Fully Functional Parameterized Suffix Trees in Compact Space

Arnab Ganguly
Dept. of Computer Science, University of Wisconsin, Whitewater, WI, USA

Rahul Shah
Dept. of Computer Science, Louisiana State University, Baton Rouge, LA, USA

Sharma V. Thankachan
Dept. of Computer Science, University of Central Florida, Orlando, FL, USA

Abstract

Two equal length strings are a parameterized match (p-match) iff there exists a one-to-one function that renames the symbols in one string to those in the other. The *Parameterized Suffix Tree* (PST) [Baker, STOC' 93] is a fundamental data structure that handles various string matching problems under this setting. The PST of a text $T[1, n]$ over an alphabet Σ of size σ takes $O(n \log n)$ bits of space. It can report any entry in (parameterized) (i) suffix array, (ii) inverse suffix array, and (iii) longest common prefix (LCP) array in $O(1)$ time. Given any pattern P as a query, a position i in T is an occurrence iff $T[i, i + |P| - 1]$ and P are a p-match. The PST can count the number of occurrences of P in T in time $O(|P| \log \sigma)$ and then report each occurrence in time proportional to that of accessing a suffix array entry. An important question is, *can we obtain a compressed version of PST that takes space close to the text's size of $n \log \sigma$ bits and still support all three functionalities mentioned earlier*? In SODA' 17, Ganguly et al. answered this question partially by presenting an $O(n \log \sigma)$ bit index that can support (parameterized) suffix array and inverse suffix array operations in $O(\log n)$ time. However, the compression of the (parameterized) LCP array and the possibility of faster suffix array and inverse suffix array queries in compact space were left open. In this work, we obtain a compact representation of the (parameterized) LCP array. With this result, in conjunction with three new (parameterized) suffix array representations, we obtain the first set of PST representations in $o(n \log n)$ bits (when $\log \sigma = o(\log n)$) as follows. Here $\varepsilon > 0$ is an arbitrarily small constant.

- Space $O(n \log \sigma)$ bits and query time $O(\log_\sigma^\varepsilon n)$;
- Space $O(n \log \sigma \log \log_\sigma n)$ bits and query time $O(\log \log_\sigma n)$; and
- Space $O(n \log \sigma \log_\sigma^\varepsilon n)$ bits and query time $O(1)$.

The first trade-off is an improvement over Ganguly et al.'s result, whereas our third trade-off matches the optimal time performance of Baker's PST while squeezing the space by a factor roughly $\log_\sigma n$. We highlight that our trade-offs match the space-and-time bounds of the best-known compressed text indexes for exact pattern matching and further improvement is highly unlikely.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Data Structures, Suffix Trees, String Algorithms, Compression

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.65

Category Track A: Algorithms, Complexity and Games

Funding Supported under US NSF grants CCF-1527435, CCF-2112643 and CCF-2137057.

1 Introduction

Text Indexing is a classical problem in Computer Science with numerous applications. The objective is to pre-process a text $T[1, n]$ over an alphabet Σ of size σ to create a data structure, such that for any pattern P given as a query, we can count/report all the positions in T where P appear as a substring. The suffix trees and suffix arrays (along with Longest Common

Prefix array, LCP array in short) are the most widely-known text indexes [21, 37, 28]. They occupy $\Theta(n)$ words of space (equivalently, $\Theta(n \log n)$ bits) and can count the number of occurrences in time $\tilde{O}(|P|)$. The time for per occurrence reporting is a constant for both structures. Although the space is linear in the number of words, there is an $O(\log_\sigma n)$ factor blowup when we consider the actual text size, which is $n\lceil \log \sigma \rceil$ bits. This factor is not negligible when $\sigma \ll n$. For example, the space occupied by the suffix tree of the human genome, even with very efficient implementation, such as in [27], requires about 40 GB of space, whereas the genome occupies less than 1GB.

To address the above issue, Grossi and Vitter [20] and Ferragina and Manzini [7] introduced succinct/compressed space alternatives, respectively known as *Compressed Suffix Array* (CSA) and *FM Index*. They can answer counting queries in $\tilde{O}(|P|)$ time and reporting in $\tilde{O}(1)$ time per occurrence. In some sense, both structures exploit the so-called *rank-preserving property* of suffixes/leaves. Specifically, consider two leaves/suffixes in the sub-tree of a non-root node in the *classical* suffix tree. If one were to chop off the first character of the suffixes corresponding to these leaves, thus leading to two different suffixes, the relative ordering of the first two suffixes within the suffix tree would be the same as their chopped counterparts. This crucial property leads to an efficient implementation of *Last-to-Front (LF) mapping*, which is defined as follows: *given the leaf i corresponding to a suffix starting at position t in the text,* $\mathsf{LF}(i)$ *is the leaf corresponding to the suffix starting at position* $(t-1)$. The LF mapping (or its inverse Ψ function) plays a pivotal role in the working of FM-Index and CSA and their subsequent improvements. Later, Sadakane [35] showed that by storing $O(n)$ extra bits, we could also compute the LCP of any two suffixes in $\tilde{O}(1)$ time, leading to the first fully functional suffix tree representation in $O(n \log \sigma)$ bits – i.e., it can report suffix array, inverse suffix array and LCP values. See [30] for further reading.

For numerous variants of the text indexing problem [1, 3, 18, 23, 32, 36] (such as parameterized matching, order-preserving matching, two-dimensional matching, cartesian tree matching, etc.), although linear space indexes are known, designing succinct/compressed indexes has been challenging [4, 5, 10, 13, 15, 11, 17, 12, 14, 25, 26, 24, 33]. We focus on the parameterized matching problem [1] defined as follows: two equal-length strings X and Y are a parameterized match (p-match) if and only if there exists a one-to-one function $f : \Sigma \to \Sigma$ such that $Y[i] = f(X[i])$ for every $i \in [1, |Y|]$. For example, $xyxz$ and $yzyx$ are p-match, but $xyxz$ and $xywz$ are not p-match. The indexing version is to count/report all substrings of $T[1, n]$ that p-match with a query pattern P. An index of size $\Theta(n \log n)$ bits, namely *parameterized suffix tree* (PST), has been known due to Baker [1]. However, the problem of designing a space-efficient avatar of PST turns out to be challenging because the above described *rank-preserving property* is no longer valid here. To that end, Ganguly et al. [16] proposed the Parameterized Burrows-Wheeler Transform (pBWT) that can support (parameterized) LF mapping in $O(\log \sigma)$ time using space close to $n \log \sigma$ bits. This led to the first sub-linear space index $\big($when $\log \sigma = o(\log n)\big)$ that can support (parameterized) suffix array and inverse suffix array operations in $\tilde{O}(1)$ time. Although this index is a significant achievement, it does not support LCP queries. In this paper, we augment this missing functionality, leading to the first fully functional PST representation in compact space. Besides this, we present three new space-time trade-offs (for suffix array access and its inverse operation) that are clear improvements over the previous results.

1.1 Baker's Parameterized Suffix Tree

We will use the following terminologies: for a string S, $|S|$ is its length, $S[i], 1 \leq i \leq |S|$, is its ith character and $S[i, j] = S[i] \circ S[i+1] \circ \cdots \circ S[j]$, where $\circ$ denotes *concatenation.* If $i > j$, $S[i, j]$ denotes an empty string. Also, S_i denotes the circular suffix starting at position i. Specifically, S_i is S if $i = 1$ and is $S[i, |S|] \circ S[1, i-1]$ otherwise.

Baker [1] introduced the following encoding scheme for matching strings over Σ. Let \$ be a special character in Σ. A string S is encoded into a string $\mathsf{prev}(S)$ of length $|S|$ by replacing the first occurrence of every character (other than \$) in S by 0 and any other occurrence by the difference in text position from its previous occurrence. Specifically, for any $i \in [1, |S|]$, $\mathsf{prev}(S)[i] = S[i]$ if $S[i] = \$$; otherwise, $\mathsf{prev}(S)[i] = (i - j)$, where $j < i$ is the last occurrence of $S[i]$ before i. If j does not exist, then $\mathsf{prev}(S)[i] = 0$. For example, $\mathsf{prev}(xy\$x) = 00\$3$. Note that $\mathsf{prev}(S)$ is a string over $\Sigma' = \{\$, 0, 1, \ldots, |S| - 1\}$, and can be computed in time $O(|S| \log \sigma)$.

▶ **Convention 1.** *In Σ', the integer characters are lexicographically smaller than \$. An integer character i comes before another integer character j iff $i < j$.*

▶ **Fact 2** ([1])**.** *Two (equal length) strings S and S' are a p-match iff $\mathsf{prev}(S) = \mathsf{prev}(S')$. Also a string P and a prefix of S are a p-match iff $\mathsf{prev}(P)$ is a prefix of $\mathsf{prev}(S)$.*

The parameterized Suffix Tree (PST) of $T[1, n]$ is a compacted trie of all strings in $\mathcal{P} = \{\mathsf{prev}(T[k, n]) \mid 1 \leq k \leq n\}$. For convenience, we assume that $T[n] = \$$ and $T[i] \neq \$$ for all $i \neq n$. Each edge is labeled with a string over Σ'. We use $\mathsf{str}(u)$ to denote the concatenation of edge labels on the path from the root to node u and $\mathsf{strLen}(u) = |\mathsf{str}(u)|$. Clearly, PST consists of n leaves (one per each encoded suffix) and at most $n - 1$ internal nodes. We use ℓ_i to denote the ith leftmost leaf and $\mathsf{str}(\ell_i)$ to denote the ith lexicographically smallest string in $\mathcal{P}$. Also, $\mathsf{PSA}[1, n]$ is an associated array called the parameterized suffix array, where $\mathsf{PSA}[i] = j$ and $\mathsf{PSA}^{-1}[j] = i$ iff $\mathsf{prev}(T[j, n]) = \mathsf{str}(\ell_i)$. Let $\mathsf{plcp}(i, j)$ be $\mathsf{strLen}(u)$, where u is the lowest common ancestor (LCA) of ℓ_i and ℓ_j; equivalently the length of the LCP of $\mathsf{prev}(T_{\mathsf{PSA}[i]})$ and $\mathsf{prev}(T_{\mathsf{PSA}[j]})$. The parameterized LCP array $\mathsf{PLCP}[1, n)$ is defined as follows: $\mathsf{PLCP}[i] = \mathsf{plcp}(i, i + 1)$. See Figure 1 for an illustration. Since $\mathsf{plcp}(i, j)$ is the smallest element in $\mathsf{PLCP}[i, j - 1]$, by maintaining an $O(n)$-bit range minimum query data structure [8] over PLCP, we can compute $\mathsf{plcp}(i, j)$ for any i, j in $O(1)$ time.

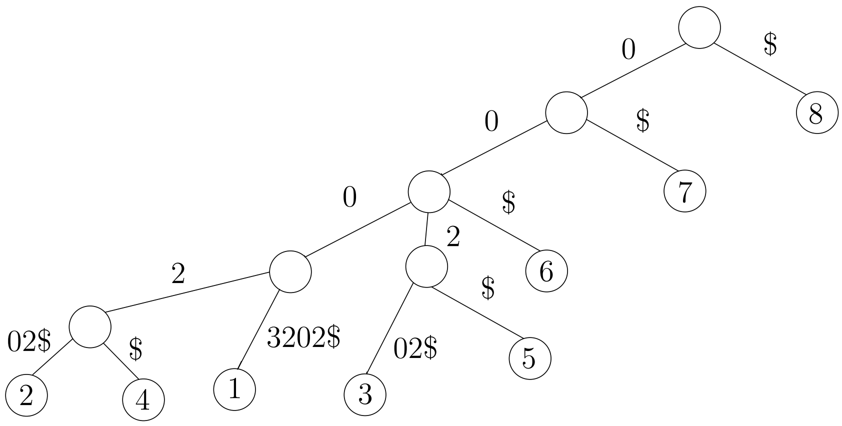
To answer a pattern matching query P (which is a string over $\Sigma - \{\$\}$), traverse the PST from the root and find the highest node u_P (if it exists) such that $\mathsf{str}(u_P)$ is prefixed by $\mathsf{prev}(P)$. This step takes $O(|P| \log \sigma)$ time. Then, find the range $[sp, ep]$ of leaves (called the suffix range of P) under u_P (this can be found in constant time by pre-processing the tree). Output $ep - sp + 1$ as the answer to counting and output $\{\mathsf{PSA}[i] \mid sp \leq i \leq ep\}$ as the answer to reporting. If u_P does not exist, we conclude that P does not have any p-match within T.

We refer to [6, 9, 29] for several other (linear space) data structures for parameterized pattern matching.

1.2 Compact Encoding of Parameterized Suffix Array

The parameterized LF mapping is defined as $\mathsf{PLF}(i) = \mathsf{PSA}^{-1}[\mathsf{PSA}[i] - 1]$. In [16], Ganguly et al. showed that one can implement PLF in $O(\log \sigma)$ time using an $n \log \sigma + o(n \log \sigma) + O(n)$ bit index. Their index constitutes the parameterized Burrows-Wheeler Transform (PWBT), which is an array of length n, such that $\mathsf{PBWT}[i]$ stores the number of distinct characters in (the prefix of) $T_{\mathsf{PSA}[i]}$ until the first occurrence $T[\mathsf{PSA}[i] - 1]$. See Figure 1 for an illustration.

By maintaining a Wavelet Tree [19] over PBWT, coupled with a succinct encoding [31] of the structure of the PST, they showed that PSA can be represented in $n \log \sigma + O\big(n + (n/\Delta) \log n\big)$ bits to support $\mathsf{PSA}[\cdot]/\mathsf{PSA}^{-1}[\cdot]$ queries in $t_{\mathsf{PSA}} = O(\Delta \cdot \log \sigma)$ time for any



i	T_i	prev(T_i)	prev($T_{\mathsf{PSA}[i]}$)	$T_{\mathsf{PSA}[i]}$	PSA[i]	f_i	PBWT[i]	W[i]	PLF(i)	$\Psi(i)$
1	xyzxzwz$	0003202$	000202$5	yzxzwz$x	2	8	3	4	3	4
2	yzxzwz$x	000202$5	0002$504	xzwz$xyz	4	5	2	3	4	5
3	zxzwz$xy	00202$50	0003202$	xyzxzwz$	1	3	$	3	8	1
4	xzwz$xyz	0002$504	00202$50	zxzwz$xy	3	2	4	2	1	2
5	zwz$xyzx	002$0043	002$0043	zwz$xyzx	5	2	3	2	2	6
6	wz$xyzxz	00$00432	00$00432	wz$xyzxz	6	8	2	4	5	7
7	z$xyzxzw	0$004320	0$004320	z$xyzxzw	7	4	4	3	6	8
8	$xyzxzwz	$0003202	$0003202	$xyzxzwz	8	∅	3	$	7	3

Figure 1 The text is $T[1,8] = xyzxzwz\$$, where $\Sigma = \{w, x, y, z, \$\}$.

$\Delta = O(\log_\sigma n)$ fixed in advance. For example, $O(n \log \sigma)$ bits of space and $O(\log n)$ query time by fixing $\Delta = \log_\sigma n$. This is the first succinct/compact space representation of PSA.[1] However, it does not support $\mathsf{plcp}(\cdot,\cdot)$ queries.

1.2.1 Challenges in Making PLF Computation Faster

Note that the product of space (in bits) and query time of Ganguly et al.'s PSA is always $\Theta(n \log n \log \sigma)$. A natural question is: *can we obtain better trade-offs*?

The current index is limited primarily because its main component for computing parameterized LF mapping needs to support queries of the following type: $\mathsf{RangeCount}_{\mathsf{PBWT}}(i,j,x,y) = |\{k \mid k \in [i,j], \mathsf{PBWT}[k] \in [x,y]\}|$. From the 4-sided range counting lower bound [34], any $O(n \log^{O(1)} \sigma)$-bit data structure needs $\Omega(1 + \log \sigma / \log \log n)$ time. This time becomes a bottleneck when it comes to some of the advanced suffix sampling techniques that are used for speeding up (classical) suffix array queries using additional space (as listed in Theorem 3); in fact, to adapt these techniques, LF mapping needs to be implemented in $O(1)$ time. Therefore, to prove Theorem 3, we need a new set of techniques.

1.2.2 Challenges in Compressing Parameterized LCP Array

Sadakane's LCP compression framework [35] for traditional text indexing relies on the following: *if two suffixes begin with the same character, their LCP after chopping the first character will be one less than their original LCP, and these two suffixes will retain*

[1] A succinct index for a data of size Z bits is a data structure having $Z + o(z)$ bits. On the other hand, a compact index needs $O(Z)$ bits.

their relative lexicographic rank after chopping. This allows one to compactly encode the LCP information. Unfortunately, this is not true for parameterized strings. For e.g., let $X = wxywabcdwx\$$ and $Y = abcdwx\$$ be two suffixes of T, then their respective prev encodings are $\mathsf{prev}(X) = 0003000058\$$ and $\mathsf{prev}(Y) = 000000\$$; hence, their p-LCP is 3. After chopping the first characters, the respective prev encodings are 000000058\$ and 00000\$, resulting in a p-LCP value of 5. Thus, chopping the first character can increase LCP; in fact, it can also decrease or even remain the same! Moreover, the order of the suffix may switch (as seen in this example), which adds to the difficulty. In short, the previous techniques are not adequate for compressing parameterized LCP array.

1.3 Our Results: Fully Functional PST in Compact Space

The suffix range $[sp, ep]$ of a pattern P can be computed in $O(|P| \log \sigma)$ time using Ganguly et al.'s index [16]; so we focus on speeding up suffix array queries and reporting LCP. We overcome the $O(\log \sigma)$ bottleneck of parameterized LF mapping by using its inverse, the Ψ-function, defined as $\Psi(i) = j$ iff $\mathsf{PLF}(j) = i$. This allows us to remove the dependence on 4-sided range-queries, instead of using simpler $\mathsf{partialRank}$ and select queries, which can be supported in $O(1)$ time using succinct space. With this, we implement Ψ-function in $O(1)$ time and thereby obtain three trade-offs, with space-time product near $n \log \sigma$. For our LCP framework, we essentially reduce a parameterized LCP query to a traditional LCP query; this allows us to leverage Sadakane's framework [35].

In summary, we have the following theorem.

▶ **Theorem 3.** *For the parameterized suffix tree of a text $T[1, n]$ over an alphabet of size σ, the following space-time trade-offs are possible in the word RAM model of computation with word-size $\Omega(\log n)$, where $\varepsilon > 0$ is an arbitrarily small constant.*

Index Size (in bits)	*Query Time (t_{PSA})*
$O(n \log \sigma)$	$O(\log_\sigma^\varepsilon n)$
$O(n \log \sigma \log \log_\sigma n)$	$O(\log \log_\sigma n)$
$O(n \log \sigma \log_\sigma^\varepsilon n)$	$O(1)$

All three basic queries (i.e., $\mathsf{PSA}[\cdot], \mathsf{PSA}^{-1}[\cdot]$ and $\mathsf{plcp}(\cdot,\cdot)$) are supported in $O(t_{\mathsf{PSA}})$ time.

Note that Baker's original definition also includes "*static characters*" for which the match has to be done in the traditional way. For the simplicity of exposition, we assume that all characters in Σ, except \$ are parameterized characters. We remark that our index can be extended to incorporate static characters without any sacrifice in time or space.

Outline. We start in Section 2 with a weaker version of Theorem 3 without the LCP claims. Specifically, we show that using an $O(n \log \sigma)$ bit index, we can support PSA and PSA^{-1} queries in $O(\log_\sigma n)$ time. Note that this is already a factor $(\log \sigma)$ faster than what is achievable using Ganguly et al.'s index [16]. Using more intricate techniques, we obtain the $\mathsf{PSA}[\cdot]/\mathsf{PSA}^{-1}[\cdot]$ trade-offs in Sections 3 and 4. Finally, the technique for encoding the LCP array is in Section 5.

2 Our Framework: A Compact Space Index

Let's start with a few definitions that we are going to use throughout this paper.

▶ **Definition 4.** *Define the following (see Figure 1 for an illustration):*

Notation	***Definition***
$\Psi(i)$	$\mathsf{PSA}^{-1}[1]$ *if* $\mathsf{PSA}[i] = n$, *else* $\mathsf{PSA}^{-1}[\mathsf{PSA}[i] + 1]$
$\mathsf{PLF}(i)$	$\mathsf{PSA}^{-1}[n]$ *if* $\mathsf{PSA}[i] = 1$, *else* $\mathsf{PSA}^{-1}[\mathsf{PSA}[i] - 1]$
f_i	$\varnothing$ if $i = n$, *else the first occurrence of* $T[\mathsf{PSA}[i]]$ *in* $T_{1+\mathsf{PSA}[i]}$
$\mathsf{W}[i]$	\$ if $i = n$, *else number of zeroes in* $\mathsf{prev}(T_{1+\mathsf{PSA}[i]})[1, \mathsf{f}_i]$
$\mathsf{PBWT}[i]$	$\mathsf{W}[\mathsf{PLF}(i)]$

Our goal is to prove the following theorem in this section.

▶ **Theorem 5.** *By using an $O(n \log \sigma)$-bit index, we can compute $\Psi(i)$ in $O(1)$ time.*

Before we prove this, we will see how we can use it to achieve an $O(n \log \sigma)$-bit index that supports PSA and PSA^{-1} queries in $O(\log_\sigma n)$ time. We explicitly store $\mathsf{PSA}[i]$ iff it equals n or is a multiple of $\Delta = \lceil \log_\sigma n \rceil$. Additionally, we store a bit-vector $B[1, n]$ as follows: set $B[i] = 1$ iff $\mathsf{PSA}[i]$ has been explicitly stored. For reporting, a $\mathsf{PSA}[j]$ can be retrieved in $O(1)$ time if $B[j] = 1$. Otherwise, we repeatedly apply Ψ starting from j until we reach an index $j' = \Psi\big(\ldots\Psi\big(\Psi(j)\big)\ldots\big)$ such that $B[j'] = 1$ (i.e., $\mathsf{PSA}[j']$ is explicitly stored). If the Ψ operation was applied k times, we get $\mathsf{PSA}[j] = \mathsf{PSA}[j'] - k$. The time complexity is $O(k)$. For PSA^{-1} queries, we store $\mathsf{PSA}^{-1}[i]$ if i equals n or if i is a multiple of Δ. To compute $\mathsf{PSA}^{-1}[j]$, we first find the largest number $j' \leq j$, such that j' is a multiple of Δ. Compute $j'' = \mathsf{PSA}^{-1}[j']$ from the sampled-PSA^{-1} in $O(1)$ time. Let $k = j - j' < \Delta$. Starting from j'' carry out k successive Ψ operations and report the final index as $\mathsf{PSA}^{-1}[j]$ in $O(k)$ time. Finally, $k < \Delta = \lceil \log_\sigma n \rceil$. The (extra) space needed is $(n/\Delta) \log n = O(n \log \sigma)$ bits. Other trade-offs may be obtained by tuning Δ, which is what we will do using a more sophisticated sampling technique along with a modified version of Theorem 5; details are in Section 4.

2.1 Succinct Data-Structure Toolkit

▶ **Fact 6** ([31]). *A tree having m nodes can be stored in $2m + o(m)$ bits, such that if each node is labeled by its pre-order rank, the following operations can be supported in $O(1)$ time:*

- $\mathsf{pre\text{-}order}(u)/\mathsf{post\text{-}order}(u)$ = *pre-order/post-order rank of node u.*
- $\mathsf{parent}(u)$ = *parent of node u.*
- $\mathsf{nodeDepth}(u)$ = *number of edges on the path from the root to u.*
- $\mathsf{lca}(u, v)$ = *lowest common ancestor (LCA) of two nodes u and v.*
- $\mathsf{lmostLeaf}(u)/\mathsf{rmostLeaf}(u)$ = *leftmost/rightmost leaf in the subtree rooted at u.*
- $\mathsf{levelAncestor}(u, D)$ = *ancestor of u such that* $\mathsf{nodeDepth}(u) = D$.

Also, we can find the pre-order rank of the ith leftmost leaf in $O(1)$ time.

▶ **Fact 7** ([2]). *Given an array $A[1, t]$ over $\Sigma = \{1, 2, \ldots, \sigma\}$, by storing an $O(t \log \sigma)$-bit structure, we can support the following operation in $O(1 + \log \frac{\log \sigma}{\log t})$ time:*

$\mathsf{rank}_A(i, c) =$ *number of occurrences of c in $A[1, i]$*

Additionally, the following operations can be supported in $O(1)$ time:

- *access $A[i]$*
- $\mathsf{partialRank}_A(i) = \mathsf{rank}(i, A[i])$, *i.e., the number of occurrences of $A[i]$ is the range $[1, i]$*
- $\mathsf{select}_A(i, c)$ = *the i^{th} occurrence of c in A*

2.2 Proof of Theorem 5

From their definitions, we observe: $\Psi(i) = j \iff \mathsf{PLF}(j) = i$, and $\mathsf{W}[i] = \mathsf{PBWT}[\Psi(i)]$. Additionally, we use $\chi(i)$ to denote the number of suffixes k such that $\Psi(k) \le \Psi(i)$ and $\mathsf{W}[k] = \mathsf{W}[i]$. Based on these, it is easy to see that if $j = \Psi(i)$, then j is the $\chi(i)^{th}$ occurrence of $\mathsf{W}[i]$ in PBWT. Given $\chi(i)$, we can compute $\Psi(i)$ as:

$$\Psi(i) = \mathsf{select}_{\mathsf{PBWT}}\Big(\chi(i), \mathsf{W}[i]\Big)$$

Note that arrays $\mathsf{PBWT}[1, n]$ and $\mathsf{W}[1, n]$ take $O(\log \sigma)$ bits per entry. Therefore, we preprocess them into compact data structures that support access/select queries in $O(1)$ time (see Fact 7). Therefore, given $\chi(i)$, we can compute $\Psi(i)$ in constant time. To this end, we present the following lemma, which completes the proof of Theorem 5.

▶ **Lemma 8.** *By using an $O(n \log \sigma)$-bit data structure, we can compute $\chi(i)$ in $O(1)$ time.*

The rest of the section is dedicated to proving Lemma 8. For brevity, throughout we use "suffix i" to denote the suffix corresponding to leaf ℓ_i in the PST.

▶ **Lemma 9.** *Let $i < n$. Suppose u is the highest node on the path from the root to ℓ_i such that $\mathsf{f}_i \le \mathsf{strLen}(u) - 1$. Then, for any leaf ℓ_j in the subtree of u, we have $\mathsf{f}_j = \mathsf{f}_i$*

Proof. Let $d = \mathsf{plcp}(i, j)$. Since $d \ge \mathsf{strLen}(u) \ge \mathsf{f}_i + 1$, we have $\mathsf{prev}\big(T_{\mathsf{PSA}[i]}[1, d]\big) = \mathsf{prev}\big(T_{\mathsf{PSA}[j]}[1, d]\big)$, i.e., the suffixes starting at $\mathsf{PSA}[i]$ and $\mathsf{PSA}[j]$ p-match until their first d characters. Clearly, the first occurrence of $T[\mathsf{PSA}[i]]$ in $T_{1+\mathsf{PSA}[i]}$ must be the same as the first occurrence of $T[\mathsf{PSA}[j]]$ in $T_{1+\mathsf{PSA}[j]}$, i.e., $\mathsf{f}_i = \mathsf{f}_j$. ◀

▶ **Lemma 10.** *If $\mathsf{W}[i] = \mathsf{W}[j]$, then $\Psi(j) < \Psi(i)$ iff*
- **Case A:** *either $\mathsf{f}_j = \mathsf{plcp}(i, j)$*
- **Case B:** *or, $\mathsf{f}_j \ne \mathsf{plcp}(i, j)$, $j < i$, and $\mathsf{f}_i \ne \mathsf{plcp}(i, j)$*

Proof. Recall Convention 1. Let $d = \mathsf{plcp}(i, j)$. If $\mathsf{f}_i = \varnothing$ or $\mathsf{f}_j = \varnothing$, then $\mathsf{W}[i] \ne \mathsf{W}[j]$. So, $\mathsf{f}_i, \mathsf{f}_j < n$. Also, $\mathsf{prev}(T_{\mathsf{PSA}[i]})[d + 1] \ne \mathsf{prev}(T_{\mathsf{PSA}[j]})[d + 1]$, by the definition of LCP. We now prove both cases (see Figure 2 for an illustration).

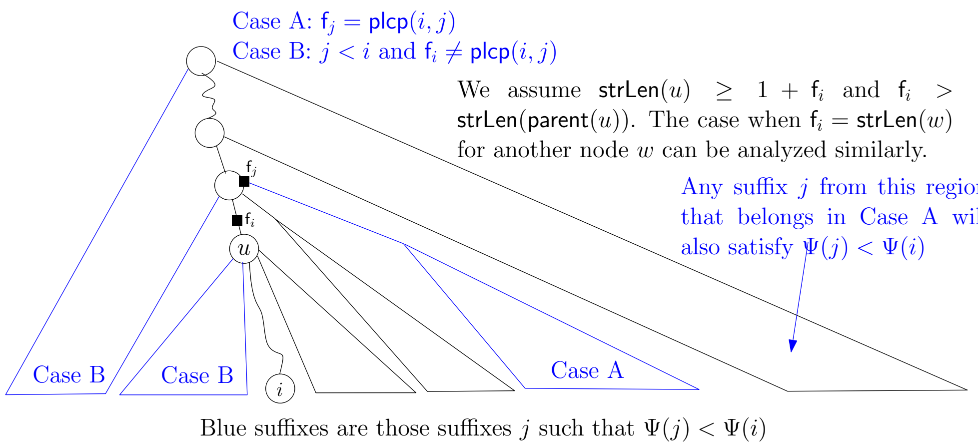


Figure 2 Illustration of Lemma 10.

- If $\mathsf{f}_j = d$, then $\mathsf{prev}(T_{\mathsf{PSA}[j]})[d+1] = d$ and $\mathsf{prev}(T_{\mathsf{PSA}[\Psi(j)]})[d] = 0$. From Lemma 9, we conclude $\mathsf{f}_i \geq d$. Moreover, $\mathsf{f}_i \neq d$ because $\mathsf{prev}(T_{\mathsf{PSA}[i]})[d+1] \neq d$ (by the definition of LCP). Therefore, $\mathsf{f}_i > d$. This implies $\mathsf{prev}(T_{\mathsf{PSA}[i]})[d+1] \neq 0$; otherwise, $\mathsf{W}[i] \neq \mathsf{W}[j]$, a contradiction. Consequently, either $\mathsf{prev}(T_{\mathsf{PSA}[\Psi(i)]})[d] > 0$ or $\mathsf{prev}(T_{\mathsf{PSA}[\Psi(i)]})[d] = \$$. Finally, note that $\mathsf{plcp}\big(\Psi(i), \Psi(j)\big) \geq d-1$. So after removing the first character of the two suffixes, their first $(d-1)$ characters will p-match. Hence, $\Psi(j) < \Psi(i)$ when $\mathsf{f}_j = \mathsf{plcp}(i,j)$.
- Now, assume $\mathsf{f}_j \neq d$. If $\mathsf{f}_j < d$, then $\mathsf{f}_i = \mathsf{f}_j$ (from Lemma 9) and $\mathsf{plcp}\big(\Psi(i), \Psi(j)\big) = d-1$. Then, $\Psi(j) < \Psi(i)$ iff $j < i$ because $\mathsf{prev}(T_{\mathsf{PSA}[\Psi(i)]})[d] = \mathsf{prev}(T_{\mathsf{PSA}[i]})[d+1]$ and $\mathsf{prev}(T_{\mathsf{PSA}[\Psi(j)]})[d] = \mathsf{prev}(T_{\mathsf{PSA}[j]})[d+1]$. If $\mathsf{f}_j > d$, then $\mathsf{prev}(T_{\mathsf{PSA}[j]})[d+1] = \mathsf{prev}(T_{\mathsf{PSA}[\Psi(j)]})[d]$. Also, $\mathsf{f}_i \geq d$ (from Lemma 9), implying either $\mathsf{prev}(T_{\mathsf{PSA}[\Psi(i)]})[d] = \mathsf{prev}(T_{\mathsf{PSA}[i]})[d+1]$ or $\mathsf{prev}(T_{\mathsf{PSA}[\Psi(i)]})[d] = 0$. The latter happens only when $\mathsf{f}_i = \mathsf{plcp}(i,j)$. Hence, we have $\Psi(j) < \Psi(i)$ when $j < i$ and $\mathsf{f}_i \neq \mathsf{plcp}(i,j)$.

This concludes the proof. ◀

To compute $\chi(i)$, we count the number n_A and n_B of Case A and B suffixes respectively; note that the cases are disjoint. Then, $\chi(i) = 1 + n_A + n_B$. We provide an overview first.

First, we locate the edge on which f_i lies, i.e., locate the edge $(\mathsf{parent}(u), u)$, such that $\mathsf{strLen}(\mathsf{parent}(u)) < 1 + \mathsf{f}_i \leq \mathsf{strLen}(u)$. This is facilitated by associating a bit with each node and set it to `1` iff $\mathsf{strLen}(\mathsf{parent}(\cdot)) < 1 + \mathsf{f}_j \leq \mathsf{strLen}(\cdot)$ for some suffix j in its sub-tree. Therefore, u is the lowest ancestor of ℓ_i that is associated with `1`. By maintaining an $O(n)$ bit structure, we can answer this query in $O(1)$ time (see Lemma 11).

For counting n_A, we walk the path from root to $\mathsf{parent}(u)$, and for each node x on this path find out the number of suffixes j satisfying Case A: $\mathsf{f}_j = \mathsf{strLen}(x)$. Note that we afford to store f_j explicitly, instead store if f_j lies on an edge from node x to its child node y. Luckily, that's enough for us – for any suffix j, if $\mathsf{f}_j = \mathsf{strLen}(x)$ lies on the edge (x, y), then for all suffixes j' in the subtree of y, we have $f_{j'} = \mathsf{strLen}(x)$ (by Lemma 9). So, the count can be obtained via a simple unary encoding of suffixes of this kind. For counting Case B: $j < i$ and $\mathsf{f}_i \neq \mathsf{plcp}(i,j)$, walk from root to a node v; here, $v = u$ if $\mathsf{f}_i > \mathsf{strLen}(u)$, and $v = \mathsf{parent}(u)$ if $\mathsf{f}_i = \mathsf{strLen}(u)$. Initialize n_B to the number of leaves that lie to the left of this path. Then, add the number of leaves lying to the left of ℓ_i within the sub-tree of v to n_B. Note that for Cases A and B, we must consider only the suffixes j satisfying $\mathsf{W}[j] = \mathsf{W}[i]$; this is achieved by collecting suffixes based on their $\mathsf{W}[\cdot]$ values into different trees. Finally, we cannot afford to walk the path; therefore, we rely on the result in Lemma 12.

▶ **Lemma 11.** *Consider a compacted tree τ having L leaves, where each node is associated with a 0 or 1. By using an $O(L)$-bit data structure, given a query leaf ℓ, we can find the lowest ancestor v (if it exists) of ℓ associated with a 1 in $O(1)$ time.*

▶ **Lemma 12.** *Consider a compacted tree τ having L leaves, where each node w is associated with an integer $g(w) \geq 0$. For any node v, we have $\sum_{u \in \mathcal{S}_v} g(u) \leq L_v$, where $\mathcal{S}_v$ is the set of nodes in the subtree of v and L_v is the number of leaves in the subtree of v. By using an $O(L)$-bit data structure, given a query leaf ℓ, we can compute $G(\ell) = \sum_v g(v)$ in $O(1)$ time, where v is an ancestor of ℓ.*

The proofs of Lemmas 11 and 12 (deferred to Section 2.3) relies on mostly standard techniques from succinct data structures. Next we present the implementation details of $\chi(i)$ computation.

The Data Structure. To compute $\chi(i)$, we only need to consider those suffixes k, such that $\mathsf{W}[k] = \mathsf{W}[i]$. To this end, we create (at most) σ compacted tries $\mathsf{PST}_1, \mathsf{PST}_2, \ldots, \mathsf{PST}_\sigma$, where PST_α is the compacted trie of the strings in $\{\mathsf{prev}(T[\mathsf{PSA}[k], n]) \mid \mathsf{W}[k] = \alpha\}$. We do not store these trees explicitly; rather, we store their topology with succinct functionalities using Fact 6.

We pre-process each PST_α using Lemma 11 as follows: associate each node w in PST_α with $\mathtt{1}$ if $\mathsf{strLen}(\mathsf{parent}(w)) \le \mathsf{f}_i < \mathsf{strLen}(w)$ for some leaf ℓ_i under w with $\mathsf{W}[i] = \alpha$. We also pre-process PST_α using Lemma 12 as follows: associate a node w with the number β_w, where β_w is the number of suffixes i in the subtree of w in PST_α such that $\mathsf{f}_i = \mathsf{strLen}(w)$. Note that $\sum_w \beta_w$ over all nodes w in all the trees is $O(n_\alpha)$, where n_α is the number of leaves in PST_α. The space needed for all such PST_α trees combined is $O(n)$ bits. Finally, we store a partial-rank data structure (Fact 7) on W. The total space needed is $O(n \log \sigma)$ bits.

Query Processing. Given i, in $O(1)$ time, we first jump to the corresponding leaf $\ell_{i'}$ in $\mathsf{PST}_{\mathsf{W}[i]}$ by using the $\mathsf{partialRank}_{\mathsf{W}}(i)$ query. Now locate the highest node u in $\mathsf{PST}_{\mathsf{W}[i]}$ such that $1 + \mathsf{f}_i \le \mathsf{strLen}(u)$ in $O(1)$ time using Lemma 11. We consider the following two scenarios separately. To determine which case a suffix falls in, we store a bit-vector $F[1, n]$, such that $F[i] = 1$ iff the suffix i belongs to the first case. In each of the following cases, we can compute $\chi(i)$ in $O(1)$ time, which completes the proof of Lemma 8.

Case 1: $\mathsf{strLen}(\mathsf{parent}(u)) < \mathsf{f}_i < \mathsf{strLen}(u)$ for an ancestor node u of ℓ_i.
Let j be such that $\mathsf{W}[j] = \mathsf{W}[i]$. Applying Lemma 10, $\Psi(j) < \Psi(i)$ if either $j < i$ or $\mathsf{f}_j = \mathsf{plcp}(i, j)$. Thus, $\chi(i) = i' + \sum_v \beta_v$, where v is an ancestor of $\ell_{i'}$. The last term can be computed in $O(1)$ time using Lemma 12.

Case 2: $\mathsf{f}_i = \mathsf{strLen}(u)$ for an ancestor node u of ℓ_i.
Let j be such that $\mathsf{W}[j] = \mathsf{W}[i]$. Applying Lemma 10, $\Psi(j) < \Psi(i)$ if either (1) $j < i$ and $\mathsf{plcp}(i, j) \ne \mathsf{f}_i$, or (2) $\mathsf{f}_j = \mathsf{plcp}(i, j)$. Let w be the child of u on the path to $\ell_{i'}$. Using Fact 6, in $O(1)$ time, we compute $\mathsf{lmostLeaf}(u)$ and $\mathsf{lmostLeaf}(w)$, which are respectively the leftmost leaf in the subtree of u and the subtree of w. Thus,

$$\chi(i) = i' - (\mathsf{lmostLeaf}(w) - \mathsf{lmostLeaf}(u)) + \sum_v \beta_v$$

where v is an ancestor of $\ell_{i'}$. The last term can be computed in $O(1)$ time using Lemma 12. This completes the proof of Lemma 8.

2.3 Proofs of Lemma 11 and Lemma 12

We rely on standard techniques from succinct data structures. For both lemmas, we employ the following marking scheme. Starting from the leftmost leaf, every $C = c\lceil \log L \rceil$ leaves form a group, where c is a constant to be decided later. (The last group may have fewer than C leaves.) Mark the LCA of the first and last leaf of each group. The number of marked nodes is $O(L/C)$ [22].

We prove Lemma 11 first. At each marked node, we store the depth of its nearest ancestor (including itself) which is associated with a 1. Traverse the subtree τ_{u^*} of a marked node u^* in pre-order, and create a bit-string B_{u^*} as follows: when entering the subtree of a node w, append 1 if w is associated with a 1, followed by a 0 to B_{u^*}. Additionally, for every $i \in [1, L_{u^*}]$, store $A_{u^*}[i] =$ node depth of the nearest ancestor of $\ell_{u^*,i}$ associated with a 1 if the ancestor is in τ_{u^*}, else $A_{u^*}[i] = -1$. For each marked node u^*, maintain a pointer to the corresponding A_{u^*} and B_{u^*} pair. Pre-process τ_{u^*} with Fact 6. Lastly, we maintain a bit-vector to detect in $O(1)$ time whether a leaf has an ancestor associated with 1, or not. The total space as before can be bounded by $O(L)$ bits. Given a query leaf ℓ_k, we check whether ℓ_k has an ancestor associated with 1 or not. Assume that it does. Then, we first locate the lowest marked node v^* as described earlier. Let d^* be the node depth stored at v^*. Let $k' = k - C\lfloor k/C \rfloor$. Check the k'th entry of the satellite array of v^*, and let it be d'. If $d' >= 0$, then the desired node is given by $\mathsf{levelAncestor}(\ell, d')$, else the desired node is given by $\mathsf{levelAncestor}(\ell, d^*)$.

Now, we prove Lemma 12. At each marked node u^*, store $G(u^*)$. Since the number of marked nodes is at most $\lceil L/C \rceil$, the space needed is $O(\frac{L}{C}\log L) = O(L)$ bits. Let τ_{u^*} be the subtree rooted at a marked node u^*. Note that τ_{u^*} has at most $2C$ nodes. Traverse tree τ_{u^*} in pre-order, and create a bit-string B_{u^*} as follows: when entering the subtree of a node w, append $g(w)$ in unaryto B_{u^*}. Additionally, for every $i \in [1, L_{u^*}]$, store $A_{u^*}[i] = G(\ell_{u^*,i})$, where L_{u^*} is the number of leaves in the subtree of u^*, and $\ell_{u^*,i}$ is the i^{th}-leftmost leaf in τ_{u^*}. The space needed to store the array A_{u^*} is $O(C\log C)$ bits. Note that $|B_{u^*}| \le 2C$; hence, the number of possible such bit-strings is at most 2^{2C}. We store all possible combinations of A_{u^*} and B_{u^*}, which requires $O(2^{2C}C\log C)$ bits, which is $o(L)$ bits for $c = 1/4$. For each marked node u^*, maintain a pointer to the corresponding A_{u^*} and B_{u^*} pair, which requires $\frac{L}{C}\log(2^{2C}) = O(L)$ bits. Finally, pre-process τ_{u^*} with Fact 6. The total space needed is $O(L)$ bits. Given a query leaf ℓ_k, we first locate the lowest marked node $v^* = \mathsf{lca}(\ell_x, \ell_y)$ of ℓ_k, where $x = 1 + C\lfloor k/C \rfloor$, $y = \min\{L, C(1 + \lfloor k/C \rfloor)\}$. Let d^* be the value stored at v^*. Let $k' = k - C\lfloor k/C \rfloor$. Check the k'th entry of the satellite array of v^*, and let it be d'. Then, $G(\ell_k) = d^* + d'$ is computed in $O(1)$ time.

3 Generalized Ψ Function

We start with a definition that we are going to use throughout this section, as well as a couple of lemmas that will form the backbone of the indexes to achieve the three trade-offs.

▶ **Definition 13.** *Define* $\Psi^k(i) = \mathsf{PSA}^{-1}[\mathsf{PSA}[i] + k]$.

Our main arsenal to obtain the three trade-offs is the following version of Theorem 5, which enables the computation of "some" $\Psi^k(\cdot)$ in time faster than $O(k)$.

▶ **Lemma 14.** *For any predefined integer Δ, we can construct an $O(n\log\sigma)$-bit structure $\mathsf{DS}(\Delta)$ that computes $\Psi^\Delta(i)$ for any i with $\mathsf{PSA}[i]$ being a multiple of Δ in $O(1)$ time.*

We prove this lemma in this section. Let's start with the intuition. Note that in Lemma 14, if one is willing to relax the time to $O(\Delta)$, we can simply apply Theorem 5 Δ times. Here, we will chop off the first Δ characters of a suffix, where the characters are from Σ. To reduce the time to $O(1)$, the main idea is to consider a character from the alphabet Σ^Δ; clearly, chopping off one character from Σ^Δ is equivalent to chopping off Δ characters from Σ. Note that each character from Σ^Δ requires $\Delta\log\sigma$ bits for representation; however, since we sample $\approx n/\Delta$ suffixes, the total space will still be $O((n/\Delta)\cdot\Delta\log\sigma) = O(n\log\sigma)$ bits. The proof techniques are similar to that of Theorem 5.

▶ **Definition 15.** *A suffix is Δ-sampled if its starting position is a multiple of Δ. Let S_Δ be the collection of all Δ-sampled suffixes of T, i.e., $S_\Delta = \{T_\Delta, T_{2\Delta}, \dots\}$. Let $n_\Delta = |S_\Delta|$ be the number of Δ-sampled suffixes. A Δ-sampled parameterized suffix array, denoted as $\mathsf{PSA}_\Delta[1, n_\Delta]$, stores the starting position of the suffixes in lexicographic order of their* prev-*encoding.*

▶ **Definition 16.** *Let $B_\Delta[1,n]$ be a bitmap, such that $B_\Delta[i]=1$ iff $\mathsf{PSA}[i]$ is a multiple of Δ.*

▶ **Lemma 17.** *Given $i \in [1,n_\Delta]$, we can find a $j \in [1,n]$, such that $\mathsf{PSA}[j]=\mathsf{PSA}_\Delta[i]$ in $O(1)$ time by maintaining an $O(n)$-bit space structure.*

Proof. We maintain a bit-vector $B_\Delta[1,n]$ using Fact 7, where $B_\Delta[i]=1$ iff $T_{\mathsf{PSA}[i]}$ is Δ-sampled. The total space needed is $O(n)$ bits. Observe that $T_{\mathsf{PSA}_\Delta[i]}$ is exactly the i^{th} Δ-sampled suffix in lexicographic order; thus, $j=\mathsf{select}_{B_\Delta}(i,1)$. ◀

▶ **Definition 18.** *Let $T_{\mathsf{PSA}[i]}$ be a Δ-sampled suffix. Define the following:*

Notation	***Definition ($\circ$ denotes concatenation)***
$\Psi^\Delta(i)$	$\mathsf{PSA}^{-1}[\mathsf{PSA}[i]+\Delta]$
$\mathsf{PLF}^\Delta(i)$	$\mathsf{PSA}^{-1}[\mathsf{PSA}[i]-\Delta]$
$\mathsf{W}^\Delta[i]$	$\mathsf{W}[i]\circ\mathsf{W}[\Psi(i)]\circ\cdots\circ\mathsf{W}[\Psi^{\Delta-1}(i)]$
$\mathsf{PBWT}^\Delta[i]$	$\mathsf{PBWT}[\mathsf{PLF}^{\Delta-1}(i)]\circ\mathsf{PBWT}[\mathsf{PLF}^{\Delta-2}(i)]\circ\cdots\circ\mathsf{PBWT}[i]$

▶ **Observation 19.** *Let $T_{\mathsf{PSA}[i]}$ be a Δ-sampled suffix. The following observations can be deduced from the above definitions:*

- $\Psi^\Delta(i)=j$ *iff* $\mathsf{PLF}^\Delta(j)=i$
- *If* $\Psi^\Delta(i)=j$, *then* $\mathsf{PBWT}^\Delta[i]=\mathsf{W}^\Delta[j]$

▶ **Definition 20.** *Let $T_{\mathsf{PSA}[i]}$ be a Δ-sampled suffix. Define*

$$\chi^\Delta(i)=|\{k,\ \textit{where } T_{\mathsf{PSA}[k]} \textit{ is } \Delta\textit{-sampled}, \Psi^\Delta(k)\le\Psi^\Delta(i) \textit{ and } \mathsf{W}^\Delta[k]=\mathsf{W}^\Delta[i]\}|$$

Reduction from function Ψ^Δ to χ^Δ. Using Observation 19, it is easy to see that if $j=\Psi^\Delta(i)$, then j is the $\chi^\Delta(i)^{th}$ occurrence of $\mathsf{W}^\Delta[i]$ in PBWT^Δ. Given $\chi^\Delta(i)$, we can compute $\Psi^\Delta(i)$ as: $\Psi^\Delta(i)=\mathsf{select}_{B_\Delta}\left(\mathsf{select}_{\mathsf{PBWT}^\Delta}(\chi^\Delta(i),\mathsf{W}^\Delta[i]),1\right)$

Note that the arrays $\mathsf{PBWT}^\Delta[1,n_\Delta]$ and $\mathsf{W}^\Delta[1,n_\Delta]$ take $O(\Delta\log\sigma)$ bit per entry. We preprocess them into compact data structures that support access/select queries in $O(1)$ time (see Fact 7). The space needed is $O(n_\Delta\cdot\Delta\log\sigma)=O(n\log\sigma)$ bits. Thus, given $\chi^\Delta(i)$, we can compute $\Psi^\Delta(i)$ in constant time. To this end, we present the following lemma,which completes the proof of Lemma 14.

▶ **Lemma 21.** *Let $T_{\mathsf{PSA}[i]}$ be a Δ-sampled suffix. By using an $O(n\log\sigma)$-bit data structure, we can compute $\chi^\Delta(i)$ in $O(1)$ time.*

3.1 Proof of Lemma 21

For the ease of notation, let $i_d=\Psi^d(i)$, $j_d=\Psi^d(j)$, and $L_d=\mathsf{plcp}(i_d,j_d)$, where $d\in[1,\Delta]$. Let $i_0=i$, $j_0=j$, and $L_0=\mathsf{plcp}(i,j)$. Our proof hinges on Lemma 22, which says, for any two suffixes $T_{\mathsf{PSA}[i]}$ and $T_{\mathsf{PSA}[j]}$ such that $i<j$ and $\mathsf{W}^\Delta[i]=\mathsf{W}^\Delta[j]$, the relative order between i and j can change at most once while applying the Ψ-operation Δ number of times.

▶ **Lemma 22.** *Consider two Δ-sampled suffixes $T_{\mathsf{PSA}[i]}$ and $T_{\mathsf{PSA}[j]}$ such that $i<j$, and $\mathsf{W}^\Delta[i]=\mathsf{W}^\Delta[j]$.*

- *If there exists a $\gamma\in[1,\Delta-1]$ such that $i_\gamma>j_\gamma$, then $\forall\gamma'\in[\gamma+1,\Delta]$, $i_{\gamma'}>j_{\gamma'}$.*
- *Consider the minimum $\gamma\in[1,\Delta]$ such that $i_\gamma>j_\gamma$, then $\mathsf{f}_{j_{\gamma-1}}=L_{\gamma-1}$.*

Proof. We prove the first part of the lemma; the second part follows directly. Consider the smallest $\gamma \in [1, \Delta - 1]$ such that $\Psi^{\gamma}(i) > \Psi^{\gamma}(j)$. Since $\Psi^{\gamma-1}(i) < \Psi^{\gamma-1}(j)$, we have $\mathsf{str}(\ell_{j_{\gamma-1}})[1 + L_{\gamma-1}] > \mathsf{str}(\ell_{i_{\gamma-1}})[1 + L_{\gamma-1}]$. Then, $\mathsf{str}(\ell_{j_\gamma})[1 + L_\gamma] = 0 < \mathsf{str}(\ell_{i_\gamma})[1 + L_\gamma]$; otherwise, applying Lemma 9, it is easy to show that $\mathsf{W}^{\Delta}[i] \neq \mathsf{W}^{\Delta}[j]$. For the purpose of contradiction, consider the smallest $\gamma' > \gamma$ such that $\Psi^{\gamma'}(i) < \Psi^{\gamma'}(j)$. Note that $L_{\gamma'-1} = L_\gamma - (\gamma' - \gamma - 1)$. Hence, $\mathsf{str}(\ell_{j_{\gamma'-1}})[1 + L_{\gamma'-1}] = 0 < \mathsf{str}(\ell_{i_{\gamma'-1}})[1 + L_{\gamma'-1}]$. Since $\Psi^{\gamma'}(i) < \Psi^{\gamma'}(j)$, applying Lemma 9, $\mathsf{str}(\ell_{i_{\gamma'}})[1 + L_{\gamma'}] = 0$ and $\mathsf{f}_{i_{\gamma'}} = L_{\gamma'}$, which contradicts $\mathsf{W}^{\Delta}[i] = \mathsf{W}^{\Delta}[j]$. This completes the proof. ◀

Using Lemmas 9, 10, and 22, we get the following.

▶ **Lemma 23.** *Consider two Δ-sampled suffixes $T_{\mathsf{PSA}[i]}$ and $T_{\mathsf{PSA}[j]}$ such that $i < j$, and $\mathsf{W}^{\Delta}[i] = \mathsf{W}^{\Delta}[j]$. If there exists a $\gamma \in [1, \Delta]$ such that $i_\gamma > j_\gamma$, then*

- *$i_\Delta > j_\Delta$, and*
- *for any k such that $T_{\mathsf{PSA}[k]}$ is Δ-sampled, $\mathsf{W}^{\Delta}[i] = \mathsf{W}^{\Delta}[k]$, and $\mathsf{plcp}(k, j) > \mathsf{plcp}(i, j)$, we have $i_\Delta > k_\Delta$.*

To compute $\chi^{\Delta}(i)$, note that we only need to consider those suffixes k, such that $\mathsf{W}^{\Delta}[k] = \mathsf{W}^{\Delta}[i]$. To this end, we create (at most) σ^{Δ} compact tries $\mathsf{PST}^{\Delta}_1, \mathsf{PST}^{\Delta}_2, \ldots, \mathsf{PST}^{\Delta}_{\sigma^{\Delta}}$, where PST_α is the compacted trie of the strings in

$$\{\mathsf{prev}(T[\mathsf{PSA}[k], n]) \mid \mathsf{W}^{\Delta}[k] = \alpha \text{ and } T_{\mathsf{PSA}[k]} \text{ is } \Delta \text{ sampled}\}$$

We do not store these trees explicitly; rather, we maintain the data-structure of Fact 6 for each tree topology. Note that given a leaf k in PST, where $T_{\mathsf{PSA}[k]}$ is Δ sampled, we can jump to its corresponding leaf k' in $\mathsf{PST}_{\mathsf{W}^{\Delta}}$ in $O(1)$ time using an $O(n \log \sigma)$ structure (the bit-array B_Δ, and Fact 7 over W^{Δ}).

Consider a tree PST^{Δ}_x. Let the number of suffixes lying in this tree be m_x. For any leaf j' in PST^{Δ}_x, let $\mathsf{map}(j')$ be the equivalent leaf in PST. Consider a node u in PST^{Δ}_x. For each $\gamma \in [1, \Delta]$ we write two numbers $G_\gamma(u)$ and $H_\gamma(u)$ defined as:

- $G_\gamma(u)$ = the number of leaves j' in the subtree of u such that $\mathsf{f}_{j_\gamma} = \mathsf{strLen}(u)$, where $j = \mathsf{map}(j')$
- If $\mathsf{f}_{j_\gamma} \neq \mathsf{strLen}(\mathsf{parent}(u))$, where j' is a leaf in the subtree of u and $j = \mathsf{map}(j')$, then $H_\gamma(u) = 0$. Else, $H_\gamma(u)$ = the number of leaves k' in the subtree of $\mathsf{parent}(u)$ such that $\mathsf{f}_{k_\gamma} \neq \mathsf{strLen}(\mathsf{parent}(u))$ and $\mathsf{pre\text{-}order}(\ell_{k'}) < \mathsf{pre\text{-}order}(u)$, where $k = \mathsf{map}(k')$

Note that $\sum_u G_\gamma(u) \leq m_x$ and $\sum_u H_\gamma(u) \leq m_x$ (using Lemmas 22 and 23). Hence, $\sum_u G_\gamma(u)$ and $\sum_u H_\gamma(u)$ over $\gamma \in [1, \Delta]$ can be stored in $O(m_x \Delta)$ bits using unary encoding. To compute $\chi^{\Delta}(i)$, we first jump to the corresponding leaf i' in $\mathsf{PST}^{\Delta}_{\mathsf{W}^{\Delta}[i]}$ in $O(1)$ time. Let $\mathcal{U}$ be the set of ancestors of $\ell_{i'}$ Now, $\chi(i) = i' + \sum_{\gamma=1}^{\Delta} \sum_{u \in \mathcal{U}} G_u(\ell_{i'}) - \sum_{\gamma=1}^{\Delta} \sum_{u \in \mathcal{U}} H_u(\ell_{i'})$, which can be computed in $O(1)$ time using (slightly adapted versions of) Lemmas 11 and 12. Since $\sum_{x=1}^{\sigma^{\Delta}} m_x = n_\Delta$, the total space is $O(n \log \sigma)$ bits; recall that m_x is the number of suffixes in PST^{Δ}_x. This concludes the proof.

4 Achieving the Three Trade-offs of PSA

We prove the trade-offs using the result in Lemma 14 as a black box. Additionally, we will use the sampled PSA and sampled PSA^{-1} in Lemma 24 for all the three cases. Let $\lambda = 2^{\lceil \log \log_\sigma n \rceil}$, the next highest power of 2 greater than or equal to $\log_\sigma n$. The strategy for computing $\mathsf{PSA}[i]$ is the same as before (refer to Section 2), i.e., find the smallest $k < \lambda$, such that $\mathsf{PSA}[i] = \mathsf{PSA}[j] - k$, where $j = \Psi^k(i)$ and $B_\lambda[j] = 1$, but in fewer number of steps.

▶ **Lemma 24** (Sampled PSA and PSA^{-1}). *A sampled-*PSA *is a structure that supports the following query: for any i, it reports* $\mathsf{PSA}[i]$ *if* $\mathsf{PSA}[i]$ *is a multiple of λ, and ∞ otherwise. Similarly, a sampled-*PSA^{-1} *is a structure that computes* $\mathsf{PSA}^{-1}[i]$ *for any i which is a multiple of λ. We can maintain them in $O(n \log \sigma)$ bits and answer queries in $O(1)$ time.*

Proof. Note that the sampled-PSA^{-1} is an array of size $O(n/\lambda)$ in which each entry can be recorded using $\lceil \log n \rceil$ bits and accessed in $O(1)$ time. For the sampled-PSA, we maintain the bitmap $B_\lambda[1, n]$ in Definition 16 by choosing $\Delta = \lambda$. Additionally, we associate $\mathsf{PSA}[i]$ with those i's where $B_\lambda[i] = 1$. The space required is $(n + (n/\lambda) \log n) = O(n \log \sigma)$ bits, and the query can be easily handled in $O(1)$ time. ◀

4.1 Achieving $t_{\mathsf{PSA}} = O(\log_\sigma^\varepsilon n)$ using $O(n \log \sigma)$ bits

Let Δ_t be $(\log_\sigma^\varepsilon n)^t$ rounded to the next highest power of 2. We maintain $\mathsf{DS}(\Delta_t)$ and $B_{\Delta_t}[1, n]$ for $t = 0, 1, 2, 3, \ldots, 1/\varepsilon$. (Recall Lemma 14 and Definition 16 for definitions of this data structures.) The space is $1/\varepsilon \times O(n \log \sigma)$ bits, as desired.

To compute $\mathsf{PSA}[i]$, we initialize $k = 0, j = i, t = 0$ and follow the steps below.

1. If $B_\lambda[j] = 1$, access $\mathsf{PSA}[j]$ in $O(1)$ time from the sampled-PSA and report $\mathsf{PSA}[j] - k$.
2. Else if $B_{\Delta_{t+1}}[j] = 1$, update $t \leftarrow t + 1$ and go to Step 1.
3. Else we compute $j' = \Psi^{\Delta_t}(j)$ using $\mathsf{DS}(\Delta_t)$ in $O(1)$ time, update $j \leftarrow j'$, $k \leftarrow k + \Delta_t$, and then we repeat from Step 2.

Then, the number of times we perform (constant time) $\Psi^{\Delta_t}(\cdot)$ operations on $\mathsf{DS}(\Delta_t)$ is at most $\Delta_{t+1}/\Delta_t = \log_\sigma^\varepsilon n$. Therefore, the overall time complexity is $O(\frac{1}{\varepsilon} \log_\sigma^\varepsilon n)$.

Note that the algorithm for $\mathsf{PSA}[i]$ computes several j's, starting with $j = i$, such that the j computed after t^{th} "step 1" guarantees that $\mathsf{PSA}[j] \geq \mathsf{PSA}[i]$ is the smallest number divisible by Δ_t. Therefore, the correctness follows from that fact that $\mathsf{PSA}[i]$ is $\mathsf{PSA}[j] - k$.

The computation of $\mathsf{PSA}^{-1}[\cdot]$ is analogous, but in the reverse order, as desired. Specifically, we perform queries on $\mathsf{DS}(\Delta_t)$'s, in descending order of t.

4.2 Achieving $t_{\mathsf{PSA}} = O(\log \log_\sigma n)$ using $O(n \log \sigma \log \log_\sigma n)$ bits

We maintain $\mathsf{DS}(\Delta_t)$ structure of Lemma 14 and $B_{\Delta_t}[1, n]$, where $\Delta_t = 2^t$, for $t = 0, 1, 2, 3, \ldots, \log \lambda$, where $\lambda = 2^{\lceil \log \log_\sigma n \rceil}$ as defined in Lemma 24. The space is $O(n \log \sigma) \times \log \lambda$ bits, as desired.
To compute $\mathsf{PSA}[i]$, we initialize $k = 0, j = i, t = 0$ and follow the steps below.

1. If $B_\lambda[j] = 1$, access $\mathsf{PSA}[j]$ in $O(1)$ time from the sampled-PSA and report $\mathsf{PSA}[j] - k$.
2. Else if $B_{\Delta_{t+1}}[j] = 1$, update $t \leftarrow t + 1$ and go to Step 1.
3. Else compute $j' = \Psi^{\Delta_t}(j)$ using $\mathsf{DS}(\Delta_t)$ in $O(1)$ time, update $j \leftarrow j'$ and $k \leftarrow k + \Delta_t$. Then update $t \leftarrow t + 1$ and go to Step 1.

We perform at most $\log \lambda$ constant-time operations on $\mathsf{DS}(\cdot)$, hence $t_{\mathsf{PSA}} = O(\log \log_\sigma n)$. The computation of $\mathsf{PSA}^{-1}[\cdot]$ (and correctness proof) is analogous as in the previous section.

4.3 Achieving $t_{\mathsf{PSA}} = O(1)$ using $O(n \log \sigma \log_\sigma^\varepsilon n)$ bits

Here we use Lemma 25, which is a slight modification of Lemma 14. We remark that the proof is rather straightforward given the proof of Lemma 14; so, we omit it.

▶ **Lemma 25.** *For any predefined integers Δ and $\delta < \Delta$ (both are powers of 2), we can construct an $O(n \log \sigma)$-bit structure that computes $\Psi^{\delta}(i)$ for any i with $(\mathsf{PSA}[i] + \delta)$ being a multiple of δ in $O(1)$ time. We call this data structure $\mathsf{DS}(\Delta, \delta)$.*

We define an array $E_{\Delta}^{\delta}[1, n]$ such that $E_{\Delta}^{\delta}[i]$ is

- $-\infty$ if $\mathsf{PSA}[i]$ is not a multiple of Δ
- an integer $f \in [0, \Delta/\delta)$, such that $(\mathsf{PSA}[i] + f \cdot \delta)$ is a multiple of Δ.

Let Δ_t be $(\log_{\sigma}^{\varepsilon} n)^t$ rounded to the next highest power of 2. We store $\mathsf{DS}(\Delta_{t+1}, f \cdot \Delta_t)$ for all $t \in [0, 1/\varepsilon]$ and $f \in [0, \Delta_{t+1}/\Delta_t)$. Additionally, we store $E_{\Delta_{t+1}}^{\Delta_t}[1, n]$ for all $t \in [0, 1/\varepsilon]$. Therefore, the total space is $n/\varepsilon \times O\big(\log \sigma \log_{\sigma}^{\varepsilon} n + \log(\log_{\sigma}^{\varepsilon} n)\big)$ bits.
To compute $\mathsf{PSA}[i]$, we initialize $k = 0, j = i, t = 0$ and follow the steps below.

1. If $B_{\lambda}[j] = 1$, access $\mathsf{PSA}[j]$ in $O(1)$ time from the sampled-PSA and report $\mathsf{PSA}[j] - k$.
2. Else if $B_{\Delta_{t+1}}[j] = 1$, update $t \leftarrow t + 1$ and go to Step 1.
3. Else find f from $E_{\Delta_{t+1}}^{\Delta_t}[1, n]$, such that $(\mathsf{PSA}[i] + f \cdot \Delta_t)$ is a multiple of Δ_{t+1}. Compute $j' = \Psi^{f \cdot \Delta_t}(j)$ using $\mathsf{DS}(\Delta_{t+1}, f \cdot \Delta_t)$ in $O(1)$ time, update $j \leftarrow j'$ and $k \leftarrow k + f \cdot \Delta_t$. Then go to Step 2.

We issue at most one (constant time) query on $\mathsf{DS}(\Delta_t, \cdot)$ per t. Therefore, $t_{\mathsf{PSA}} = O(1/\varepsilon)$. The computation of $\mathsf{PSA}^{-1}[\cdot]$ (and correctness proof) is analogous to the discussion in the previous two sections.

5 Encoding Parameterized Longest Common Prefix (pLCP) Array

Recall that $\mathsf{PLCP}[i] = \mathsf{plcp}(i, i+1)$ for $1 \leq i < n$. We introduce a new encoding scheme, which converts a string S to a string $\mathsf{encode}(S)$ over an alphabet $\Sigma'' = \{0, 1, \ldots, \sigma\}$ as follows. We replace each character $S[i]$ with 0 if i is the first occurrence of $S[i]$, else replace it with the number of distinct characters in $S[j, i]$, where $j < i$ is the rightmost occurrence of $S[i]$ before i. For example, $\mathsf{encode}(xyxxzyx) = 0021033$. For any two strings S and S', $\mathsf{encode}(S) = \mathsf{encode}(S')$ iff $\mathsf{prev}(S) = \mathsf{prev}(S')$; the proof is straightforward using mathematical induction.

Let $T' = \mathsf{encode}(T)$. Let $\mathsf{SA}_{T'}[1, n]$ be the suffix array of T', i.e., $\mathsf{SA}_{T'}[i] = j$ and $\mathsf{SA}_{T'}^{-1}[j] = i$ iff the i^{th} lexicographically smallest suffix of T' starts at position j. Also, let $\mathsf{lcp}_{T'}(i, j)$ be the length of the longest common prefix of the suffixes of T' starting at $\mathsf{SA}_{T'}[i]$ and $\mathsf{SA}_{T'}[j]$. The following is immediate from known results on encoding suffix trees.

▶ **Fact 26** ([20, 35]). *We can answer $\mathsf{SA}_{T'}[\cdot]$ and $\mathsf{SA}_{T'}^{-1}[\cdot]$ queries as follows:*

- *in $\mathsf{t_{SA}} = O(\log_{\sigma}^{\varepsilon} n)$ time using an $O(n \log \sigma)$-bit index*
- *in $\mathsf{t_{SA}} = O(\log \log_{\sigma} n)$ time using an $O(n \log \sigma \log \log_{\sigma} n)$-bit index*
- *in $\mathsf{t_{SA}} = O(1)$ time using an $O(n \log \sigma \log_{\sigma}^{\varepsilon} n)$-bit index*

Moreover, we can answer $\mathsf{lcp}_{T'}(\cdot, \cdot)$ queries in $O(t_{\mathsf{t_{SA}}})$ time using $O(n)$ extra bits.

We have the following crucial lemma.

▶ **Lemma 27.** *Let x_i be the smallest number such that the number of distinct characters in $T_{\mathsf{PSA}[i]}[1, \mathsf{PLCP}[i]]$ and $T_{\mathsf{PSA}[i]}[1, x_i]$ are the same. Then,*

$$\mathsf{PLCP}[i] = x_i + \mathsf{lcp}_{T'}\Big(\mathsf{SA}_{T'}^{-1}\big[\mathsf{PSA}[i] + x_i\big], \mathsf{SA}_{T'}^{-1}\big[\mathsf{PSA}[i+1] + x_i\big]\Big)$$

Proof. Since $\mathsf{PLCP}[i] \geq x_i$, $y_i = \mathsf{PLCP}[i] - x_i$ is the length of the longest common prefix of the strings obtained by deleting the first x_i characters of $\mathsf{prev}(T_{\mathsf{PSA}[i]})$ and $\mathsf{prev}(T_{\mathsf{PSA}[i+1]})$ respectively. Equivalently, y_i is the longest common prefix of the suffixes of $\mathsf{encode}(T)$ starting at positions $\mathsf{PSA}[i] + x_i$ and $\mathsf{PSA}[i+1] + x_i$ respectively. The proof follows from the definition of x_i. ◀

▶ **Theorem 28.** *Suppose* $\mathsf{PSA}[\cdot]$ *and* $\mathsf{SA}^{-1}_{T'}[\cdot]$ *values are accessible in times* $\mathsf{t_{PSA}}$ *and* $\mathsf{t_{SA}}$ *respectively, we can compute* $\mathsf{PLCP}[i] = x_i + y_i$ *for any* i *in time* $O(\mathsf{t_{SA}} + \mathsf{t_{PSA}})$ *using an* $O(n \log \sigma)$*-bit structure. We can also support* $\mathsf{plcp}(\cdot, \cdot)$ *queries in the same time.*

Proof. We first describe the structure for computing x_i. If $\sigma > \log n$, we store x_i explicitly in $\log n$ bits if $x_i > \sigma \log n$ and in $O(\log(\sigma \log n))$ bits otherwise. All x_i's that are larger than $\sigma \log n$ can be stored in $O(n)$ bits as they are no more than $n/\log n$. The space needed for the rest is $n \log(\sigma \log n) = O(n \log \sigma)$ bits. If $\sigma \leq \log n$, maintain an array C, where $C[i] = T_{\mathsf{PSA}[i]}[x_i]$ and a rank-select data structure (Fact 7) over T. The space is $O(n \log \sigma)$ bits. Since x_i is the first occurrence of $C[i]$ in $T[\mathsf{PSA}[i], n]$, we compute $x_i = \mathsf{select}_T\big(\mathsf{rank}_T\big(\mathsf{PSA}[i] - 1, C[i]\big) + 1, C[i]\big) - \mathsf{PSA}[i] + 1$ in time $t_{\mathsf{PSA}} + O(\log(\log \sigma / \log \log n)) = O(t_{\mathsf{PSA}})$.

We now focus on computing y_i. Find $j = \mathsf{SA}^{-1}_{T'}[\mathsf{PSA}[i] + x_i]$ and $k = \mathsf{SA}^{-1}_{T'}[\mathsf{PSA}[i+1] + x_i]$ in time $O(\mathsf{t_{SA}} + \mathsf{t_{PSA}})$. From Lemma 27, we have $y_i = \mathsf{lcp}_{T'}(j, k)$. We handle $\mathsf{lcp}_{T'}(\cdot, \cdot)$ queries in time $O(t_{\mathsf{SA}})$ using $O(n)$ extra bits [35].

Finally, to answer $\mathsf{plcp}(\cdot, \cdot)$ queries, we maintain a Range Minimum Query (RMQ) structure [8] of size $2n + o(n)$ over the PLCP array with $O(1)$ query time. Then, given any i and $j > i$, compute $k = \arg\min\{\mathsf{PLCP}[k] \mid k \in [i, j)\}$ and report $\mathsf{plcp}(i, j) = \mathsf{PLCP}[k]$. ◀

Theorem 3 follows from Theorem 28, Fact 26, and the trade-offs in Section 4.

References

1 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.

2 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015. doi:10.1145/2629339.

3 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.

4 Gianni Decaroli, Travis Gagie, and Giovanni Manzini. A compact index for order-preserving pattern matching. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4-7, 2017*, pages 72–81. IEEE, 2017. doi:10.1109/DCC.2017.35.

5 Gianni Decaroli, Travis Gagie, and Giovanni Manzini. A compact index for order-preserving pattern matching. *Softw. Pract. Exp.*, 49(6):1041–1051, 2019. doi:10.1002/spe.2694.

6 Diptarama, Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, and Ayumi Shinohara. Position heaps for parameterized strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPIcs*, pages 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.8.

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

8 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

9 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. The parameterized suffix tray. In Tiziana Calamoneri and Federico Corò, editors, *Algorithms and Complexity - 12th International Conference, CIAC 2021, Virtual Event, May 10-12, 2021, Proceedings*, volume 12701 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 2021. doi:10.1007/978-3-030-75242-2_18.

10 Travis Gagie, Giovanni Manzini, and Rossano Venturini. An encoding for order-preserving matching. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 38:1–38:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.38.

11 Arnab Ganguly, Wing-Kai Hon, Yu-An Huang, Solon P. Pissis, Rahul Shah, and Sharma V. Thankachan. Parameterized text indexing with one wildcard. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2019, Snowbird, UT, USA, March 26-29, 2019*, pages 152–161. IEEE, 2019. doi:10.1109/DCC.2019.00023.

12 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. Space-efficient dictionaries for parameterized and order-preserving pattern matching. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 2:1–2:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.2.

13 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. A framework for designing space-efficient dictionaries for parameterized and order-preserving matching. *Theor. Comput. Sci.*, 854:52–62, 2021. doi:10.1016/j.tcs.2020.11.036.

14 Arnab Ganguly, Wing-Kai Hon, and Rahul Shah. A framework for dynamic parameterized dictionary matching. In Rasmus Pagh, editor, *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22-24, 2016, Reykjavik, Iceland*, volume 53 of *LIPIcs*, pages 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.SWAT.2016.10.

15 Arnab Ganguly, Dhrumil Patel, Rahul Shah, and Sharma V. Thankachan. LF successor: Compact space indexing for order-isomorphic pattern matching. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 71:1–71:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.71.

16 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pbwt: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.

17 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural pattern matching - succinctly. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation, ISAAC 2017, December 9-12, 2017, Phuket, Thailand*, volume 92 of *LIPIcs*, pages 35:1–35:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ISAAC.2017.35.

18 Raffaele Giancarlo. The suffix of a square matrix, with applications. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.*, pages 402–411, 1993. URL: http://dl.acm.org/citation.cfm?id=313559.313842.

19 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`.

20 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. `doi:10.1137/S0097539702402354`.

21 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997. `doi:10.1017/cbo9780511574931`.

22 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`.

23 Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theor. Comput. Sci.*, 302(1-3):223–238, 2003. `doi:10.1016/S0304-3975(02)00828-9`.

24 Sung-Hwan Kim and Hwan-Gue Cho. Indexing isodirectional pointer sequences. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 35:1–35:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.ISAAC.2020.35`.

25 Sung-Hwan Kim and Hwan-Gue Cho. A compact index for cartesian tree matching. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.CPM.2021.18`.

26 Sung-Hwan Kim and Hwan-Gue Cho. Simpler fm-index for parameterized string matching. *Inf. Process. Lett.*, 165:106026, 2021. `doi:10.1016/j.ipl.2020.106026`.

27 Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, 29(13):1149–1171, 1999. `doi:10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O`.

28 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22:935–948, 1993. `doi:10.1137/0222058`.

29 Katsuhito Nakashima, Noriki Fujisato, Diptarama Hendrian, Yuto Nakashima, Ryo Yoshinaka, Shunsuke Inenaga, Hideo Bannai, Ayumi Shinohara, and Masayuki Takeda. Dawgs for parameterized matching: Online construction and related indexing structures. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 26:1–26:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.CPM.2020.26`.

30 Gonzalo Navarro. *Compact data structures: A practical approach.* Cambridge University Press, 2016.

31 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. `doi:10.1145/2601073`.

32 Sung Gwan Park, Amihood Amir, Gad M. Landau, and Kunsoo Park. Cartesian tree matching and indexing. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.CPM.2019.16`.

33 Dhrumil Patel and Rahul Shah. Inverse suffix array queries for 2-dimensional pattern matching in near-compact space. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPIcs*, pages 60:1–60:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ISAAC.2021.60`.

34 Mihai Patrascu. Lower bounds for 2-dimensional range counting. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 40–46. ACM, 2007. `doi:10.1145/1250790.1250797`.

35 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.

36 Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. In *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, pages 393–406, 2000. doi:10.1007/3-540-44985-X_34.

37 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.