Deterministic Indexing for Packed Strings

Philip Bille*[1], Inge Li Gørtz†[2], and Frederik Rye Skjoldjensen‡[3]

1 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
phbi@dtu.dk
2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
inge@dtu.dk
3 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
fskj@dtu.dk

Abstract

Given a string S of length n, the classic string indexing problem is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. In the *deterministic* variant the goal is to solve the string indexing problem without any randomization (at preprocessing time or query time). In the *packed* variant the strings are stored with several character in a single word, giving us the opportunity to read multiple characters simultaneously. Our main result is a new string index in the deterministic *and* packed setting. Given a packed string S of length n over an alphabet σ, we show how to preprocess S in $O(n)$ (deterministic) time and space $O(n)$ such that given a packed pattern string of length m we can support queries in (deterministic) time $O\left(m/\alpha + \log m + \log\log\sigma\right)$, where $\alpha = w/\log\sigma$ is the number of characters packed in a word of size $w = \Theta(\log n)$. Our query time is always at least as good as the previous best known bounds and whenever several characters are packed in a word, i.e., $\log\sigma \ll w$, the query times are faster.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, H.3.1 Content Analysis and Indexing

Keywords and phrases suffix tree, suffix array, deterministic algorithm, word packing

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.6

1 Introduction

Let S be a string of length n over an alphabet of size σ. The *string indexing problem* is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. Typical queries include *existential queries* (decide if the pattern occurs in S), *reporting queries* (return all positions where the pattern occurs), and *counting queries* (returning the number of occurrences of the pattern).

The string indexing problem is a classic well-studied problem in combinatorial pattern matching and the standard textbook solutions are the suffix tree and the suffix array (see e.g., [9, 10, 11, 14]). A straightforward implementation of suffix trees leads to an $O(n)$ preprocessing time and space solution that given a pattern of length m supports existential and counting queries in time $O(m\log\sigma)$ and reporting queries in time $O(m\log\sigma + \mathrm{occ})$, where occ is the number of occurrences of the pattern. The suffix array implemented with additional arrays storing longest common prefixes leads to a solution that also uses $O(n)$ preprocessing time and space while supporting existential and counting queries in time

* Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
† Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
‡ Supported by the Danish Research Council (DFF – 1323-00178).

© Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen;
licensed under Creative Commons License CC-BY
28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017).
Editors: Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter; Article No. 6; pp. 6:1–6:11

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

$O(m + \log n)$ and reporting queries in time $O(m + \log n + \text{occ})$. If we instead combine suffix trees with perfect hashing [7] we obtain $O(n)$ *expected* preprocessing time and $O(n)$ space, while supporting existential and counting queries in time $O(m)$ and reporting queries in time $O(m + \text{occ})$. The above bounds hold assuming that the alphabet size σ is polynomial in n. If this is not the case, additional time for sorting the alphabet is required [5]. For simplicity, we adopt this convention in all of the bounds throughout the paper.

In the *deterministic* variant the goal is to solve the string indexing problem without any randomization. In particular, we cannot combine suffix trees with perfect hashing to obtain $O(m)$ or $O(m + \text{occ})$ query times. In this setting Cole et al. [4] showed how to combine the suffix tree and suffix array into the *suffix tray* that uses $O(n)$ preprocessing time and space and supports existential and counting queries in $O(m + \log \sigma)$ time and reporting queries in $O(m + \log \sigma + \text{occ})$ time. Recently, the query times were improved by Fischer and Gawrychowski [6] to $O(m + \log\log \sigma)$ and $O(m + \log\log \sigma + \text{occ})$, respectively.

In the *packed* variant the strings are given in a *packed representation*, with several characters in a single word [3, 2, 1, 13]. For instance, DNA-sequences have an alphabet of size 4 and are therefore typically stored using 2 bits per character with 32 characters in a 64-bit word. On packed strings we can read multiple characters in constant time and hence potentially do better than the immediate $\Omega(m)$ or $\Omega(m + \text{occ})$ lower bound for existential/counting queries and reporting queries, respectively. In this setting Takagi et al. [13] recently introduced the *packed compact trie* that stores packed strings succinctly and also supports dynamic insertion and deletions of strings. In a static and deterministic setting their data structure implies a linear space and superlinear time preprocessing solution that uses $O(\frac{m}{\alpha} \log\log n)$ and $O(\frac{m}{\alpha} \log\log n + \text{occ})$ query time, respectively.

In this paper, we consider the string indexing problem in the deterministic and packed setting simultaneously, and present a solution that improves all of the above bounds.

1.1 Setup and result

We assume a standard unit-cost word RAM with word length $w = \Theta(\log n)$, and a standard instruction set including arithmetic operations, bitwise boolean operations, and shifts. All strings in this paper are over an alphabet Σ of size σ. The *packed representation* of a string A is obtained by storing $\alpha = w/\log \sigma$ characters per word thus representing A in $O(|A| \log \sigma / w)$ words. If A is given in the packed representation we simply say that A is a *packed string*.

Throughout the paper let S be a string of length n. Our goal is to preprocess S into a compact data structure that given a packed pattern string P supports the following queries.

- Count(P): Return the number of occurrence of P in S.
- Locate(P): Report all occurrences of P in S.
- Predecessor(P): Returns the predecessor of P in S, i.e., the lexicographically largest suffix in S that is smaller than P.

We show the following main result.

▶ **Theorem 1.** *Let S be a string of length n over an alphabet of size σ and let $\alpha = w/\log \sigma$ be the number of characters packed in a word. Given S we can build an index in $O(n)$ deterministic time and space such that given a packed pattern string of length m we can support Count and Predecessor in time $O(\frac{m}{\alpha} + \log m + \log\log \sigma)$ and Locate in time $O(\frac{m}{\alpha} + \log m + \log\log \sigma + \text{occ})$ time.*

Compared to the result of Fischer and Gawrychowski [6], Thm 1 is always at least as good and whenever several characters are packed in a word, i.e., $\log \sigma \ll w$, the query times are faster. Compared to the result of Takagi et al. [13], our query time is a factor $\log\log n$ faster.

Technically, our results are obtained by a novel combination of previous techniques. Our general tree decomposition closely follows Fischer and Gawrychowski [6], but different ideas are needed to handle packed strings efficiently. We also show how to extend the classic suffix array search algorithm to handle packed strings efficiently.

2 Preliminaries

2.1 Deterministic hashing and predecessor

We use the following results on deterministic hashing and predecessor data structures.

▶ **Lemma 2** (Ružić [12, Theorem 3]). *A static linear space dictionary on a set of k keys can be deterministically constructed in time $O(k(\log\log k)^2)$, so that lookups to the dictionary take time $O(1)$.*

Fischer and Gawrychowski [6] use the same result for hashing characters. In our context we will apply it for hashing words of packed characters.

▶ **Lemma 3** (Fischer and Gawrychowski [6, Proposition 7]). *A static linear space predecessor data structure on a set of k keys from a universe of size u can be constructed deterministically in $O(k)$ time and $O(k)$ space such that predecessor queries can be answered deterministically in time $O(\log\log u)$.*

2.2 Suffix tree

The suffix tree T_S of S is the compacted trie over the n suffixes from the string S. We assume that the special character $\$ \notin \Sigma$ is appended to every suffix of S such that each string is ending in a leaf of the tree. The edges are sorted lexicographically from left to right. We say that a leaf *represents* the suffix that is spelled out by concatenating the labels of the edges on the path from the root to the leaf. For a node v in T_S, we say that the *subtree* of v is the tree induced by v and all proper descendants of v. We distinguish between implicit and explicit nodes: implicit nodes are conceptual and refer to the original non branching nodes from the trie without compacted paths. Explicit nodes are the branching nodes in the original trie. When we refer to nodes that are not specified as either explicit or implicit, then we are always referring to explicit nodes. The lexicographic ordering of the suffixes represented by the leaves corresponds to the ordering of the leaves from left to right in the compacted trie. For navigating from node to child, each node has a predecessor data structure over the first characters of every edge going to a child. With the predecessor data structure from Lemma 3 navigation from node to child takes $O(\log\log\sigma)$ time and both the space and the construction time of the predecessor data structure is linear in the number of children.

2.3 Suffix array

Let $S_1, S_2, \ldots, S_n$ be the n suffixes of S from left to right. The suffix array SA_S of S gives the lexicographic ordering of the suffixes such that $S_{\mathsf{SA}_S[i]}$ refers to the ith lexicographically largest suffix of S. This means that for every $1 < i \leq n$ we have that $S_{\mathsf{SA}_S[i-1]}$ is lexicographically smaller than $S_{\mathsf{SA}_S[i]}$. For simplicity we let $\mathsf{SA}_S[i]$ refer to the suffix $S_{\mathsf{SA}_S[i]}$ and we say that $\mathsf{SA}_S[i]$ represents the suffix $S_{\mathsf{SA}_S[i]}$. Every suffix from S with pattern P as a prefix will be located in a consecutive range of SA_S. This range corresponds to the range of consecutive leaves in the subtree spanned by the explicit or implicit node that represents P in T_S. We can find the range of SA_S where P prefixes every suffix by performing binary search twice

over SA_S. A naïve binary search takes $O(m \log n)$ time: We maintain the boundaries, L and R, of the current search interval and in each iteration we compare the median string from the range L to R in SA_S, with P, and update L and R accordingly. This can be improved to $O(m + \log n)$ time if we have access to additional arrays storing the value of the longest common prefixes between a selection of strings from SA_S. We construct the suffix array from the suffix tree in $O(n)$ time.

3 Deterministic index for packed strings

In this section we describe how to construct and query our deterministic index for packed strings. This structure is the basis for our result in Thm 1. For short patterns where $m < \log_\sigma n - 1$ we store tabulated data that enables us to answer queries fast. We construct the tables in $O(n)$ time and space and answer queries in $O(\log\log\sigma + \text{occ})$ time. For long patterns where $m \geq \log_\sigma n - 1$ we use a combination of a suffix tree and a suffix array that we construct in $O(n)$ time and space such that queries take $O(m/\alpha + \log\log n + \text{occ})$ time. For $m \geq \log_\sigma n - 1$ we have that $\log\log n = \log(\frac{\log n}{\log\sigma}\log\sigma) = \log\log_\sigma n + \log\log\sigma \leq \log(\log_\sigma n - 1) + 1 + \log\log\sigma \leq \log m + 1 + \log\log\sigma$. This gives us a query time of $O(m/\alpha + \log m + \log\log\sigma + \text{occ})$ for the deterministic packed index. We need the following connections between T_S and SA_S: For each explicit node t in T_S we store a reference to the range of SA_S that corresponds to the leaves spanned by the subtree of t and for each index in SA_S we store a reference to the corresponding leaf in T_S that represents the same string.

We first describe our word accelerated algorithm for matching patterns in SA_S that we need for answering queries on long patterns. Then we describe how to build and use the data structures for answering queries on short and long patterns.

3.1 Packed matching in SA_S

We now show how to word accelerate the suffix array matching algorithm by Manber and Myers [10]. They spend $O(m)$ time reading P but by reading α characters in constant time we can reduce this to $O(m/\alpha)$. We let $\text{LCP}(i,j)$ denote the length of the longest common prefix between the suffixes $\mathsf{SA}_S[i]$ and $\mathsf{SA}_S[j]$ and obtain the result in Lemma 4.

▶ **Lemma 4.** *Given the suffix array SA_S over the packed string S and a data structure for answering the relevant* LCP *queries, we can find the lexicographic predecessor of a packed pattern P of length m in SA_S in $O(m/\alpha + \log n)$ time where α is the number of characters we can pack in a word.*

In the algorithm by Manber and Myers we maintain the left and right boundaries of the current search interval of SA_S denoted by L and R and the length of the longest common prefix between $\mathsf{SA}_S[L]$ and P, and between $\mathsf{SA}_S[R]$ and P, that we denote by l and r, respectively. Initially the search interval is the whole range of SA_S such that $L = 1$ and $R = n$. In an iteration we do as follows: If $l = r$ we start comparing $\mathsf{SA}_S[M]$ with P from index $l + 1$ until we find a mismatch and update either L and l, or R and r, depending on whether $\mathsf{SA}_S[M]$ is lexicographically larger or smaller than P. Otherwise, when $l \neq r$, we perform an LCP query that enable us to either halve the range of SA_S without reading from P or start comparing $\mathsf{SA}_S[M]$ with P from index $l+1$ as in the $l = r$ case. When $l > r$ there are three cases: If $\text{LCP}(L, M) > l$ then P is lexicographically larger than $\mathsf{SA}_S[M]$ and we set L to M and continue with the next iteration. If $LCP(L, M) < l$ then P is lexicographically smaller than $\mathsf{SA}_S[M]$ and we set R to M and set r to $\text{LCP}(L, M)$ and continue with the next iteration. If $LCP(L, M) = l$ then we compare $\mathsf{SA}_S[M]$ and P from index $l + 1$ until

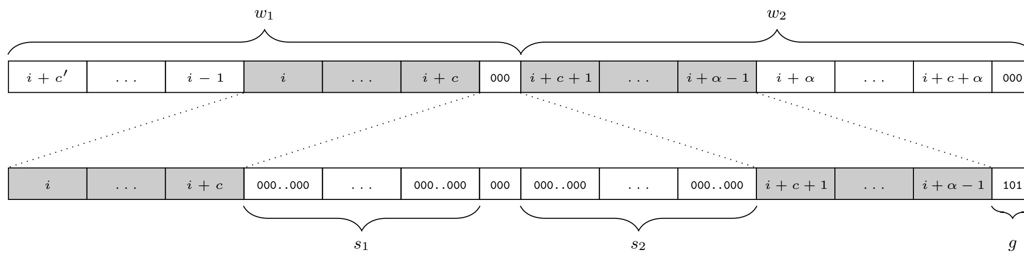

Figure 1 Alignment of α characters that extends over a word boundary where $c' = c + 1 - \alpha$. The relevant part of the lower word w_1 and upper word w_2 is combined with bitwise shifts, a bitwise or and the g bits on the right is set to 0.

we find a mismatch. Let that mismatch be at index $l + i$. If the mismatch means that P is lexicographically smaller than $\mathsf{SA}_S[M]$ then we set R to M and set r to $l + i - 1$ and continue with the next iteration. If the mismatch means that P is lexicographically larger than $\mathsf{SA}_S[M]$ then we set L to M and set l to $l + i - 1$ and continue with the next iteration. Three symmetrical cases exists when $r > l$.

We generalize their algorithm to work on word packed strings such that we can compare α characters in constant time. In each iteration where we need to read from P we align the next α characters from P and $\mathsf{SA}_S[M]$ such that we can compare them in constant time: Assume that we need to read the range from i to $i + \alpha - 1$ in P. If this range of characters is contained in one word we do not need to align. Otherwise, we extract the relevant parts of the words that contain the range with bitwise shifts and combine them in w_{align} with a bitwise or. See Figure 1. We align the α characters from $\mathsf{SA}_S[M]$ in the same way and store them in w'_{align}.

We use a *bitwise exclusive or* operation between w_{align} and w'_{align} to construct a word where the most significant set bit is at a bit position that belong to the mismatching character with the lowest index. We obtain the position of the most significant set bit in constant time with the technique of Fredman and Willard [8]. From this we know exactly how many of the next α characters that match and we can increase i accordingly. Since every mismatch encountered result in a halving of the search range of SA_S we can never read more than $O(\log n)$ incomplete chunks. The number of complete chunks we read is bounded by $O(m/\alpha)$. Overall we obtain a $O(m/\alpha + \log n)$ time algorithm for matching in SA_S. This result is summarized in Lemma 4.

3.2 Handling short patterns

Now we show how to answer count, locate and lexicographic predecessor queries on short patterns. We store an array containing an index for every possible pattern P where $m < \log_\sigma n - 1$ and at the index we store a pointer to the deepest node in T_S that prefixes P. We call this node d_P. We use d_P as the basis for answering every query on short patterns. We assume that the range in SA_S spanned by d_P goes from l to r. We answer predecessor queries as follows: If P is lexicographically smaller than $\mathsf{SA}_S[0]$ then P has no predecessor in SA_S. Otherwise, we find the predecessor as follows: If d_P represents P then the predecessor of P is located at index $l - 1$ of SA_S. Otherwise, we assume that d_P prefixes P with i characters and need to decide whether P continues on an edge out of d_P or P deviates from T_S in d_P. We do this by querying the predecessor data structure over the children of d_P with the character at position $i + 1$ of P. If this query does not return an edge, then $P[i + 1]$ is lexicographically

smaller than the first character of every edge out of d_P, and the predecessor of P is the string located at index $l-1$ of SA_S. If this query returns an edge e_{pred} then there are two cases.

Case 1: The first character of e_{pred} is not identical to $P[i+1]$. Then the predecessor of P is the lexicographically largest string in the subtree under e_{pred}.

Case 2: The first character on e_{pred} is identical to $P[i+1]$. In this case, if there exists an edge e'_{pred} out of d_P on the left side of e_{pred}, then the predecessor of P is the lexicographically largest string in the subtree under e'_{pred} and otherwise the predecessor is the string at index $l-1$ of SA_S.

We report the node in T_S that represents the predecessor of P.

We let e_{pred} be defined as above and answer count queries as follows: If d_P represents P we return the number of leaves spanned by d_P in T_S. If P instead continues and ends on e_{pred} we report the number of leaves spanned by the subtree below e_{pred}. We answer locate queries in the same way but instead of reporting the range we report the strings in the range.

We find d_P in $O(1)$ time and e_{pred} in $O(\log\log\sigma)$ time. In total we answer predecessor and count queries in $O(\log\log\sigma)$ time and locate queries in $O(\log\log\sigma + \mathrm{occ})$ time

Since $m < \log_\sigma n - 1$ there exists $\sigma + \sigma^2 + \ldots + \sigma^{\lfloor \log_\sigma n-1 \rfloor} \leq \sigma^{\lfloor \log_\sigma n \rfloor} \leq \sigma^{\log_\sigma n} = n$ short patterns and we compute them in $O(n)$ time by performing a preorder traversal of T_S bounded to depth $\log_\sigma n - 1$. Let d_P be the node we are currently visiting and let d_{next} be the node we visit next. When we visit d_P we fill the tabulation array for every string that is lexicographically larger than or equal to the string represented by d_P and lexicographically smaller than the string represented by d_{next}. Every short string can be stored in a word of memory and therefore we can index the tabulation array with the numerical value of the word that represent the string. We fill each of these indices with a pointer to d_P since d_P is the deepest node in T_S that represents a string that prefixes these strings. We can store the tabulation array in $O(n)$ space.

3.3 Handling long patterns

Now we show how to answer count, locate and lexicographic predecessor queries on long patterns. We first give an overview of our solution followed by a detailed description of the individual parts. In T_S we distinguish between *light* and *heavy* nodes. If a subtree under a node spans at least $\log^2\log n$ leaves, we call the node heavy, otherwise we call it light. A node is a heavy branching node if it has at least two heavy children and all the heavy nodes constitute a subtree that we call the heavy tree. We decompose the heavy tree into micro trees of height α and we augment every micro tree with a data structure that enables navigation from root to leaf in constant time. For micro trees containing a heavy branching node we do this with deterministic hashing and for micro trees without a heavy branching node we just compare the relevant part of P with the one unique path of the heavy tree that goes through the micro tree. To avoid navigating the light nodes we in each light node store a pointer to the range of SA_S that the node spans. We construct two predecessor data structures for each micro tree: The *light predecessor* structure over the strings represented by the light nodes that are connected to the heavy nodes in the micro tree and the *heavy predecessor* structure over the heavy nodes in the micro tree. We answer queries on P as follows: We traverse the heavy tree in chunks of α characters until we are unable to traverse a complete micro tree. This means that P either continues in a light node, ends in the micro tree or deviates from T_S in the micro tree. We can decide if P continues in a light node with the light predecessor structure and if this is the case we answer the query with the packed matching algorithm on the range of SA_S spanned by the light node. Otherwise, we use the heavy predecessor structure for finding d_P in the micro tree and use d_P for answering the

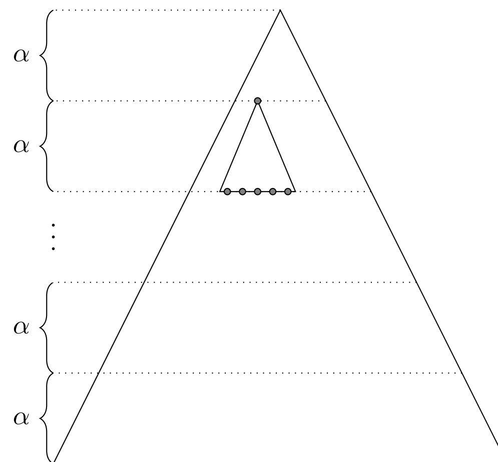

Figure 2 The decomposition of HT_S in micro trees of height α. One micro tree is shown with the root at string depth α and the boundary nodes at string depth 2α.

query as in section 3.2. The following sections describes in more detail how we build our data structure and answer queries.

3.3.1 Data structure

This section describes our data structure in details. If a subtree under a node in T_S spans at least $\log^2 \log n$ leaves, we call the node heavy. The heavy tree HT_S is the induced subgraph of all the the heavy nodes in T_S. We decompose HT_S into *micro trees* of string depth α. This decomposition into micro trees of height α was also employed by Takagi et al. [13]. A node, explicit or implicit, is a boundary node if its string depth is a multiple of α. Except for the original root and leaves of HT_S, each boundary node belongs to two micro trees i.e., a boundary node at depth $d\alpha$ is the root in a micro tree that starts at string depth $d\alpha$ and is a leaf in a micro tree that starts at string depth $(d-1)\alpha$. Figure 2 shows the decomposition of HT_S into micro trees of string depth α.

We augment every micro tree with information that enables us to navigate from root to leaf in constant time. To avoid using too much space we promote only some of the implicit boundary nodes to explicit nodes. We distinguish between three kinds of micro trees:

- **Type 1.** At least one heavy branching node exists in the micro tree: We promote the root and leaves to explicit nodes and use deterministic hashing to navigate the micro tree from root to leaf. Because the micro tree is of height α, each of the strings represented by the leaves in the micro tree fits in a word and can be used as a key for hashing. We say that the root is a hashing node and the leaves are hashed nodes. We will postpone the analysis of time and space used by the micro trees that use hashing for navigation.
- **Type 2.** No heavy branching node exists in the micro tree: When the micro tree does not contain a heavy branching node, the micro tree is simply a path from root to leaf. Here we distinguish between two cases:
  - **Type 2a.** The micro tree contains an explicit non branching heavy node: We promote the root and leaf to explicit nodes. Navigating from root to leaf takes constant time by comparing the string represented by the leaf with the appropriate part of P. We charge the space increase from the promotion of the root and leaf to the explicit non branching heavy node. Since there are at most n explicit non branching heavy nodes we never promote more than $2n$ implicit nodes from type 2a micro trees.

- **Type 2b.** The micro tree does not contain an explicit heavy node: Let t be a micro tree with no explicit heavy nodes. If the root of t is a leaf in a micro tree that contains an explicit heavy node, we promote the root of t to an explicit node and store a pointer to the root of the nearest micro tree below t that contains an explicit heavy node. The path from root to root corresponds to a substring in S that we navigate by comparing this string to the appropriate part of P. We charge the space increase from the promotion of the root to the heavy node descendant. Since we have at most n explicit heavy nodes we promote no more than n implicit nodes from type 2b micro trees. If the root of t is a leaf in a micro tree without an explicit heavy node we do not promote the root of t.

We say that a node in T_S is a heavy leaf if it is a heavy node with no heavy children. We want to bound the number of heavy branching nodes and heavy leaves. Every heavy leaf spans at least $\log^2 \log n$ leaves of T_S. This means we can have at most $n/\log^2 \log n$ heavy leaves in T_S. Since we have at most one branching heavy node per heavy leaf the number of heavy branching nodes is at most $n/\log^2 \log n$.

We want to bound the number of implicit nodes that are promoted to explicit hashed nodes. This number is critical for constructing all hash functions in $O(n)$ time. We bound the number of promoted hashed nodes by associating each with the nearest descendant that is either a heavy branching node or a heavy leaf: Let l be a promoted hashed node in a micro tree that contain a heavy branching node h. Then every promoted hashed node above l is associated with h or a node above h in the tree. Hence, no other promoted node can be associated with the first encountered heavy branching or leaf node below l. Since we have at most $O(n/\log^2 \log n)$ heavy branching and heavy leaf nodes we also have at most $O(n/\log^2 \log n)$ implicit nodes that are promoted to explicit hashed nodes.

With deterministic hashing from Lemma 2 the total time for constructing the explicit hashing nodes are given as follows. Here H is the set of all the hash functions and we bound the elements in every hash function h to $n/\log^2 \log n$.

$$O\left(\sum_{h\in H} |h| \log^2 \log |h|\right) = O\left(\sum_{h\in H} |h| \log^2 \log(n/\log^2 \log n)\right)$$
$$= O\left(\log^2 \log(n/\log^2 \log n) \cdot \sum_{h\in H} |h|\right) = O\left(\log^2 \log(n/\log^2 \log n)\frac{n}{\log^2 \log n}\right) = O(n)$$

Summing the elements of every hash function is bounded by the maximum number of promoted nodes, i.e. $O(n/\log^2 \log n)$. To conclude, we spend linear time constructing the hash functions in the micro trees that contain a heavy branching node.

We associate two predecessor data structures with each micro tree that contains a heavy node: The first predecessor structure contains every light node that is a child of a heavy node in the micro tree. We call this predecessor data structure for the *light predecessor structure* of the micro tree. The key for each light node is the string on the path from the root of the micro tree to the node itself padded with character \$ such that every string has length α. These keys are ordered lexicographically in the predecessor data structure and a successful query yields a pointer to the node. The second predecessor structure is similar to the first but contains every heavy node in the micro tree. We call this predecessor structure for the *heavy predecessor structure*. We use Lemma 3 for the predecessor structures. The total size of every light and heavy predecessor structures is $O(n)$ and a query in both takes $O(\log \log n)$ because the universe is of size $(\sigma + 1)^\alpha$.

For each light node that are a child of a heavy node we additionally store pointers to the range of SA_S that corresponds to the leaves in T_S that the light node spans.

3.3.2 Answering queries

We answer queries on long patterns as follows. First we search for the deepest micro tree in HT_S where the root prefixes P. We do this by navigating the heavy tree in chunks of α characters starting from the root. Assuming that we have already matched a prefix of P consisting of i chunks of α characters we need to show how to match the $(i+1)$th chunk: If the micro tree is of type 1 and P has length at least $(i+1)\alpha$, we try to hash the substring $P[i\alpha, (i+1)\alpha]$. If we obtain a node v from the hash function we continue matching chunk $P[(i+1)\alpha, (i+2)\alpha]$ from v. If the micro tree is of type 2 we compare α sized chunks of P with the string on the unique path from root to the first micro tree with an explicit root and continue matching from here. We have found the deepest micro tree where the root prefixes P when we are unable to match a complete chunk of α characters or are unable to reach a micro tree with an explicit root. From this micro tree we need to decide whether the query is answered by searching SA_S from a light node or answered by finding d_P in the micro tree, where d_P is defined as in Section 3.2, i.e. the deepest node in T_S that prefixes P. We check if P continues in a light node by querying the light predecessor structure of the micro tree with the next unmatched α characters from P and pad with character \$ if less than α characters remain unmatched in P. If the light node returned by the query represents a string that prefixes P we answer the query by searching the range of SA_S spanned by the light node with the packed matching algorithm.

When P does not continue in a light node we instead find and use d_P for answering the query: If the micro tree is of type 2b or the root of the micro tree represents P then d_P is the root of the micro tree. Otherwise, we find d_P with a technique, very similar to a technique used by Fredman and Willard [8], that queries the heavy predecessor structure three times as follows: We call the remaining part of P, padded to length α with character \$, for p_0. We first query the predecessor structure with p_0 which yields a node that represents a string n_0. We then construct a string, p_1, that consists of the longest common prefix of p_0 and n_0, and as above, padded to length α. We query the predecessor structure with p_1 which yields a new node that represents a string n_1. We then construct a string, p_2, that consists of the longest common prefix of p_0 and n_1, again padded to length α. At last, we query the predecessor structure with p_2 which returns d_P. Given d_P, we answer count, locate and lexicographic predecessor queries exactly as we did in Section 3.2.

Now we prove the correctness of our queries. First we prove that if P continues in a light node then the query in the light predecessor structure returns that light node: Assume that P goes through the light node l_P that has a heavy parent in the micro tree T_p and that we query the light predecessor structure with the string Q_α. Let L_{pred} be the string that represents l_P in the light predecessor structure. Since P goes through l_P then L_{pred} is identical or lexicographically smaller than Q_α. Let L'_{pred} be the successor of L_{pred} in the light predecessor structure. Since L_{pred} is lexicographically smaller than L'_{pred} and has a longer common prefix with Q_α than L'_{pred} has with Q_α, then L'_{pred} must be lexicographically larger than Q_α. Since Q_α is identical or lexicographically larger than L_{pred} and lexicographically smaller than L'_{pred}, a query on Q_α in the light predecessor structure will return l_P.

We now prove that the queries in the heavy predecessor structure always returns d_P: Because P is not prefixed by a leaf of the micro tree or a light node from the light predecessor structure we know that d_P is a heavy node in the micro trie. In Figure 3, d_P is depicted and P either ends on or deviates from the edge e that leads to the tree T_2. The trees T_1,

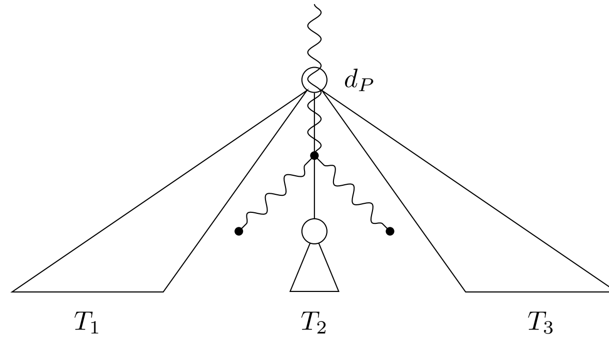

Figure 3 Searching for a prefix of P in HT_S.

T_2 and T_3 combined with d_P and the edge e constitute the subtree of d_P. If P deviates to the left or ends on e then P is lexicographically smaller than every string represented in T_2. If P deviates to the right then P is lexicographically larger than every string represented in T_2. Assume that P deviates to the right on e. Then the query to the heavy predecessor structure with pattern p_0 will yield n_0 that represents the lexicographically largest string in T_2. The pattern p_1 will then be represented by the implicit node from where P deviates from e. The pattern p_1 is lexicographically smaller than every string represented in T_2 and a query will yield n_2 as the lexicographically largest node in T_1 or, if T_1 is empty, the node d_P. Either way, the query on p_2 will yield the node d_P. We can make similar arguments for the other cases where P ends on e, deviates left from e, ends at d_P or goes through d_P without following e.

The following gives an analysis of the running time of our queries. We spend at most $O(m/\alpha)$ time traversing the heavy tree. Both predecessor structures contains strings over a universe of size n such that a query takes $O(\log\log n)$ time using Lemma 3. Each light node spans at most $\log^2\log n$ leaves which corresponds to an interval of length $\log^2\log n$ in SA_S that we search in $O(m/\alpha + \log\log\log n)$ time with the word accelerated algorithm for matching in SA_S. Overall, we spend $O(m/\alpha + \log\log n)$ time for answering count and lexicographic predecessor queries and $O(m/\alpha + \log\log n + \mathrm{occ})$ time for answering locate queries. Since we only query this data structure for patterns where $m \geq \log_\sigma n - 1$ we have that $\log\log n = \log(\frac{\log n}{\log\sigma}\log\sigma) = \log\log_\sigma n + \log\log(\sigma) \leq \log(\log_\sigma n - 1) + 1 + \log\log(\sigma) \leq \log(m) + 1 + \log\log(\sigma)$, such that we answer count and lexicographic predecessor queries in $O(m/\alpha + \log m + \log\log\sigma)$ time and locate queries in $O(m/\alpha + \log m + \log\log\sigma + \mathrm{occ})$ time. Combined with our solution for patterns where $m < \log_\sigma n - 1$, that answer the queries in $O(\log\log\sigma)$ and $O(\log\log\sigma + \mathrm{occ})$ time, respectively, we can for patterns of *any* length answer count and lexicographic predecessor queries in $O(m/\alpha + \log m + \log\log\sigma)$ time and locate queries in $O(m/\alpha + \log m + \log\log\sigma + \mathrm{occ})$ time. This is our main result which is summarized in Thm 1.

References

1 Djamal Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *J. Discrete Algorithms*, 14:91–106, 2012. doi:10.1016/j.jda.2011.12.011.

2 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Towards optimal packed string matching. *Theor. Comput. Sci.*, 525:111–129, 2014. doi:10.1016/j.tcs.2013.06.013.

3 Philip Bille. Fast searching in packed strings. *J. Discrete Algorithms*, 9(1):49–56, 2011. doi:10.1016/j.jda.2010.09.003.

4 Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Proceedings of the 33rd International Colloquium on Automata, Languages, and Programming (ICALP 2006)*, volume 4051 of *LNCS*, pages 358–369. Springer, 2006. doi:10.1007/11786986_32.

5 Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

6 Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 160–171. Springer, 2015. doi:10.1007/978-3-319-19929-0_14.

7 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

8 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

9 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

10 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

11 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

12 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Proceedings of the 35th International Colloquium on Automata, Languages, and Programming (ICALP 2008)*, volume 5125 of *LNCS*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

13 Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. Packed compact tries: A fast and efficient data structure for online string processing. In Veli Mäkinen, Simon J. Puglisi, and Leena Salmela, editors, *Proceedings of the 27th International Workshop on Combinatorial Algorithms (IWOCA 2016)*, volume 9843 of *LNCS*, pages 213–225. Springer, Springer, 2016. doi:10.1007/978-3-319-44543-4_17.

14 Peter Weiner. Linear pattern matching algorithms. In H. Raymond Strong, editor, *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.