Space-Efficient Dictionaries for Parameterized and Order-Preserving Pattern Matching

Arnab Ganguly*[1], Wing-Kai Hon†[2], Kunihiko Sadakane[3], Rahul Shah[4], Sharma V. Thankachan[5], and Yilin Yang[6]

1 **School of Electrical Engineering and Computer Science, Louisiana State University, USA**
agangu4@lsu.edu
2 **Department of Computer Science, National Tsing Hua University, Taiwan**
wkhon@cs.nthu.edu.tw
3 **Department of Mathematical Informatics, University of Tokyo, Japan**
sada@mist.i.u-tokyo.ac.jp
4 **School of Electrical Engineering and Computer Science, Louisiana State University, USA; and**
National Science Foundation, USA
rahul@csc.lsu.edu, rahul@nsf.gov
5 **School of Computational Science and Engineering, Georgia Institute of Technology, USA**
sharma.thankachan@gatech.edu
2 **Department of Computer Science, National Tsing Hua University, Taiwan**
yilinyang@cs.nthu.edu.tw

Abstract

Let S and S' be two strings, having the same length, over a totally-ordered alphabet. We consider the following two variants of string matching.

- *Parameterized Matching:* The characters of S and S' are partitioned into static characters and parameterized characters. The strings are a parameterized match iff the static characters match exactly, and there exists a one-to-one function which renames the parameterized characters in S to those in S'.
- *Order-Preserving Matching:* The strings are an order-preserving match iff for any two integers $i, j \in [1, |S|]$, $S[i] \prec S[j] \iff S'[i] \prec S'[j]$, where $\prec$ denotes the precedence order of the alphabet.

Let $\mathcal{P}$ be a collection of d patterns $\{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_d\}$ of total length n characters, which are chosen from a totally-ordered alphabet Σ. Given a text T, also over Σ, we consider the dictionary indexing problem under the above definitions of string matching. Specifically, the task is to index $\mathcal{P}$, such that we can report all positions j (called *occurrences*) where at least one of the patterns $\mathsf{P}_i \in \mathcal{P}$ is a parameterized match (resp. an order-preserving match) with the same-length substring of T starting at j. Previous best-known indexes occupy $O(n \log n)$ bits, and can report all *occ* occurrences in $\mathcal{O}(|T| \log |\Sigma| + occ)$ time. We present space-efficient indexes that occupy $\mathcal{O}(n \log |\Sigma| + d \log n)$ bits, and reports all *occ* occurrences in $\mathcal{O}(|T|(\log |\Sigma| + \log_{|\Sigma|} n) + occ)$ time for parameterized matching, and in $\mathcal{O}(|T| \log n + occ)$ time for order-preserving matching.

1998 ACM Subject Classification F.2.2 Pattern Matching

* The work of Arnab Ganguly was supported by National Science Foundation Grants CCF–1017623 and CCF–1218904.
† The work of Wing-Kai Hon was supported by National Science Council Grants 102-2221-E-007-068-MY3 and 105-2918-I-007-006.

27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016).
Editors: Roberto Grossi and Moshe Lewenstein; Article No. 2; pp. 2:1–2:12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Parameterized Matching, Order-preserving Matching, Dictionary Indexing, Aho-Corasick Automaton, Sparsification

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.2

1 Introduction

Designing succinct data-structures for the classical pattern matching problem of finding all occurrences of a pattern P in a fixed text T can be traced back to the seminal work of Grossi and Vitter [14], Ferragina and Manzini [8], and Sadakane [26]. This established an active research area of designing succinct data structures. (See [25] for a comprehensive survey.) The focus was now on either improving these initial breakthroughs [5, 9, 10, 11, 13, 22, 23, 27], or on designing succinct data structures for other variants [4, 6, 12, 17, 24, 30]. Dictionary matching, a typical example of these variants, is a classical problem in string matching and is defined as follows. Let $\mathcal{P}$ be a collection of d patterns $\{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_d\}$ of total length n characters which are chosen from a totally-ordered alphabet Σ of size σ. Given a text T, also over Σ, the task is to report all positions j (called *occurrences*) such that at least one of the patterns $\mathsf{P}_i \in \mathcal{P}$ exactly matches an equal-length substring of T that starts at j. The classical solution for this problem is the Aho-Corasick (AC) automaton [1] which occupies $\Theta(m \log m)$ bits of space, where $m \leq n + 1$ is the number of states in the automaton, and finds all *occ* occurrences in time $\mathcal{O}(|T| \log \sigma + occ)$. The query complexity can be improved to optimal $\mathcal{O}(|T| + occ)$ using perfect-hashing techniques. To the best of our knowledge, the first two succinct indexes for the problem are by Hon et al. [16] and Tam et al. [29]. Later, Belazzougui [4] presented an $m \log \sigma + \mathcal{O}(m) + \mathcal{O}(d \log(n/d))$ bit index with optimal $\mathcal{O}(|T| + occ)$ query time.

The first problem that we consider is popularly known as the *Parameterized Pattern Matching* problem. The problem has significant attention [2, 15, 18, 19, 21] since its inception by Baker in 1993 [3]. The alphabet Σ is partitioned into two disjoint sets: Σ_s containing static-characters (s-characters) and Σ_p containing parameterized characters (p-characters). Two strings S and S', both over Σ, are a parameterized match (p-match) iff $|S| = |S'|$ and there is a one-to-one function f such that $S[i] = f(S'[i])$. For any s-character $c \in \Sigma_s$, we have $f(c) = c$. Thus, for $\Sigma_s = \{A, B, C\}$ and $\Sigma_p = \{w, x, y, z\}$, the strings $AxBxCy$ and $AzBzCx$ are a p-match, but $AxBxCy$ and $AzBwCx$ are not. We consider the *Parameterized Dictionary Matching* problem which was introduced by Idury and Schäffer [18]. This is similar to the standard dictionary problem, just that Σ is partitioned into Σ_s and Σ_p, and we consider the p-matches of a pattern with the text. Idury and Schäffer presented an AC-automaton like solution which occupies $\mathcal{O}(m \log m) = \mathcal{O}(n \log n)$ bits and reports all *occ* occurrences in $\mathcal{O}(|T| \log \sigma + occ)$ time. The following theorem summarizes our contribution.

▶ **Theorem 1.** *By maintaining an index of $\mathcal{P}$ in $\mathcal{O}(n \log \sigma + d \log n)$ bits, all occ occurrences where a pattern in $\mathcal{P}$ and T are a p-match can be reported in $\mathcal{O}(|T|(\log \sigma + \log_\sigma n) + occ)$ time.*

The second problem we consider is a variant of the recently introduced *Order-Preserving Pattern Matching* problem [7, 20]. Two strings S and S' are an order-preserving match (o-match) iff $|S| = |S'|$ and for any two integers $i, j \in [1, |S|]$, we have $S[i] \prec S[j] \iff S'[i] \prec S'[j]$. Thus, for the alphabet $\{A, B, C, D\}$ with the total-order $A \prec B \prec C \prec D$, the string ABC is an o-match with BCD, but not with CDB. Likewise, AAB matches CCD, but does not match ABC. We consider the *Order-Preserving Dictionary Matching* problem introduced by Kim et al. [20]. As with the p-dictionary matching problem, the

match in this case is defined according to order-preserving matching. Kim et al. presented an AC-automaton like approach which occupies $\mathcal{O}(n \log n)$ bits and reports all occurrences in $\mathcal{O}(|T| \log \sigma + occ)$ time. The following theorem summarizes our contribution.

▶ **Theorem 2.** *By maintaining an index of $\mathcal{P}$ in $\mathcal{O}(n \log \sigma + d \log n)$ bits, all occ occurrences where a pattern in $\mathcal{P}$ and T are an o-match can be reported in $\mathcal{O}(|T| \log n + occ)$ time.*

1.1 Map

Our techniques are largely based on the sparsification technique of Hon et al. [16] for the classical dictionary matching problem. For a parameter Δ, this technique condenses every Δ characters of each pattern separately and then creates an AC-automaton for the condensed patterns. Likewise, the text is also condensed starting at a position i. Now the condensed text is matched in the AC-automaton, and all occurrences are reported. The occurrences reported in this run lie in the set $\{i, i + \Delta, i + 2\Delta, \dots\}$. All occurrences are reported by repeating the process for $i = 1, 2, 3, \dots, \Delta$. By properly choosing Δ, different trade-offs for index sizes and query time can be obtained. Broadly speaking, we use this technique to sparsify the AC-automaton based approaches of Idury and Schäffer [18] for p-dictionary matching and of Kim et al. [20] for o-dictionary matching. However, the sparsification technique does not immediately extend to the case of parameterized matching and order-preserving matching. For example, it is not clear whether a condensed alphabet has to be treated as a p-character or an s-character. Also, how do we define the one-to-one mapping? Similarly, how do we impose the total-order on the condensed alphabet in the case of order-preserving matching? A more serious issue is how to handle truncating of characters at the beginning of a currently matched text, which is essential for the AC-automaton based approaches of Idury and Schäffer and of Kim et al.

In Section 2, we first address the p-dictionary problem, and prove Theorem 1. In Section 3, using similar techniques, we arrive at Theorem 2.

2 Parameterized Dictionary Matching

We assume that the p-characters in $\mathsf{P}_i \in \mathcal{P}$ are from the set $\{0, 1, \dots, |\Sigma_p| - 1\}$. Also, the s-characters are disjoint from the set of integers. (The latter assumption can be easily removed by mapping the s-characters onto the set $\{|\Sigma_p|, |\Sigma_p| + 1, \dots, \sigma - 1\}$ such that the kth smallest s-character has value $|\Sigma_p| + k - 1$.) The patterns can be initially processed in $\mathcal{O}(n \log \sigma)$ time to ensure that these conditions hold.

2.1 Encoding Scheme

[3] introduced the following encoding scheme to enable matching of parameterized strings. Given a string S, obtain a string $\mathsf{prev}(S)$ by replacing the first occurrence of every p-character in S by 0 and any other occurrence by the difference in position from its previous occurrence. Thus, $\mathsf{prev}(A1B2A1C0) = A0B0A4C0$. Baker [3] showed that two strings S and S' are a p-match iff $\mathsf{prev}(S) = \mathsf{prev}(S')$. Although this scheme makes p-matching of strings easier to handle, for our purposes, it suffers from a drawback. Specifically, $\mathsf{prev}(S)$ is a string over an alphabet of size $\Theta(n)$ in the worst case, whereas the original alphabet size σ may be much smaller in comparison.

In order to alleviate this, we introduce the following encoding scheme, which is still simple and does not suffer from this drawback. Given a string S over Σ, let $c_0, c_1, \dots, c_k$ be the order in which every $c_i \in \Sigma_p$ appears in S. We obtain a string $\mathsf{pEncode}(S)$ by replacing every

occurrence of c_i by i in S. Thus, $\mathsf{pEncode}(A1B2A1C0) = A0B1A0C2$. By maintaining an integer-array of length $|\Sigma_p|$, we can compute $\mathsf{pEncode}(S)$ in $\mathcal{O}(|S|)$ time[1]. The following observations are immediate.

▶ **Observation 3.** *Two strings S and S' are a p-match iff $\mathsf{pEncode}(S) = \mathsf{pEncode}(S')$. A string S matches another string S' at a position i iff $\mathsf{pEncode}(S) = \mathsf{pEncode}(S'[i, i+|S|-1])$.*

▶ **Observation 4.** *For a string S, assume that the parameterized characters in $\mathsf{pEncode}(S)$ belong to the set $\{0, 1, 2, \ldots, |\Sigma_p| - 1\}$. Then, $\mathsf{pEncode}(S[i, |S|]) = \mathsf{pEncode}(\mathsf{pEncode}(S)[i, |S|])$.*

2.2 Overview

We design our index by classifying the patterns into *long* and *short* based on a parameter $\Delta = \lceil \log_\sigma n \rceil$. The patterns are encoded and maintained explicitly occupying $n \log \sigma$ bits in total. For short patterns (having length less than Δ), we create a trie and use a rather brute-force approach to find all occurrences. On the other hand, reporting the occurrences of long patterns (having length at least Δ) requires sophisticated (and more involved) indexing and querying techniques. Moving forward, when we refer to an occurrence, we imply both the position in the text where a pattern occurs and also the pattern itself. Also, we report all patterns that occur at a particular position. (The query process can be easily adapted to the case when only the position is to be reported.) Then, the set of occurrences of long patterns and short patterns are mutually disjoint and are handled separately. Specifically, we prove the following lemmas of which Theorem 1 is an immediate consequence.

▶ **Lemma 5.** *Let $\mathcal{P}$ be a dictionary consisting of d patterns, each having length at least $\lceil \log_\sigma n \rceil$. By indexing $\mathcal{P}$ in a data-structure occupying $\mathcal{O}(n \log \sigma + d \log n)$ bits, we can report all occ occurrences of the patterns in $\mathcal{O}(|T|(\log \sigma + \log_\sigma n) + occ)$ time.*

▶ **Lemma 6.** *Let $\mathcal{P}$ be a dictionary consisting of d patterns, each having length less than $\lceil \log_\sigma n \rceil$. By indexing $\mathcal{P}$ in a data-structure occupying $n \log \sigma + \mathcal{O}(d \log n)$ bits, we can report all occ occurrences of the patterns in $\mathcal{O}(|T|(\log \sigma + \log_\sigma n) + occ)$ time.*

We assume that no two patterns P_i and P_j exist such that $\mathsf{pEncode}(\mathsf{P}_i) = \mathsf{pEncode}(\mathsf{P}_j)$. For such patterns, we can keep only one pattern in the dictionary, and it is trivial to handle reporting of all patterns for an occurrence in the claimed space-time bounds. We also assume that the p-characters in T are from $\{0, 1, \ldots, |\Sigma_p| - 1\}$ and the s-characters are either disjoint from the set of integers or belong to the set $\{|\Sigma_p|, |\Sigma_p| + 1, \ldots, \sigma - 1\}$. An initial pre-processing of the text in $\mathcal{O}(|T| \log \sigma)$ time ensures that these conditions hold. The $\mathcal{O}(|T| \log \sigma)$ factor in the query complexity of Lemmas 5 and 6 and Theorem 1 is due to this pre-processing.

2.3 Long Patterns (Proof of Lemma 5)

We consider the patterns which are of length at least Δ, where $\Delta = \lceil \log_\sigma n \rceil$. For a string S and Δ, we use $\mathsf{tail}(S)$ to denote the largest suffix of S whose length is a multiple of Δ and $\mathsf{head}(S)$ is the remaining (possibly empty) prefix of S. We begin by obtaining

[1] Initialize a counter $C = 0$ and an integer array A such that $A[c] = -1$ for every $c \in \Sigma_p$. Traverse the string S from left to right. If $S[i] \in \Sigma_p$ (i.e., $S[i] < |\Sigma_p|$) check $A[S[i]]$; otherwise, $\mathsf{pEncode}(S)[i] = S[i]$. If $A[S[i]] = -1$ then assign $\mathsf{pEncode}(S)[i] = A[S[i]] = C$, increment C by one and proceed. Otherwise, assign $\mathsf{pEncode}(S)[i] = A[S[i]]$ and proceed. Note that s-characters remain unchanged.

$\mathsf{pEncode}(\mathsf{tail}(\mathsf{P}_i))$ for every $\mathsf{P}_i \in \mathcal{P}$, and maintain the encoded tails explicitly. Now, we encode $\mathsf{head}(\mathsf{P}_i)$ from right to left using the same encoding that was used for its tail. More specifically, form the string P'_i by concatenating $\mathsf{tail}(\mathsf{P}_i)$ with the reverse of $\mathsf{head}(\mathsf{P}_i)$. Then, the desired encoding of the jth character in the reversed head is given by $\mathsf{pEncode}(\mathsf{P}'_i)[|\mathsf{tail}(\mathsf{P}_i)| + j]$. The following observation is due to the definition of p-match and Observation 3.

▶ **Observation 7.** *Let S and S' be two strings having equal length. Then S and S' are a p-match iff both the conditions are satisfied: (i) the p-encoded tails of both S and S' are equal, and (ii) the p-encoded heads (as described above) of both S and S' are equal.*

The space needed for maintaining the encoded heads and tails of all patterns combined is $n \log \sigma$ bits.

2.3.1 Creating the Index

We create a tree $\mathcal{T}_{out}$ with d nodes where node v_i corresponds to the pattern P_i. A node v_j is the parent of a node v_i iff P_j is the longest pattern such that it is a p-match with a proper-suffix of P_i. In other words, v_j is the parent of a node v_i iff P_j is the longest pattern such that $|\mathsf{P}_j| < |\mathsf{P}_i|$ and $\mathsf{pEncode}(\mathsf{P}_j) = \mathsf{pEncode}(\mathsf{P}_i[|\mathsf{P}_i| - |\mathsf{P}_j| + 1, |\mathsf{P}_i|])$. This *output tree* will be useful for reporting occurrences of a pattern and is analogous to the report links in the AC-automaton [1]. Specifically, let k be a position in the text T such that P_i is the longest pattern which has an occurrence ending at k. Then all patterns whose occurrence ends at k can be found out by following the parent pointers starting at node v_i. Clearly, the start position of all such occurrences can be easily found. Space occupied by the tree is $\mathcal{O}(d \log n)$ bits.

Let Σ' be an alphabet such that each character in Σ' corresponds to a Δ-length string over the alphabet Σ. Thus, Σ' contains at most σ^{Δ} characters, and each character can be represented in $\Delta \log \sigma$ bits. Starting from left, we group every Δ characters of $\mathsf{pEncode}(\mathsf{tail}(\mathsf{P}_i))$, and replace it by the corresponding character from Σ'. In order to efficiently map this Δ-length string over Σ to its corresponding character in Σ', we maintain a perfect hash-table $\mathcal{H}$. Note that the number of Δ-length strings to be stored is at most $\lceil n/\Delta \rceil$. The space occupied by $\mathcal{H}$ is $\mathcal{O}(n/\Delta \times \Delta \log \sigma) = \mathcal{O}(n \log \sigma)$ bits. Create a trie $\mathcal{T}_{tail}$ for all the condensed encoded pattern tails of $\mathcal{P}$. Specifically, if the pattern length is not a multiple of Δ, then we ignore its head while creating the trie. Note that $\mathcal{T}_{tail}$ has at most $\lceil n/\Delta \rceil$ nodes. Each edge in $\mathcal{T}_{tail}$ corresponds to a Δ-length substring of some $\mathsf{pEncode}(\mathsf{P}_i)$. We maintain a pointer to the start location of this substring in $\mathsf{pEncode}(\mathsf{P}_i)$. (Given an edge, this allows us to find any jth character of the corresponding Δ-length substring of $\mathsf{pEncode}(\mathsf{P}_i)$ in $\mathcal{O}(1)$ time. The purpose of this will become clear when we discuss how to query the trie.) The space needed to store this information is $\mathcal{O}((n/\Delta) \log n) = \mathcal{O}(n \log \sigma)$ bits.

For any node u in $\mathcal{T}_{tail}$, we use $\mathsf{path}(u)$ to denote the string obtained by concatenating the edge labels (which are characters from Σ') one the path from root to the node u, and $\mathsf{path}_e(u)$ to denote the expanded string for $\mathsf{path}(u)$ i.e., the string obtained by mapping each character of $\mathsf{path}(u)$ to its corresponding Δ-length string over Σ. For each node u, we maintain the following information.

- a *goto link* as in the case of the AC-automaton for navigating the trie: given a node u and a character $c \in \Sigma'$, we can find its child v where the edge (u, v) is (conceptually) labeled by c, or report that no such child exists. (This is facilitated by the hash-table $\mathcal{H}$, whereby we read Δ characters from T, encode it, and use it to find the corresponding character from Σ'.)

- a *failure link* as in the case of the AC-automaton: Let S be the largest proper suffix of $\mathsf{path}(u)$ for which there exists a node v, such that $\mathsf{path}_e(v)$ is same as the string obtained by expanding S, re-encoding it according to Observation 4, and then compressing it back. Then, the failure link of u points to v.
- an *output link* from u to the node v_i in $\mathcal{T}_{out}$ such that P_i is the longest pattern satisfying $\mathsf{pEncode}(\mathsf{P}_i) = \mathsf{pEncode}(\mathsf{path}_e(u)[|\mathsf{path}_e(u)| - |\mathsf{P}_i| + 1, |\mathsf{path}_e(u)|])$, where the re-encoding is according to Observation 4.
- $\mathsf{alphaDepth}(u)$ i.e., the number of distinct integers less than $|\Sigma_p|$ in $\mathsf{path}_e(u)$. (Conceptually, this is the number of distinct p-characters.)

The space required to maintain goto links, failure links, output links, and alphabet depth over all nodes is $\mathcal{O}(\lceil n/\Delta \rceil(\Delta \log \sigma + \log n + \log \sigma)) = \mathcal{O}(n \log \sigma)$ bits.

Lastly, we maintain a succinct representation of $\mathcal{T}_{tail}$ using the techniques of Sadakane and Navarro [28]. Using this, in $\mathcal{O}(1)$ time, we can find (i) node-depth of a node, and (ii) $\mathsf{levelAncestor}(u, D)$ = the node (if any) on the path from root to u that has node-depth D. (The root has depth zero.) The space needed is $2\lceil n/\Delta \rceil + o(n/\Delta) = \mathcal{O}(n/\Delta)$ bits.

In summary, $\mathcal{T}_{tail}$ occupies $\mathcal{O}(n \log \sigma + n/\Delta + d \log n) = \mathcal{O}(n \log \sigma + d \log n)$ bits.

Now, we focus on the head of each pattern. Consider a pattern P_i. First, we reverse $\mathsf{head}(\mathsf{P}_i)$, then encode it (as described in the beginning of this section). Create two copies of the resultant head, each of which is obtained by appending two special s-characters $\$_i$ and $\#_i$, neither of which belongs to Σ. Locate the (distinct) node u such that $\mathsf{path}_e(u)$ is same as $\mathsf{pEncode}(\mathsf{tail}(\mathsf{P}_i))$. Note that u is defined and we call it the *locus* of P_i. Consider all patterns which have the same locus u. Create a compacted trie for the modified heads of all those patterns, and let u be the root of that trie. We call this the *head-trie* of u and is denoted by $\mathcal{T}_{head}(u)$. The parent of each leaf in $\mathcal{T}_{head}(u)$ corresponds to a pattern, say P_j, in the dictionary. We mark all such nodes in $\mathcal{T}_{head}(u)$ and label them with the corresponding pattern index j. Furthermore, for each node in $\mathcal{T}_{head}(u)$, we maintain a pointer to its nearest marked ancestor. The space occupied by each node for marking and labeling is $\mathcal{O}(\log n)$ bits. Each edge in $\mathcal{T}_{head}(u)$ is labeled by a substring (of length less than Δ) of the encoded head of some pattern P_j. We maintain a pointer to the start point of the corresponding substring of $\mathsf{pEncode}(\mathsf{P}_j)$, and also its length. This occupies $\mathcal{O}(\log n)$ bits for each edge. We also equip $\mathcal{T}_{head}(u)$ to allow constant time navigation operation from a node to the edge where the next character of an encoded head matches. This can be facilitated using perfect hashing based on the (unique) first character of the edge to its children, and occupies $\mathcal{O}(\log \sigma)$ bits for each transition (edge). Since there are d patterns, the number of nodes and edges in all such tries combined is $\mathcal{O}(d)$. Thus, the total space occupied for maintaining all head-tries is $\mathcal{O}(d \log n + d \log \sigma) = \mathcal{O}(d \log n)$ bits.

In summary, the total space occupied by the resultant trie (denoted as $\mathcal{T}_{long}$), all encoded patterns, and the hash-table $\mathcal{H}$ is $\mathcal{O}(n \log \sigma + d \log n)$ bits.

2.3.2 Finding Occurrences

Starting from position $j = 1$, we obtain $\mathsf{pEncode}(T[j, \Delta])$ and use its corresponding character from Σ' to traverse the trie $\mathcal{T}_{long}$ from the root. We repeat this process for the next Δ characters from T, and so on. More specifically, suppose we have reached a node u in $\mathcal{T}_{head}$ such that $\mathsf{path}_e(u) = \mathsf{pEncode}(T[j, j + |\mathsf{path}_e(u)| - 1])$. At this point, we have the following cases to consider.

- There is an output link associated with u, implying the existence of a pattern which is a p-match with a suffix of T ending at $j + |\mathsf{path}_e(u)| - 1$. All such patterns and starting locations can be found out in $\mathcal{O}(1)$ time per output by using the output link and $\mathcal{T}_{out}$.

- $\mathcal{T}_{head}(u)$ is non-empty implying that there is a pattern P_i such that the encoded tail of P_i is same as $\mathsf{path}_e(u)$. To report all possible occurrences of such patterns ending at $(j + |\mathsf{path}_e(v)| - 1)$, we use the encoded characters corresponding to $T[j-1], T[j-2], \ldots, T[j-\Delta+1]$ to traverse $\mathcal{T}_{head}(u)$ until no more traversal is possible. Suppose the last encountered node in this trie is v. We report all patterns with an occurrence ending at j by following the marked ancestor linkage from v.
- There is a child v of u such that the edge label of (u, v) is same as the character from Σ' corresponding to the last Δ characters of $\mathsf{pEncode}(T[j, j + |\mathsf{path}_e(v)| - 1])$. In this case, we traverse to v, and continue the process. Otherwise, follow the failure link of u.

Note that following the output link results in occurrence of at least one pattern. Each occurrence (i.e., the index and the corresponding pattern) can be reported in $\mathcal{O}(1)$ time. Moving forward we show how to deal with the head-trie, failure links, and goto links.

For our purposes, we maintain an array A of length $|\Sigma_p|$ such that for any $c \in \Sigma_p$, $A[c]$ equals the last position at which c appeared in T that has been read so far. (Initially each entry in the array A is empty.) We also maintain an array B of length $|\Sigma_p|$ such that for any $c \in \Sigma_p$, $B[c]$ gives us the desired encoding.

First, we show how to appropriately encode the incoming characters $T[j-1], T[j-2], \ldots, T[j-\Delta+1]$ when we traverse $\mathcal{T}_{head}(u)$. Initialize the array B to be empty. Note that it suffices to find the encoding for the first occurrence of every p-character starting from $j-1$ as the encoding for all future occurrences remains the same and can be obtained using B. Let c be a p-character. If $B[c]$ is not empty then use it to obtain the desired encoding. Otherwise, find the last occurrence of c using $A[c]$. We use the state of the array A at node u, and do not modify it while traversing the head-trie. We have the following two cases.

- **c appears in $T[j, j + |\mathsf{path}_e(u)| - 1]$:** Assume that the last occurrence is the λth character starting from j and (u', v') be the edge on which this occurrence lies i.e., $|\mathsf{path}(u')| < \lambda/\Delta \leq |\mathsf{path}(v')|$. We locate $v' = \mathsf{levelAncestor}(u, D)$ and $u' = \mathsf{levelAncestor}(u, D-1)$, where $D = |\mathsf{path}(v')| = \lceil \lambda/\Delta \rceil$ is the node-depth of v'. The encoding corresponding to c is exactly the $(\lambda - \Delta \cdot |\mathsf{path}(u')|)$th character of the label on this edge, and can be found using the pointer from the edge to the start of the corresponding substring of some encoded pattern tail. Set $B[c]$ to the encoded value. The time needed is $\mathcal{O}(1)$ per character.
- **c does not appear in $T[j, j + |\mathsf{path}_e(u)| - 1]$:** We maintain a counter C initialized to $\mathsf{alphaDepth}(u)$. Whenever we encounter such a c, the encoding is given by the value of C. Set $B[c]$ to the value of the counter. Following this, we increment C by one. The time needed is $\mathcal{O}(1)$ per character.

Thus, the time required to traverse each head-trie is $\mathcal{O}(\Delta)$ and each occurrence in the head-trie is reported in $\mathcal{O}(1)$ time by following the marked ancestor linkage.

Now, we concentrate on the failure link from u to v and show how to re-encode the text when we truncate characters from position j. Assume that k is the number of edges on the path from root to u (i.e., k is the node-depth of u) and that the failure link truncates $k'\Delta$ characters starting from j. Clearly, $1 \leq k' \leq k$. Therefore, we are now trying to find a match for the positions starting from $j' = j + k'\Delta$ and we need to re-encode the text T starting from j'. Since it is ensured that $\mathsf{pEncode}(T[j', j' + (k-k')\Delta - 1])$ is same as $\mathsf{path}_e(v)$, we are required to find the encoding of every p-character starting from $j'' = j' + (k-k')\Delta$.

Initialize the array B to be empty. Note that it suffices to find the encoding for the first occurrence of every p-character starting from j'', as the encoding for all future occurrences remains the same and can be obtained using the array B. Let c be a p-character. If $B[c]$ is

non-empty, then use it to obtain the desired encoding. Otherwise, find the last occurrence of c using $A[c]$. Note that we need the state of the array A at node u, which can be easily obtained by maintaining a copy of it whenever a new edge is traversed. (We delete the old copy when a new edge is traversed as it will not be required any more.) We have the following two cases.

- **c appears in $T[j', j'' - 1]$:** As described previously using $\mathsf{levelAncestor}(\cdot, \cdot)$ queries, we locate the position on the edge of the last occurrence. Then using the pointer from the edge we find the desired encoding and set $B[c]$. The time needed is $\mathcal{O}(1)$ per character.
- **c does not appear in $T[j', j''-1]$:** We maintain a counter C initialized to $\mathsf{alphaDepth}(v)$. Whenever we encounter such a c, the encoding is given by the value of C. Following this, we increment C by one. The time needed is $\mathcal{O}(1)$ per character.

The goto transition is achieved easily as follows. We read the next Δ characters from the text, encode them, and use the hash table $\mathcal{H}$ to traverse to the desired node. Since encoding each character can be performed in $\mathcal{O}(1)$ time (using the arrays A and B as described previously), each goto operation takes $\mathcal{O}(\Delta)$ time.

Now, we bound the query complexity. Initially, encoding the string T starting from $j = 1$ can be performed in $\mathcal{O}(|T|)$ time. Recall that on following a failure link, we truncate at least Δ characters starting from j. We read Δ characters on the failed edge (i.e., the one which was read unsuccessfully immediately before following the failure link). Thus, we can charge the characters on the failed edge to the first truncated Δ characters. This gives us an amortized complexity of $\mathcal{O}(1)$ per character. The number of failure link operations is at most $\lceil |T|/\Delta \rceil$. Thus, the number of nodes and edges traversed in the tail-trie is $\mathcal{O}(|T|/\Delta)$. For each edge, we read Δ characters and encoding the p-characters can be performed in $\mathcal{O}(1)$ time per character. For each node in the tail-trie, we will examine less than Δ characters in the head-trie; each of these characters can be appropriately encoded in $\mathcal{O}(1)$ time. Thus, the time required to traverse $\mathcal{T}_{long}$ (without reporting occurrences) is $\mathcal{O}((|T|/\Delta) \cdot \Delta) = \mathcal{O}(|T|)$. Each occurrence in the head-trie or the output tree is reported in $\mathcal{O}(1)$ time.

At the end of this process, for $j = 1$, we have reported occurrences of all patterns which end at a position of the form $j, j + \Delta, j + 2\Delta, \ldots$. The time required is $\mathcal{O}(|T| + occ_j)$. By repeating the process for $j = 2, 3, \ldots, \Delta$, all occ_ℓ occurrences of long patterns are reported in $\mathcal{O}(|T|\Delta + occ_\ell) = \mathcal{O}(|T| \log_\sigma n + occ_\ell)$ time.

Summarizing the discussions in this section, we obtain Lemma 5.

2.4 Short Patterns (Proof of Lemma 6)

Processing short patterns (having length less than Δ) is similar to that for head-tries. For all short patterns P_i, we create a compacted trie $\mathcal{T}_{short}$ for the strings $\mathsf{pEncode}(\mathsf{P}_i) \circ \$_i$ and $\mathsf{pEncode}(\mathsf{P}_i) \circ \#_i$, where $\circ$ denotes concatenation. The number of nodes in the trie is $\mathcal{O}(d)$. As in case of tail tries, we maintain a pointer from each edge to the start of the corresponding substring labeling the edge, and the length of the substring. We also equip each edge of $\mathcal{T}_{short}$ to support constant time navigation. Mark all nodes u if there is an encoded pattern which is the same as that obtained by concatenating the edge labels from root to u. The total space is bounded by $\mathcal{O}(d \log n)$ bits.

To find occurrences of short patterns, we use a rather brute force approach. Starting from $j = 1$, simply encode the next Δ characters of T, and use it to traverse the trie $\mathcal{T}_{short}$ until no more traversal is possible. Report j if at least one marked node is encountered in this traversal, and in that case, also report the patterns corresponding to these marked nodes. We repeat the process for $j = 2, 3, \ldots, |T|$. Since for each j at most 2Δ characters are

checked, the time required to report all occ_s occurrences of short patterns is $\mathcal{O}(|T|\Delta + occ_s) = \mathcal{O}(|T| \log_\sigma n + occ_s)$.

Summarizing the discussions in this section, we obtain Lemma 6.

3 Order-Preserving Dictionary Matching

As in the case of parameterized matching, we assume that the patterns are over an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, where the total-order on Σ is the natural order of integers. By initially pre-processing the patterns in $\mathcal{O}(n \log \sigma)$ time this condition is ensured. We use the following encoding scheme to convert a string S over Σ to a string $\mathsf{oEncode}(S)$. For every character $S[i]$, $\mathsf{oEncode}(S)[i]$ is the number of distinct characters in $S[1, i]$ having value at most $S[i]$. For example, consider the string $S = 5452316$, where each character is a single-digit integer. Then, $\mathsf{oEncode}(S) = 1121216$. The following is due to Kim et al. [20].

▶ **Observation 8.** *Two strings S and S' are an* o-match *iff* $\mathsf{oEncode}(S) = \mathsf{oEncode}(S')$. *A string S matches another string S' at a position i iff* $\mathsf{oEncode}(S) = \mathsf{oEncode}(S'[i, i + |S| - 1])$.

3.1 Creating the Index

As in case of p-patterns, we categorize o-patterns into long and small w.r.t the same parameter $\Delta = \lceil \log_\sigma n \rceil$. We also define the head and tail of the pattern similar to that in case of p-patterns. The tails are encoded using $\mathsf{oEncode}(\cdot)$ and are maintained explicitly. The encoding for the head of a pattern P_i is obtained as follows. Create a string P'_i first by reversing $\mathsf{head}(\mathsf{P}_i)$, then appending it at the end of $\mathsf{tail}(\mathsf{P}_i)$. Then, the encoding of the jth character in the reversed head is given by $\mathsf{oEncode}(\mathsf{P}'_i)[|\mathsf{tail}(\mathsf{P}_i)| + j]$. The following observation is due to the definition of o-match and Fact 8.

▶ **Observation 9.** *Let S and S' two be strings having equal length. Then S and S' are an o-match iff both the conditions are satisfied: (i) the o-encoded tails of both S and S' are equal and (ii) the o-encoded heads (as described above) of both S and S' are equal.*

For long patterns, the index is similar to that for p-patterns, except that we use the above encoding scheme. Also, we do not pre-process the resultant trie $\mathcal{T}_{tail}$ for answering $\mathsf{levelAncestor}(\cdot, \cdot)$-queries. For short patterns, the index is same except that for the encoding scheme. Thus, space is bounded by $\mathcal{O}(n \log \sigma + d \log n)$ bits.

3.2 Finding Occurrences

We assume that the characters in $|T|$ are from $\{1, 2, \ldots, \sigma\}$ with the total order being same as in that of the patterns. An initial pre-processing of the text in $\mathcal{O}(|T| \log \sigma)$ time ensures that this condition holds. Note that this does not affect the final query complexity of Theorem 2.

The querying process remains exactly similar to that for p-matching. Obviously, we use $\mathsf{oEncode}(\cdot)$ for encoding T, computing which requires a different technique. We maintain an array A of length σ such that $A[c]$ equals the position of last occurrence (if any) of $c \in \Sigma$ in the text read so far. Initially, for each $c \in \Sigma$, we assign $A[c] = -1$. Also, we maintain a balanced binary search tree (BST) $\mathcal{T}_{bin}$, which is initially empty. Suppose we are at position k in the text. If $A[T[k]] = -1$, we add $T[k]$ to $\mathcal{T}_{bin}$. We find the number of characters in $\mathcal{T}_{bin}$ that are at most $T[k]$, which gives us the desired encoding. Then, we update $A[T[k]] = k$ and proceed. Note that the size of $\mathcal{T}_{bin}$ is $\mathcal{O}(\sigma)$, which implies every deletion, insertion, and search operation requires $\mathcal{O}(\log \sigma)$ time.

The difficulty comes when we follow a failure link or when we traverse a head-trie. We first discuss the case of a failure link in which we may have to remove several characters from $\mathcal{T}_{bin}$. Suppose, after following a failure link, we are trying to find an occurrence for position j' and we are processing characters of the text starting from j''. (See Section 2.3.2 for detailed definitions of j' and j''.) Clearly, we have to remove those characters c from $\mathcal{T}_{bin}$ for which $A[c] < j'$. The total number of such deletions is at most $|T|$, each requiring $\mathcal{O}(\log \sigma)$ time yielding an amortized time complexity of $\mathcal{O}(\log \sigma)$ per character. To find these characters efficiently, we maintain the characters $c' \in \Sigma$ keyed by $A[c']$ in another BST $\mathcal{T}'_{bin}$. Note that the size of $\mathcal{T}'_{bin}$ is $\mathcal{O}(\sigma)$, which implies every insertion, update, and search operation requires $\mathcal{O}(\log \sigma)$ time. Using $\mathcal{T}'_{bin}$, we can find the desired characters to be removed in $\mathcal{O}(\log \sigma + output_k)$ time, where $output_k$ is the number of characters to be deleted from $\mathcal{T}_{bin}$ when we follow the kth failure link. Note that $\sum_k output_k \leq |T|$. Therefore, maintaining $\mathcal{T}'_{bin}$ and finding the desired characters to be removed on following a failure link have an amortized time complexity of $\mathcal{O}(\log \sigma)$ per character.

Traversing a head-trie is achieved similarly. Suppose, we are considering the string $T[j, j' - 1]$. Then, we have to encode characters $T[j-1], T[j-2], \ldots, T[j-\Delta+1]$ based on $T[j, j' - 1]$. With the aid of $\mathcal{T}'_{bin}$, we maintain another BST that contains only the characters in the interval $[j, j' - 1]$. The desired encoding of each character can be obtained in $\mathcal{O}(\log \sigma)$ amortized time.

Thus, the jth running of the algorithm requires $\mathcal{O}(|T| \log \sigma + occ_j)$ time for long patterns. Reporting all occ_ℓ occurrences of long patterns requires $\mathcal{O}(|T|\Delta \log \sigma + occ_\ell) = \mathcal{O}(|T| \log n + occ_\ell)$ time. For short patterns, since we follow the same brute-force strategy, it is easy to see that time required to report all occ_s occurrences is $\mathcal{O}(|T| \log n + occ_s)$.

This completes the proof of Theorem 2.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

2 Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994. doi:10.1016/0020-0190(94)90086-8.

3 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80, 1993. doi:10.1145/167088.167115.

4 Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 88–100, 2010. doi:10.1007/978-3-642-13509-5_9.

5 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014. doi:10.1145/2635816.

6 Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Forbidden extension queries. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, pages 320–335, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.320.

7 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving incomplete suffix trees and order-preserving indexes. In *String Processing and Information Retrieval – 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, pages 84–95, 2013. doi:10.1007/978-3-319-02432-5_13.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

9 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

10 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly fm-index. In *String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, Padova, Italy, October 5-8, 2004, Proceedings*, pages 150–160, 2004. doi:10.1007/978-3-540-30213-1_23.

11 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007. doi:10.1145/1240233.1240243.

12 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Succinct non-overlapping indexing. In *Combinatorial Pattern Matching – 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 – July 1, 2015, Proceedings*, pages 185–195, 2015. doi:10.1007/978-3-319-19929-0_16.

13 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

14 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000. doi:10.1145/335305.335351.

15 Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. In *Algorithms – ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, pages 414–425, 2004. doi:10.1007/978-3-540-30140-0_38.

16 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Compressed index for dictionary matching. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA*, pages 23–32, 2008. doi:10.1109/DCC.2008.62.

17 Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *J. ACM*, 61(2):9:1–9:36, 2014. doi:10.1145/2590774.

18 Ramana M. Idury and Alejandro A. Schäffer. Multiple matching of parameterized patterns. In *Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994, Proceedings*, pages 226–239, 1994. doi:10.1007/3-540-58094-8_20.

19 Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 – March 2, 2013, Kiel, Germany*, pages 400–411, 2013. doi:10.4230/LIPIcs.STACS.2013.400.

20 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.

21 S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 631–637, 1995. doi:10.1109/SFCS.1995.492664.

22 Veli Mäkinen and Gonzalo Navarro. Compressed compact suffix arrays. In *Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings*, pages 420–433, 2004. doi:10.1007/978-3-540-27801-6_32.

23 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005, Proceedings*, pages 45–56, 2005. doi:10.1007/11496656_5.

24 J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top- k term-proximity in succinct space. In *Algorithms and Computation – 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 169–180, 2014. doi:10.1007/978-3-319-13075-0_14.

25 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007. doi:10.1145/1216370.1216372.

26 Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*, pages 410–421, 2000. doi:10.1007/3-540-40996-3_35.

27 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003. doi:10.1016/S0196-6774(03)00087-7.

28 Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 134–149, 2010. doi:10.1137/1.9781611973075.13.

29 Alan Tam, Edward Wu, Tak Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, pages 39–50, 2009. doi:10.1007/978-3-642-03784-9_5.

30 Dekel Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013. doi:10.1016/j.ipl.2013.03.012.