be the target of the c-labeled direct failure-link from u. The algorithm locates v as follows. Recall from Section 2 that, if u' is the lowest ancestor of u in $\mathsf{SLT}(\mathcal{D})$ that has a c-labeled forward-link, then v is the target of the c-labeled forward-link from u'. Thus, the goal of the algorithm is to locate u' and then

[6] Recall that the pointers to fragments are one of the standard ways of storing edge labels in a GST.

return v using the hash table stored at u'. However, u' may not exist (in the case c does not occur in any pattern of $\mathcal{D}$). For simplicity, the algorithm description first assumes that u' exists and later shows how this assumption is supported.

If u has a c-labeled forward-link, then $u' = u$. Otherwise, notice that π_u contains u' and corresponds to a path in $\mathsf{E}(\mathcal{D}^R)$ from u^R to the root. Node u' can be either explicit or implicit. In either case, if the algorithm locates an *explicit* node w^R in $\mathsf{E}(\mathcal{D}^R)$ such that u' is the lowest ancestor of the (possibly implicit) corresponding node w in $\mathsf{SLT}(\mathcal{D})$ that has a c-labeled forward-link, then u'^R is the lowest ancestor of w^R colored with c (since u' is explicit in $\mathsf{T}(\mathcal{D})$ by Claim 8), and u' is located using a single colored ancestor query on w^R and c.

If u is explicit, then u^R satisfies the requirements for w^R. If u is implicit, let x be the highest explicit descendant of u in $\mathsf{T}(\mathcal{D})$. Recall that the main pointer provides direct access to x. Through x, the algorithm gains access to the fragment $P[\ell..r]$ of a pattern $P \in \mathcal{D}$ such that $P[\ell..r] = S(u)$. Let z be the locus of $P[1..r]$ in $\mathsf{T}(\mathcal{D})$. Notice that π_u is a sub-path of π_z and the sub-path of π_z from z to u does not contain any explicit nodes. Hence, u' is the lowest ancestor of z in $\mathsf{SLT}(\mathcal{D})$ that has a c-labeled forward-link, and so z satisfies the requirements for w. Therefore, w^R is the locus of $(P[1..r])^R$ in $\mathsf{E}(\mathcal{D}^R)$, which is accessible through $\lambda_{P,r}$.

Finally, if u' does not exist, then w'^R does not exist either. Thus, the algorithm identifies this case when executing the colored ancestor query. Such a case implies that c does not appear in any pattern from $\mathcal{D}$ (since the root of $\mathsf{E}(\mathcal{D}^R)$ is colored by all characters that appear in $\mathcal{D}$), and so v is the root of $\mathsf{T}(\mathcal{D})$.

s

Updates. Upon insertion of a pattern P, both $\mathsf{T}(\mathcal{D})$ and $\mathsf{T}(\mathcal{D}^R)$ are updated in $O(|P|)$ time using Lemma 3. Updates to $\mathsf{T}(\mathcal{D}^R)$ are reiterated in $\mathsf{E}(\mathcal{D}^R)$ as follows. If a new node u is inserted by subdividing an edge e, and the implicit nodes on e are implicit in $\mathsf{E}(\mathcal{D}^R)$ too, then the algorithm executes the same insertion in $\mathsf{E}(\mathcal{D}^R)$. Otherwise, u is explicit in $\mathsf{E}(\mathcal{D}^R)$, and so the array A_e already stores a pointer to u in $\mathsf{E}(\mathcal{D}^R)$. Note that the task of maintaining array A_e subject to splits is a straightforward task. The insertion of a leaf u is also straightforward since parent of u already knows its counterpart in $\mathsf{E}(\mathcal{D}^R)$.

The second phase of updates to $\mathsf{E}(\mathcal{D}^R)$ is to introduce an explicit node u^R for each new explicit node u of $\mathsf{T}(\mathcal{D})$. The new explicit nodes of $\mathsf{T}(\mathcal{D})$ are processed by non-decreasing depths, and so when processing a node u, the target v of the suffix-link from u is guaranteed to have already been processed; hence, v already has a pointer to v^R in $\mathsf{E}(\mathcal{D}^R)$. Notice that $S(u) = cS(v)$ for a character $c \in \Sigma$ and u^R is the target of the c-labeled forward-link from v^R (in $\mathsf{E}(\mathcal{D}^R)$). The algorithm retrieves u^R using the hash table stored at v^R. If u^R is an implicit node on an edge e in $\mathsf{E}(\mathcal{D}^R)$, then the algorithm converts all implicit nodes on e into explicit nodes so that Claim 7 holds at all times (including during the update process). The number of implicit nodes converted to explicit nodes is proportional to the length of e. Nevertheless, by Claim 7, the total number of nodes converted during an update is still $O(|P|)$.

However, every update to $\mathsf{E}(\mathcal{D}^R)$ imposes an update to the data structure of Lemma 6. Moreover, for every new edge (u, v) added to $\mathsf{T}(\mathcal{D})$, the algorithm retrieves the first character c of the edge label and colors u^R with color c. Thus, the total update time is $O(|P| \log\log \|\mathcal{D}\|)$ whp. $\square$

5 Reporting Patterns

Having updated the location u of the main pointer (i.e., the locus of the longest substring of some $P \in \mathcal{D}$ which occurs as a suffix of T), the algorithm reports all patterns which occur as suffixes of T. Notice that π_u contains the locus of every suffix of $S(u)$, so the task is reduced to reporting all patterns whose loci lie on π_u. The terminal nodes are explicit in $\mathsf{T}(\mathcal{D})$, so the corresponding nodes are explicit in $\mathsf{E}(\mathcal{D}^R)$, and the algorithm *marks* them.

If u is explicit, then the task reduces to reporting all marked ancestors of the corresponding node u^R of $\mathsf{E}(\mathcal{D}^R)$. Otherwise, via the same arguments as in Section 4, the locus of $(P[1..r])^R$, where $P[\ell..r] = S(u)$ for $P \in \mathcal{D}$, is used instead of u^R. In either case, the task reduces to reporting marked ancestors of a node w^R in $\mathsf{E}(\mathcal{D}^R)$.

If all the patterns are of the same length, the algorithm maintains marks in $\mathsf{E}(\mathcal{D}^R)$ for all ancestors of the loci of P^R for $P \in \mathcal{D}$, and so the marked nodes satisfy the fringe property. In this case, if w^R has an ancestor v^R representing a pattern, then v^R is the nearest marked ancestor of w^R, and v^R is found in $O(\log\log \|\mathcal{D}\|)$ time using a fringe marked ancestor data structure [10]. Upon inserting a pattern P, at most $O(|P|)$ nodes may need to be marked, which costs $O(|P| \log\log \|\mathcal{D}\|)$ time. This completes the proof of Theorem 1.

The proof of Theorem 2 is deferred to the full version of the paper.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Amir, A., Farach, M., Galil, Z., Giancarlo, R., Park, K.: Dynamic dictionary matching. J. Comput. Syst. Sci. **49**(2), 208–222 (1994)
3. Amir, A., Farach, M., Idury, R.M., Poutré, J.A.L., Schäffer, A.A.: Improved dynamic dictionary matching. Inf. Comput. **119**(2), 258–282 (1995)
4. Amir, A., Kopelowitz, T., Levy, A., Pettie, S., Porat, E., Shalom, B.R.: Mind the gap! Online dictionary matching with one gap. Algorithmica **81**(6), 2123–2157 (2019)
5. Amir, A., Levy, A., Porat, E., Shalom, B.R.: Dictionary matching with one gap. In: 25th Annual Symposium on Combinatorial Pattern Matching, CPM. LNCS, vol. 8486, pp. 11–20 (2014)
6. Amir, A., Levy, A., Porat, E., Shalom, B.R.: Dictionary matching with a few gaps. Theor. Comput. Sci. **589**, 34–46 (2015)
7. Athar, T., Barton, C., Bland, W., Gao, J., Iliopoulos, C.S., Liu, C., Pissis, S.P.: Fast circular dictionary-matching algorithm. Mathematical Structures in Computer Science **27**(2), 143–156 (2017)

8. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: 21st Annual Symposium on Combinatorial Pattern Matching, CPM. LNCS, vol. 6129, pp. 88–100 (2010)
9. Bremler-Barr, A., Hay, D., Koral, Y.: Compactdfa: Scalable pattern matching using longest prefix match solutions. IEEE/ACM Trans. Netw. **22**(2), 415–428 (2014)
10. Breslauer, D., Italiano, G.F.: Near real-time suffix tree construction via the fringe marked ancestor problem. J. Discrete Algorithms **18**, 32–48 (2013)
11. Clifford, R., Fontaine, A., Porat, E., Sach, B., Starikovskaya, T.A.: Dictionary matching in a stream. In: 23rd Annual European Symposium of Algorithms, ESA. LNCS, vol. 9294, pp. 361–372 (2015)
12. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. **34**(4), 894–923 (2005)
13. Feigenblat, G., Porat, E., Shiftan, A.: Linear time succinct indexable dictionary construction with applications. In: 2016 Data Compression Conference, DCC. pp. 13–22 (2016)
14. Feigenblat, G., Porat, E., Shiftan, A.: A grouping approach for succinct dynamic dictionary matching. Algorithmica **77**(1), 134–150 (2017)
15. Fischer, J., Gagie, T., Gawrychowski, P., Kociumaka, T.: Approximating LZ77 via small-space multiple-pattern matching. In: 23rd Annual European Symposium of Algorithms, ESA. LNCS, vol. 9294, pp. 533–544 (2015)
16. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with $O(1)$ worst case access time. J. ACM **31**(3), 538–544 (1984)
17. Ganguly, A., Hon, W., Sadakane, K., Shah, R., Thankachan, S.V., Yang, Y.: Space-efficient dictionaries for parameterized and order-preserving pattern matching. In: 27th Annual Symposium on Combinatorial Pattern Matching, CPM. LIPIcs, vol. 54, pp. 2:1–2:12 (2016)
18. Ganguly, A., Hon, W., Shah, R.: A framework for dynamic parameterized dictionary matching. In: 15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT. LIPIcs, vol. 53, pp. 10:1–10:14 (2016)
19. Golan, S., Kopelowitz, T., Porat, E.: Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In: 45th International Colloquium on Automata, Languages, and Programming, ICALP. LIPIcs, vol. 107, pp. 65:1–65:16 (2018)
20. Golan, S., Porat, E.: Real-time streaming multi-pattern search for constant alphabet. In: 25th Annual European Symposium on Algorithms, ESA. LIPIcs, vol. 87, pp. 41:1–41:15 (2017)
21. Kopelowitz, T.: On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In: 53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS. pp. 283–292 (2012)
22. Kopelowitz, T., Porat, E., Rozen, Y.: Succinct online dictionary matching with improved worst-case guarantees. In: 27th Annual Symposium on Combinatorial Pattern Matching, CPM. LIPIcs, vol. 54, pp. 6:1–6:13 (2016)
23. Sahinalp, S.C., Vishkin, U.: Efficient approximate and dynamic matching of patterns using a labeling paradigm. In: 37th Annual Symposium on Foundations of Computer Science, FOCS. pp. 320–328 (1996)
24. Tan, L., Sherwood, T.: A high throughput string matching architecture for intrusion detection and prevention. In: 32st International Symposium on Computer Architecture, ISCA. pp. 112–122 (2005)
25. Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic memory-efficient string matching algorithms for intrusion detection. In: 23rd IEEE International Conference on Computer Communications, INFCOM. pp. 2628–2639 (2004)