Succinct Online Dictionary Matching with Improved Worst-Case Guarantees

Tsvi Kopelowitz*[1], Ely Porat[2], and Yaron Rozen[3]

1 **University of Michigan, Ann Arbor, Michigan, USA**
kopelot@gmail.com

2 **Bar Ilan University, Ramat Gan, Israel**
porately@cs.biu.ac.il

3 **Bar Ilan University, Ramat Gan, Israel**
yaron1828@gmail.com

Abstract

In the online dictionary matching problem the goal is to preprocess a set of patterns $D = \{P_1, \dots, P_d\}$ over alphabet Σ, so that given an online text (one character at a time) we report all of the occurrences of patterns that are a suffix of the current text before the following character arrives. We introduce a succinct Aho-Corasick like data structure for the online dictionary matching problem. Our solution uses a new succinct representation for multi-labeled trees, in which each node has a set of labels from a universe of size λ. We consider lowest labeled ancestor (LLA) queries on multi-labeled trees, where given a node and a label we return the lowest proper ancestor of the node that has the queried label.

In this paper we introduce a succinct representation of multi-labeled trees for $\lambda = \omega(1)$ that support LLA queries in $O(\log\log\lambda)$ time. Using this representation of multi-labeled trees, we introduce a succinct data structure for the online dictionary matching problem when $\sigma = \omega(1)$. In this solution the worst case cost per character is $O(\log\log\sigma + occ)$ time, where occ is the size of the current output. Moreover, the amortized cost per character is $O(1 + occ)$ time.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Succinct indexing, dictionary matching, Aho-Corasick, labeled trees

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.6

1 Introduction

One of the crucial components of Network Intrusion Detection Systems (NIDS) is the 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. While searching for one such signature is often a fairly simple task, NIDS has to deal with the task of searching for many signatures in parallel. In such settings it is required that both the time spent on each packet of data and the total space usage are extremely small. Currently, the task of finding these signatures dominates the performance of such security tools [32], and several practical approaches have been suggested [9, 10]. The theoretical model for this problem is known as the (online) dictionary matching problem, which is a well studied problem [1, 2, 3, 4, 11, 13, 14] and is defined next.

* Work supported in part by NSF grants CCF-1217338, CNS-1318294, and CCF-1514383.

Dictionary matching. In the *dictionary matching problem* the input is a dictionary $D = \{P_1, P_2, ..., P_d\}$ of patterns and a text $T = t_1 t_2 ... t_N$, all over alphabet Σ, where $\sigma = |\Sigma|$. The goal is to list all pairs (i, j) such that $t_{i-|P_j|+1} ... t_i = P_j$. Let $n = \sum_{i=1}^{d} |P_i|$, and let $n_{max} = \max_{P \in D}\{|P|\}$. For a dictionary D the *prefix set* of D, denoted by $P(D)$, is the set of all prefixes of patterns in D. Let $m = |P(D)|$ and notice that $m \leq n + 1$. We assume Σ is an integer alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, and that $\sigma \leq m$. The Aho-Corasick (AC) data structure [1] solves the dictionary matching problem using $O(m \log m)$ bits of space and in $O(|T| + occ)$ time, where occ is the size of the output.

Online dictionary matching. In the *online dictionary matching problem* the input is the same as in the dictionary matching problem, but here the text T arrives online (character by character) and the goal is to report all of the occurrences of patterns from D as soon as they appear (before the next character arrives). For a dictionary D and text T let S_i be the longest suffix of $t_1 t_2 \ldots t_i$ such that $S_i \in P(D)$. The AC data structure works in the online model by repeatedly finding S_{i+1} from S_i and t_{i+1} (and then also reporting all of the patterns from D that are suffixes of S_{i+1}). The amortized cost for this process, ignoring the work for reporting the output, is constant. However, the worst-case time per character in the AC data structure can be as large at $\Theta(n_{max})$. This may be too large for real-time applications, such as those that occur in NIDS.

One naïve way of tackling this problem is by using an automata with a state for each prefix in $P(D)$, where each state has σ outgoing transitions. However, this approach introduces a blow up in space, which in practice means that the entire data structure cannot fit in fast memory. Moreover, even the $O(m \log m)$ bit implementation of the AC data structure may be too large. Thus, a large body of recent work has focused on succinct representations of the AC data structure.

Succinct data structures. Given a combinatorial object a representation of the object is *succinct* if it uses $z + o(z)$ bits of space where z is the *information theoretic lower bound* for the number of bits representing the object. The main challenge when using a succinct representation is supporting the algorithmic operations with costs that are as efficient as in the non-succinct representation.

A growing trend in recent years has focused on developing succinct representations for the dictionary matching problem; see Table 1. The information theoretic lower bound for a dictionary of size n over alphabet σ is $n \log \sigma$ bits which is significantly less than the $O(m \log m)$ bits used by the AC data structure, when $\sigma << n$. However, much like in the AC data structure, current succinct representations also pay $\Theta(n_{max})$ time per character in the worst-case. We emphasize that Hon et al. [21] presented a solution using $O(m \log \sigma)$ bits (which is not succinct) and the worst-case cost per character is $O(\log \log m)$ time.

1.1 Our Results

In this paper we introduce a new succinct representation of the AC data structure with an implementation that supports low time cost per character in the worst-case. Such a solution addresses the type of constraints that show up in practical settings, such as in NIDS, where the space usage is limited and the worst-case time per character needs to remain low. Our succinct representation is summarized as follows.

▶ **Theorem 1.** *For $\sigma = \omega(1)$ there exists a succinct data structure for the online dictionary matching problem using $m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$ bits of space where the worst-*

Table 1 Comparison of the results.

Algorithm	Space	Worst-case Time per Character	Total Time
AC (NFA) [1]	$O(m \log m)$	$O(n_{max})$	$O(\|T\| + occ)$
AC (DFA) [1]	$O(m\sigma \log (m\sigma))$	$O(1)$	$O(\|T\| + occ)$
Chan et al. [12]	$O(m\sigma)$	$O(\log^2 m)$	$O((\|T\| + occ) \log^2 m)$
Hon et al. [21]	$O(m \log \sigma)$	$O(\log \log m)$	$O(\|T\| \log \log m + occ)$
Belazzougui [7]	$m(H_0(D) + 3.443 + o(1)) + O(d \log \frac{n}{d})$	$O(n_{max})$	$O(\|T\| + occ)$
Hon et al. [22]	$m(H_k(D) + 5 + o(1)) + O(d \log \frac{n}{d})$	$O(n_{max})$	$O(\|T\| + occ)$
New ($\sigma = \omega(1)$)	$m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$	$O(\log \log \sigma)$	$O(\|T\| + occ)$

case time per character is $O(\log \log \sigma)$, and the total time for a text query T is $O(|T| + occ)$ where occ is the size of the output.

Our main technique is a succinct representation of multi-labeled trees of size n, where each node in the tree has a set of labels drawn from a set $\mathcal{L}$ where $\lambda = |\mathcal{L}|$. The operations of interest on multi-labeled trees are label dependent. In particular we will be interested in lowest labeled ancestor (LLA) queries where given a node u and a label ℓ we need to report the lowest proper ancestor of u that has label ℓ. We show in Sections 4 and 5 how to support such operations for general trees. Strikingly, the type of trees in our implementation of the AC data structure exhibit some special combinatorial properties. Their properties allow an even more succinct representation for these trees which efficiently support LLA queries and other label dependent operations.

In this paper we propose a representation of multi-labeled trees that is succinct when $\lambda = \omega(1)$. Although we mainly consider the *LLA* operation, our representation supports many other operations as well and is succinct for more cases. Moreover, we find our implementation of the *LLA* operation to be simpler than previous approaches (see below).

1.2 Related Work

The notion of succinct data structures was introduced by Jacobson [24] with succinct data structures for bit-arrays, trees and graphs. Many succinct representations for combinatorial objects have since been developed, including succinct representations of sets [24, 26, 30], strings [28, 8], and trees [27, 17].

The first solution for the dictionary matching problem using less than $O(m \log m)$ bits was introduced by Chan et al. in [12]. Their solution also solves the dynamic variant of the problem. Other solutions are based on using suffix trees [23, 21] and are slower than the AC algorithm.

The first representation for the dictionary matching problem in succinct space without a query slowdown was introduced by Belazzougui [7] which was slightly improved by Hon et al. [22]. Succinct representations have also been developed for some variations of the dictionary matching problem, such as dynamic dictionary matching [21, 15], 2D dictionary matching [29], and approximate dictionary matching [21].

Labeled and multi-labeled trees. The problem of representing labeled trees was first considered by Geary, Raman and Raman [17]. However, their solution is succinct only for $\lambda = o(\frac{\log \log n}{\log \log \log n})$. Ferragina et al. [16] proposed a representation of labeled trees based on the XML Burrows–Wheeler transform. However, their representation does not support *LLA* queries. Barbay et al. [5, 6] introduced a representation for labeled trees and multi-labeled trees supporting a restricted set of operations which does not include *LLA* queries. Moreover, their representation is succinct only when $\frac{t}{n} = \lambda^{o(1)}$.

The only known representation of labeled trees which supports *LLA* queries using succinct space are the solutions of He et al. [20] and Tsur [31]. Although these solutions are for the labeled case, they can be extended for multi-labeled trees using the same techniques of Barbay et al. [5], but then they would only be succinct when $\frac{t}{n} = \lambda^{o(1)}$.

2 Preliminaries

2.1 The Aho-Corasick data structure

The Aho-Corasick (AC) data structure [1] is a multi-pattern extension of the KMP data structure [25]. Since the AC data structure is in the core of this paper, we present its internals in some more detail.

The AC data structure is built upon a *trie* storing the patterns in D. The trie edges have the properties that each edge is labeled by a character $\sigma \in \Sigma$, and any two edges leaving the same node have different labels. Thus, there is a bijection between nodes in the trie and prefixes in $P(D)$. For a prefix $u \in P(D)$ let $state(u)$ be the node in the trie corresponding to u. Then u is the concatenation of the edge labels on the path from the root of the trie to $state(u)$. When it is clear from context, we sometimes abuse notation and refer to $state(u)$ as u itself.

The edges of the trie are termed as *forward links*. In addition to the forward links, there are also *failure links* and *report links*. For $u, v \in P(D)$ there is a failure link from node u to node v if and only if v is the longest string in $P(D)$ that is a proper suffix of u. Similarly, for $u \in P(D)$ and $v \in D$ there is a report link from node u to node v if and only if v is the longest string in D that is a proper suffix of u.

In order to solve the online prefix matching problem, we will move from a node u in the AC structure that corresponds to S_i to the node v that corresponds to S_{i+1}. To do this, the AC algorithm first tries to use a forward link from u with the character t_{i+1}. If no such forward link exists, then the algorithm recursively follows failure links until either no failure links are found (in which case v is the root of the trie) or until we reach a node that has a forward link with the character t_{i+1}. One can show that the cost per character of this process is $O(1)$ amortized time. Once v is found we use report links to report the current occurrences.

2.2 Succinct Representation of Trees

Representing ordinal trees. An *ordinal* tree $\mathcal{T}$ with n nodes is a rooted tree where the children of each node are ordered. Each node is given a unique id from $1, \ldots, n$. We use succinct representations of ordinal trees, where each node is given a unique id (the actual tree is not stored). The id is the rank of the node in the pre-order traversal of $\mathcal{T}$.

We use the Balanced Parentheses (BP) representation introduced by Jacobson [24]. In this representation we use parentheses to represent a pre-order traversal of the tree where the first time we visit a node is represented with an open parentheses and the last time we visit a node is represented with a close parentheses. This creates an array of $2n$ bits. For a node u let $open(u)$ and $close(u)$ denote the open and close parentheses of u.

Base set of operations. Munro and Raman [27] showed how to support the following operations in constant time using another $o(n)$ bits on top of the BP representation, for a total of $2n + o(n)$ bits. By supporting this *base set* of operations on the BP representation one can also support many other common operations in constant time.

- $findclose(l)$ – Given an index $l = open(u)$ for some node u, return $close(u)$.
- $findopen(r)$ – Given an index $r = close(u)$ for some node u, return $open(u)$.
- $enclose(i)$ – Return the pair of indices (l, r) such that: (1) l and r correspond to the same node, (2) $l \leq i \leq r$, and (3) $r - l$ is minimized.
- $pre\_rank(i)/post\_rank(i)$ – Return the number of open/close parentheses in the the first i parentheses.
- $pre\_select(i)/post\_select(i)$ – Return the index of the i'th open/close parenthesis.

It is important to notice that given an interval $[l, r]$ that corresponds to a node v, the id of v is $pre\_rank(l)$. Similarly, given the id i of v we have $l = pre\_select(i)$ and $r = findclose(l)$. To simplify these operations we use the notion $v = node([l, r])$ and $[l, r] = interval(v)$. Our algorithms will also make use of the following two properties of the BP representation.

▶ Property 2.1. Let $\mathcal{T}$ be an ordinal tree. Let u be a node in $\mathcal{T}$ whose rank in the pre-order (post-order) on $\mathcal{T}$ is i (j). Then the open (close) parenthesis in the BP representation of $\mathcal{T}$ is i (j).

▶ Property 2.2. Let $\mathcal{T}$ be an ordinal tree and let $[a, b]$ and $[c, d]$ be two subintervals in the BP representation of $\mathcal{T}$ that correspond to two different nodes. Then either one subinterval is completely contained in the other or both subintervals are disjoint.

We emphasize that the $2n + o(n)$ bit representation of Geary, Raman and Raman [17] subsumes the representation of Munro and Raman [27], and in particular supports the base set of operations on the BP representation in constant time.

2.3 Labeled Trees and Multi-Labeled Trees

A *labeled tree* is an ordinal tree where each node has a label drawn from a set $\mathcal{L}$ of size $\lambda = |\mathcal{L}|$. A *multi-labeled tree* is an ordinal tree where each node is associated with a (possibly empty) subset of $\mathcal{L}$. For multi-labeled trees we denote the sum of the sizes of the label subsets by t. We assume without loss of generality that $\lambda \leq t$. Notice that the information-theoretic lower bound for representing a multi-labeled tree is $\log \binom{n\lambda}{t} + \log \binom{2n}{n} + o(n)$.

Our algorithms will make use of lowest labeled ancestor (LLA) queries on multi-labeled trees, where given a node id u and a label ℓ we can quickly return a node id v that is the lowest *proper* ancestor of u which has the label ℓ, or report that no such node exists. This operation is denoted by $v = LLA(\ell, u)$. For succinctness sake, from now on we refer to a node id as the node itself.

Representations supporting same label operations. In order to support fast LLA queries in succinct space we will make use of succinct representations of trees that allow us to compute in constant time some specific operations. These operations are on a label ℓ and a node u where u is also labeled by ℓ. The same label operations that we require are LLA, $pre\_rank_{\mathcal{T}}$ and $post\_rank_{\mathcal{T}}$ queries. We will also want to support $pre\_select_{\mathcal{T}}$ and $post\_select_{\mathcal{T}}$ queries in constant time. For sake of simplicity we refer to all of these operations as same label operations (although the select operations do not have any node as input). See Table 2 for the list of these operations.

In Section 4 we prove the following theorem.

▶ **Theorem 2.** *Assume there is a representation for a multi-labeled tree $\mathcal{T}$ using $f(\mathcal{T})$ bits that supports the same-label operations and the base set operations on the BP representation*

Table 2 Same label operations for multi-labeled trees.

Operation	Description
$LLA(\ell, u)$	The closest proper ancestor of u labeled by ℓ
$pre\_rank_{\mathcal{T}}(\ell, u)$	The rank of u (by the preorder of $\mathcal{T}$) in the set of nodes labeled by ℓ
$post\_rank_{\mathcal{T}}(\ell, u)$	The rank of u (by the postorder of $\mathcal{T}$) in the set of nodes labeled by ℓ
$pre\_select_{\mathcal{T}}(\ell, i)$	The i'th node with label ℓ in the preorder of $\mathcal{T}$
$post\_select_{\mathcal{T}}(\ell, i)$	The i'th node with label ℓ in the postorder of $\mathcal{T}$

in $O(1)$ time each. Then there exists a representation of $\mathcal{T}$ that for any $\lambda = \omega(1)$ uses $f(\mathcal{T}) + o(n+t)$ bits and answers any LLA query in $O(\log\log\lambda)$ time.

We are also able to represent any tree so it can support same-label LLA queries, as long as the label universe is an integer universe $\mathcal{L} = \{1, 2, \ldots, \lambda\}$. This is discussed in Section 5, where combined with Theorem 2 we prove the following theorem.

▶ **Theorem 3.** *For any multi-labeled tree $\mathcal{T}$ with a label set $\mathcal{L} = \{1, 2, \ldots, \lambda\}$ with $\lambda = \omega(1)$, there exists a representation of $\mathcal{T}$ that uses $\lceil \log \binom{n\lambda}{t} \rceil + 2(n + t + \lambda) + o(n + t + \lambda)$ bits and supports LLA queries in $O(\log\log\lambda)$ time.*

3 Dictionary Matching and Same Label Operations

The *c-extended prefix subset* of D, denoted by $P_c(D)$, is the subset of $P(D)$ which contains all $u \in P(D)$ such that $uc \in P(D)$ (the concatenation of u and c).

For each $u \in P(D)$ let u^R be the the string u in reverse order, and let $P(D)^R$ be the set of all reversed prefixes of D. The *suffix-lexicographic order* of $P(D)$ is an ordering of the elements in $P(D)$ where the order is determined by the lexicographic order of the corresponding elements in $P(D)^R$. Thus, for $u \in P(D)$, the rank of u in the suffix-lexicographic order of $P(D)$, denoted by $rank(u)$, is the lexicographic rank of u^R in $P(D)^R$. Since each prefix in $u \in P(D)$ has a unique node $state(u)$ in the AC data structure, let $rank(u)$ be the unique id of $state(u)$. Unless specified otherwise we will abuse notation and assume that $state(u) = rank(u)$.

Belazzougui's data structure. Belazzougui in [7] showed how one can leverage the suffix-lexicographic order of $P(D)$ in order to implement the AC data structure with $n(H_0(D) + 3.443 + o(1)) + O(d \log \frac{n}{d})$ bits. Our solution replaces only one particular component of Belazzougui's data structure which is called the *failure tree*, denoted by $\mathcal{T}_{fail}$. This tree is defined by the failure links in the AC data structure, so that for two nodes $state(u)$ and $state(v)$ we have $fail(state(u)) = state(v)$ if and only if $parent_{\mathcal{T}_{fail}}(state(u)) = state(v)$. An important property of $\mathcal{T}_{fail}$ is that the pre-order traversal of $\mathcal{T}_{fail}$ is exactly the suffix-lexicographic order of $P(D)$. Thus, Belazzougui's data structure uses succinct representations of ordinal trees for representing $\mathcal{T}_{fail}$ that support *parent* operations in constant time, thereby simulating the failure links.

3.1 Final-Failure Links

As discussed above, given some $S_i \in P(D)$ and $c \in \Sigma$ such that $S_ic \notin P(D)$ the time for finding S_{i+1} in the AC algorithm is $\Theta(n_{max})$. This expensive runtime occurs since the AC algorithm may traverse many failure links. However, the traversal stops when the algorithm reaches a node for which there exists a forward link labeled by c. If such a node exists then this node is the final node in the traversal. We call this node the *final-failure* node for S_i and c, denoted by $ff(c, S_i)$. Notice that $ff(c, u) = \mathit{state}(v)$ where v is the longest suffix of u for which $v \in P_c(D)$. If no such node exists then we say that $ff(c, u) = \perp$. The key idea for improving the time cost per character of the AC algorithm is to find the final-failure node directly instead of traversing all of the failure links. We emphasize that the rest of Belazzougui's data structure remains the same. The only thing we change is the component for finding the final-failure.

In order to support locating the final-failure node we extend the definition of the failure tree. Instead of representing $\mathcal{T}_{fail}$ as an unlabeled ordinal tree, we represent $\mathcal{T}_{fail}$ as a multi-labeled tree. For each node $\mathit{state}(u) \in \mathcal{T}_{fail}$ we say that $\mathit{state}(u)$ is labeled by c if and only if $u \in P_c(D)$. Notice that a node may have many labels, or no labels at all (which is why we use a multi-labeled tree). Now the process of finding the final-failure node for $\mathit{state}(u)$ and character c reduces to finding $LLA(c, \mathit{state}(u))$ in the multi-labeled version of $\mathcal{T}_{fail}$.

Same label operations on $\mathcal{T}_{fail}$. We will now show how the properties of the AC structure and the implementations we consider allow us to support the same label operations in Table 2 on $\mathcal{T}_{fail}$ in constant time. This will allow us to use Theorem 2.

▶ **Lemma 4.** *There exists an implementation of $\mathcal{T}_{fail}$ that supports the parent operation, same-label operations and the base set operations on the BP representation in $O(1)$ time using $m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$ bits.*

Proof. Our implementation of $\mathcal{T}_{fail}$ contains two components. The first component is an implementation of the forward links of the AC data structure which is another part of the data structure of Belazzougui [7]. For $u \in P_c(D)$, the forward link from $\mathit{state}(u)$ with character $c \in \Sigma$ is implicitly represented by the ordered pair $(c, \mathit{state}(u))$. Using Belazzougui's implementation we can move from $(c, \mathit{state}(u))$ to $\mathit{state}(uc)$ or backwards in constant time.

The second component is a representation of a slightly modified version of $\mathcal{T}_{fail}$. A key observation with regard to the structure of $\mathcal{T}_{fail}$ is that for any child of the root of $\mathcal{T}_{fail}$, all of the nodes in the subtree of this child correspond to prefixes of the form uc for some $c \in \Sigma$ and $u \in P_c(D)$. However, it is possible that suffixes of the form uc are partitioned among several subtrees of children of the root. For purposes that will be clear later, it is helpful to have all of the nodes corresponding to prefixes ending with character c in one unique subtree of a child of the root. To support this, we add σ new dummy nodes, one for each character in Σ. These nodes will be the only children of the root. The i'th dummy has in its subtree all of the nodes of the form vi for each $P_i(D)$ (recall that $\Sigma = \{1, 2, \ldots, \sigma\}$). This is guaranteed by having each old child of the root become a child of the appropriate new dummy node. Notice that the pre-order and post-order of the nodes in $\mathcal{T}_{fail}$, excluding the dummy nodes, do not change with this modification. Rather, the i'th dummy node is inserted between the nodes corresponding to prefixes ending with $i-1$ and the nodes corresponding to prefixes ending with i in the pre-order. Thus, for ui we have $\mathit{pre\_rank}_{\mathcal{T}_{fail}}(ui) = \mathit{pre\_rank}_{\mathcal{T}'_{fail}}(ui) - i$. Similarly, the i'th dummy node is inserted between the nodes corresponding to prefixes ending with i and the nodes corresponding to prefixes ending with $i+1$ in the post-order. For the rest of this proof we refer to this slightly modified tree as $\mathcal{T}'_{fail}$. Notice that $\mathcal{T}'_{fail}$ has $m + \sigma$ nodes.

We represent $\mathcal{T}'_{fail}$ with the data structure of Geary, Raman and Raman [17] using $2m + 2\sigma + o(m)$ bits. Recall that this implementation supports the base set operations on the BP representation in constant time. The particular constant time operations we use with this representation on $\mathcal{T}'_{fail}$ are:

- $parent_{\mathcal{T}'_{fail}}(u) = parent(u)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ return the id of the parent of u in $\mathcal{T}'_{fail}$.
- $child_{\mathcal{T}'_{fail}}(u, i) = child(u, i)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ and a positive integer i, return the id of the i'th child of u in $\mathcal{T}'_{fail}$.
- $pre\_rank_{\mathcal{T}'_{fail}}(u) = pre\_rank(u)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ return its location in the pre-order traversal of $\mathcal{T}'_{fail}$.
- $post\_rank_{\mathcal{T}'_{fail}}(u) = post\_rank(u)$: Given the id of a node $u \in \mathcal{T}'_{fail}$ return its location in the post-order traversal of $\mathcal{T}'_{fail}$.
- $pre\_select_{\mathcal{T}'_{fail}}(i) = pre\_select(i)$: Given an integer $1 \le i \le m + \sigma$ return the id of the i'th node in the pre-order traversal of $\mathcal{T}'_{fail}$.
- $post\_select_{\mathcal{T}'_{fail}}(i) = post\_select(i)$: Given an integer $1 \le i \le m + \sigma$ return the id of the i'th node in the post-order traversal of $\mathcal{T}'_{fail}$.

We use the parent operations on $\mathcal{T}'_{fail}$ to simulate parent operations on $\mathcal{T}_{fail}$ as follows. Due to the dummy nodes, when invoking the parent operation on u we check if the parent of u is a child of the root (by invoking another call to the parent operation), and if so we treat the root as the parent of u. Otherwise, the parent of u in $\mathcal{T}'_{fail}$ is also the parent of u in $\mathcal{T}_{fail}$.

Same label LLA. For $u, v \in P_c(D)$ we have that $LLA(c, state(u)) = state(v)$ if and only if $parent_{\mathcal{T}_{fail}}(state(uc)) = state(vc)$. This gives lead to supporting same label LLA queries in constant time. To do this, we first move from $state(u)$ to $state(uc)$ in constant time with the forward links structure, then we move from $state(uc)$ to $parent_{\mathcal{T}_{fail}}(state(uc)) = state(vc)$ in constant time using parent operations on $\mathcal{T}'_{fail}$, and then we move from $state(vc)$ to $state(v)$ using the forward links structure (going backwards) in constant time. The transition from $state(vc)$ to $state(v)$ is executed by first finding the pair $c, state(v)$ via a select operation on $state(v, c)$. This pair is represented using $\log m + \log \sigma$ bits. Extracting the $\log m$ bits representing $state(v)$ completes the transition.

Pre-order and post-order rank/select queries. We focus on the details for implementing $pre\_rank_{\mathcal{T}_{fail}}(c, u)$ for some $u \in P_c(D)$ as the rest of the operations are implemented using similar ideas (and the implementations are mostly technical). Recall that by definition, for $u \in P_c(D)$, $pre\_rank_{\mathcal{T}_{fail}}(c, u)$ is exactly the rank of uc in the pre-order $\mathcal{T}_{fail}$, minus $\sum_{c'<c} |P_{c'}(D)|$. Recall that $pre\_rank_{\mathcal{T}_{fail}}(uc) = pre\_rank_{\mathcal{T}'_{fail}}(uc) - c$, so the rank of u in the pre-order of $\mathcal{T}_{fail}$ among the nodes labeled by c can be computed in constant time by invoking $pre\_rank_{\mathcal{T}'_{fail}}(uc)$. Next, let $b = pre\_rank(child(r, c))$ where r is the root of $\mathcal{T}'_{fail}$. Since the c'th child of r is the dummy corresponding to c, then its rank in the pre-order of $\mathcal{T}'_{fail}$ is exactly $\sum_{c'<c} |P_{c'}(D)| + (c - 1)$. So we can compute $pre\_rank_{\mathcal{T}_{fail}}(c, u) = pre\_rank_{\mathcal{T}'_{fail}}(uc) - b$ in constant time.

Space usage. Our data structure uses the same space as Belazzougui's data structure, with the exception that instead of using $2m + o(m)$ bits for representing the failure tree, we use $2m + 2\sigma + o(m)$ bits via the representation of Geary, Raman and Raman [17] (which also supports base set of operations on the BP representation). Thus the total space used is $m(\log \sigma + \frac{2\sigma}{m} + 3.443 + o(1)) + O(d \log \frac{n}{d})$ bits. We further reduce the space usage using the technique of Hon et al. [22] to compress the forward links component into its *k'th order entropy*, thereby achieving a representation that uses $m(H_k(D) + 5 + o(1)) + 2\sigma + O(d \log \frac{n}{d})$ bits. ◀

3.2 Proof of Theorem 1

By combining Lemma 4 and Theorem 2 we obtain a succinct representation of $\mathcal{T}_{fail}$ which supports finding failure links in worst-case constant time and finding a final-failure in worst-case $O(\log\log\sigma)$ time, while using $m(H_k(D)+5+o(1))+2\sigma+O(d\log\frac{n}{d})$ bits.

For the text processing, each time a new character arrives we traverse at most $\log\log\sigma$ failure links. By Lemma 4, each such traversal takes constant time via a parent operation on $\mathcal{T}_{fail}$. If one of these links leads to the final failure, then we are done. Otherwise, we invoke the final failure procedure, which costs another $O(\log\log\sigma)$ time. Thus, the runtime is never worse than the runtime of the AC algorithm, and so the worst-case cost per character is $O(\log\log\sigma)$ (ignoring the cost of reporting the output) and the total cost for the entire text is $O(|T|+occ)$.

4 Solving General LLA With Same Label Operations

In this section we prove Theorem 2.

Successor Search. Recall that by the assumption of Theorem 2, the base set of operations on the BP representation of $\mathcal{T}$ are supported in constant time. For each label ℓ let $I_{\ell,open}$ and $I_{\ell,close}$ be the set of indices in the BP representation of the open and close parentheses, respectively, that correspond to nodes with label ℓ.

Let M be a subset of an ordered universe U. For an element $x\in U$ the successor of x in M is $succ_M(x)=\text{argmin}_{y\in M}\{y>x\}$. For sake of completeness we say that if $x>\max_{y\in M}\{y\}$ then $succ_M(x)=\infty$. In the following we show how successor operations on the sets $I_{\ell,open}$ and $I_{\ell,close}$ are used for answering LLA queries.

▶ **Lemma 5.** *Let $\mathcal{T}$ be a multi-labeled tree over label set $\mathcal{L}$. For a node $u\in\mathcal{T}$ and a label $\ell\in\mathcal{L}$ let $l=succ_{I_{\ell,open}}(close(u))$ and $r=succ_{I_{\ell,close}}(close(u))$. If $r<l$ then $LLA(\ell,u)=v$ where $v=node([findopen(r),r])$. If $r>l$ then $LLA(\ell,u)=LLA(\ell,w)$ where $w=node([l,findclose(l)])$. If $r=l$ then there is no node $LLA(\ell,u)$.*

Proof. Our proof has three cases. In the first case $r<l$, and so by Property 2.2 it must be that $open(u)>findopen(r)$. Therefore, the interval $[findopen(r),r]$ contains the interval $[open(u),close(u)]$ implying that v is an ancestor of u. Since v is labeled with ℓ and $r=close(v)=succ_{I_{\ell,close}}(close(u))$ there is no node on the internal path from v to u in $\mathcal{T}$ that is labeled with ℓ. Thus, $v=LLA(\ell,u)$.

In the second case $l<r$. We first show that $LLA(\ell,u)$ is necessarily an ancestor of $LLA(\ell,[l,findclose(l)])$ and then show that $LLA(\ell,[l,findclose(l)])$ is necessarily an ancestor of $LLA(\ell,u)$. Thus, the two must be the same.

Recall that the interval defined by $enclose(LLA(\ell,u))$ contains the interval $[open(u), close(u)]$. Moreover, since $l<r$ there is no closing parentheses of a node with label ℓ at the indices strictly between $close(u)$ and l. Therefore, the interval corresponding to $LLA(\ell,u)$ must contain the index l. Combining this with Property 2.2 it must be that the interval corresponding to $LLA(\ell,u)$ contains the interval $[open(u),findclose(l)]$ and so $LLA(\ell,u)$ is necessarily an ancestor of $LLA(\ell,[l,findclose(l)])$. For the other direction, by Property 2.2 the interval corresponding to $LLA(\ell,[l,findclose(l)])$ must contain the interval $[l,findclose(l)]$. Since there is no index in $I_{\ell,open}$ between $close(u)$ and l, the interval corresponding to $LLA(\ell,[l,findclose(l)])$ must contain the index $close(u)$. Combining with Property 2.2 the interval corresponding to $LLA(\ell,[l,findclose(l)])$ must contain the interval $[open(u),findclose(l)]$, and so $LLA(\ell,[l,findclose(l)])$ must be an ancestor of $LLA(\ell,u)$.

In the third case $r = l$. Then it must be that $r = l = \infty$ since otherwise we have a single index for both an open and close parentheses. Thus, there is no index of close parentheses in the range $[close(u) + 1, 2n]$ that corresponds to a node labeled with ℓ. If u has a proper ancestor v that is labeled with ℓ, then by Property 2.2 $close(v) > close(u)$. Therefore, there is no such ancestor, and $LLA(\ell, u)$ does not exist. ◀

By Lemma 5, once we perform two successor operations and a constant number of base set operations, we either find a node $v = LLA(\ell, u)$ or we find a node w that is labeled with ℓ such that $LLA(\ell, u) = LLA(\ell, w)$. Computing $LLA(\ell, w)$ in the second case takes $O(1)$ time since w is labeled with ℓ (and so this is a same label LLA query). What remains to be shown is how to execute the two successor queries on the sets of indices.

Successor queries on subsets of indices. Let R be a binary matrix of size $[a] \times [b]$. For integers $1 \le x \le a$ and $1 \le y \le b$ let $rank_{col}(y, x)$ be the number of 1s in the first x entries of the y'th column of R. For integers $1 \le j \le a$ and $1 \le y \le b$ let $select_{col}(y, j)$ be the index of the j'th 1 in the y'th column of R.

We focus on $I_{\ell,open}$ as the treatment of $I_{\ell,close}$ is the same. Consider the binary matrix R_{open} of size $[n] \times [\lambda]$, where $R_{open}[i][\ell] = 1$ if and only if $i \in I_{\ell,open}$. Given a node u in $\mathcal{T}$ we can find the row that corresponds to u in R_{open} in constant time by executing a single $pre\_rank(u)$ operation (which is a base set operation). Using the encoding of Barbay et al. [5] on R_{open} we can answer $rank_{col}$ and $select_{col}$ queries in $O(\log\log\lambda)$ and $O(1)$ time respectively. However, this encoding makes use of $O(t \log \lambda)$ bits (since there are t non zero values in the matrix). We reduce this space usage using indirection as follows.

Let $\tau = \log^2 \lambda$. For each set of indices $I_{\ell,open}$ let $\hat{I}_{\ell,open} \subset I_{\ell,open}$ be the indices whose rank in $I_{\ell,open}$ is a multiple of τ. Notice that if $|I_{\ell,open}| < \tau$ then $\hat{I}_{\ell,open} = \emptyset$. The treatment of such cases is discussed after explaining the more challenging case. Consider the binary matrix $\hat{R}_{open}$ of size $[n] \times [\lambda]$, where $\hat{R}_{open}[i][\ell] = 1$ if and only if $i \in \hat{I}_{\ell,open}$. Notice that the number of non-zero entries in $\hat{R}_{open}$ is $t' = O(\frac{t}{\tau})$. We further reduce the matrix $\hat{R}_{open}$ by removing all of the rows that have only zeros, and use another rank and select data structure to move between the row indices of these matrices. This uses another $t' \log \frac{n}{t'} + O(t') + o(n)$ bits [30], which is $o(n + t)$ bits[1].

Thus, we answer rank and select queries on the rows of $\hat{R}_{open}$ using the encoding of Barbay et al. [5] with $O(t' \log \lambda) = o(t)$ bits. Given an index i for which we wish to compute $s = succ_{I_{\ell,open}}(i)$ we first find $\hat{s} = succ_{\hat{I}_{\ell,open}}(i)$ using the data structure on $\hat{R}_{open}$ after finding the appropriate row in the matrix $\hat{R}_{open}$ with a single $rank_{col}$ operation in $O(\log\log\lambda)$ time. If we have successfully found $\hat{s}$ it must be that $|rank_{I_{\ell,open}}(s) - rank_{I_{\ell,open}}(\hat{s})| \le \tau$. Thus, with $O(\log \tau)$ executions of $pre\_select(\ell, i)$ (each costing $O(1)$ time since it is a same label operation), we perform a binary search to find s in $O(\log \tau)$ time.

Finally, if we were not successful in finding $\hat{s}$ (either $\hat{I}_{\ell,open} = \emptyset$ or $i \ge \max \hat{I}_{\ell,open}$) then there are at most τ possible elements to consider (the last τ elements) and a binary or exponential search with $pre\_select(\ell, i)$ operations finds s in $O(\log \tau)$ time. This completes the proof of Theorem 2

[1] If $t \le n$ then let $t = n/x$ for some x. Then $t' \log \frac{n}{t'} = \frac{n}{x\tau} \log x\tau = o(n)$. If $t > n$ then $t' \log \frac{n}{t'} = \frac{t}{\tau} \log \frac{n\tau}{t} \le \frac{t}{\tau} \log \tau = o(t)$.

5 Same Label Operations for General Multi-labeled Trees

In this section we prove Theorem 3. The representation of $\mathcal{T}$ uses two main components. The first component is a *label-ordered tree*, which functions in a way that is similar to the modified failure tree in the dictionary matching data structure. The second component is an implementation of a *transition operator* which functions in a way that is similar to the forward links in the dictionary matching data structure.

Label-ordered tree. The encoding technique for the label-ordered tree is similar to the *tree extraction* technique used in [18, 19, 20]. For each $\ell \in \mathcal{L}$ let $\mathcal{F}_\ell$ be the induced forest obtained by inducing $\mathcal{T}$ on the nodes with label ℓ. By an inducing we mean that for two nodes $u, v \in \mathcal{F}_\ell$, u is the parent of v if and only if both u and v are labeled with ℓ, and $u = LLA(\ell, v)$. Notice that the sum of the sizes of all of the forests is exactly t. For each such forest we create a dummy node and make it the parent of all of the roots of trees in the forest. This adds another λ nodes. Finally, we add a special new root whose children are the dummy nodes, ordered by their labels, thereby creating the label-ordered tree. We denote this tree by $\hat{\mathcal{T}}$. The size of $\hat{\mathcal{T}}$ is $t + \lambda + 1$. We use the data structure of Geary, Raman and Raman [17] to represent $\hat{\mathcal{T}}$ with $2t + 2\lambda + o(t + \lambda)$ bits, while supporting the base set of operations on the BP representation of $\hat{\mathcal{T}}$ in constant time.

Transition operator. The transition operator translates in constant time between the rank of a node u in $\mathcal{T}$ and a label ℓ, and the rank of the copy of u in $\hat{\mathcal{T}}$ which is associated with ℓ. This translation works in both directions. To do so, for each $u \in \mathcal{T}$ and for each label ℓ of u, the transition operator creates the ordered pair $(\ell, pre\_rank_{\mathcal{T}}(u))$. Notice that $rank((\ell, pre\_rank_{\mathcal{T}}(u))) = pre\_rank_{\hat{\mathcal{T}}}(u\ell) - \ell - 1$, where the rank is taken over all ordered pairs. Similarly, one can use a select query to translate from $pre\_rank_{\hat{\mathcal{T}}}(u\ell)$ to $pre\_rank_{\mathcal{T}}(u)$. We use a data structure that supports rank and select queries in constant time [30] using $\lceil \log \binom{n\lambda}{t} \rceil + o(t) + O(\log\log(n\lambda))$ bits.

Same label LLA. Using the above representations, we support same label LLA queries exactly like we do in the proof of Lemma 4. Since we used a representation for supporting the base set of operations on the BP representation of $\hat{\mathcal{T}}$ in constant time, again using the ideas in the proof of Lemma 4 we support same label operations and the base set of operations on the BP representation of $\mathcal{T}$ in constant time. Thus, together with Theorem 2 we have completed the proof of Theorem 3.

References

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

2 Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved dynamic dictionary matching. *Inf. Comput.*, 119(2):258–282, 1995.

3 Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.

4 Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap. *CoRR*, abs/1503.07563, 2015.

5 Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theor. Comput. Sci.*, 387(3):284–297, 2007.

6 Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Trans. on Algorithms*, 7(4):52, 2011.

7 Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, CPM*, pages 88–100, 2010.

8 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014.

9 Anat Bremler-Barr, David Hay, and Yaron Koral. Compactdfa: Generic state machine compression for scalable pattern matching. In *INFOCOM*, pages 659–667. IEEE, 2010.

10 Anat Bremler-Barr, David Hay, and Yaron Koral. Compactdfa: Scalable pattern matching using longest prefix match solutions. *IEEE/ACM Trans. Netw.*, 22(2):415–428, 2014.

11 Gerth Stølting Brodal and Leszek Gasieniec. Approximate dictionary queries. In *Combinatorial Pattern Matching, CPM*, pages 65–74, 1996.

12 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *Symposium on Discrete Algorithms, (SODA)*, pages 13–22, 2005.

13 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *Europ. Symp. Algorithms, (ESA)*, pages 361–372, 2015.

14 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

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

16 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

17 Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

18 Meng He, J. Ian Munro, and Gelin Zhou. Path queries in weighted trees. In *International Symposium on Algorithms and Computation, (ISAAC)*, pages 140–149, 2011.

19 Meng He, J. Ian Munro, and Gelin Zhou. Succinct data structures for path queries. In *European Symposium on Algorithms, (ESA)*, pages 575–586, 2012.

20 Meng He, J. Ian Munro, and Gelin Zhou. A framework for succinct labeled ordinal trees over large alphabets. *Algorithmica*, 70(4):696–717, 2014.

21 Wing-Kai Hon, Tsung-Han Ku, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressing dictionary matching index via sparsification technique. *Algorithmica*, 72(2):515–538, 2015.

22 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. In *String Processing and Information Retrieval, (SPIRE)*, pages 191–200, 2010.

23 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Compressed index for dictionary matching. In *Data Compression Conference (DCC)*, pages 23–32, 2008.

24 Guy Jacobson. Space-efficient static trees and graphs. In *Symposium on Foundations of Computer Science, (FOCS)*, pages 549–554, 1989.

25 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

26 J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

27 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

28 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

29 Shoshana Neuburger and Dina Sokol. Succinct 2d dictionary matching with no slowdown. In *Algorithms and Data Structures Symposium, (WADS)*, pages 619–630, 2011.

30 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

31 Dekel Tsur. Succinct representation of labeled trees. *TCS*, 562:320–329, 2015.

32 Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *INFOCOM*, 2004.