Compressed Cache-Oblivious String B-tree[⋆]

Paolo Ferragina and Rossano Venturini

Dipartimento di Informatica, Università di Pisa, Italy

Abstract In this paper we address few variants of the well-known prefix-search problem in a dictionary of strings, and provide solutions for the cache-oblivious model which improve the best known results.

1 Introduction

The Prefix Search problem is probably the most well-known problem in data-structural design for strings. It asks for preprocessing a set $\mathcal{S}$ of K strings, having total length N, in such a way that, given a query-pattern P, it can return efficiently in time and space (the range of) all strings in $\mathcal{S}$ having P as a prefix. This easy-to-formalize problem is the backbone of many other algorithmic applications, and recently it received a revamped interest because of its Web-search (e.g., auto-completion search) and Internet-based (e.g., IP-lookup) applications.

In order to prove our surprising statement, and thus contextualize the contribution of this paper, we need to survey the main achievements in this topic and highlight their missing algorithmic points. The first solution to the prefix-search problem dates back to Fredkin [13], who introduced in 1960 the notion of (Compacted) trie to solve it. The trie structure became very famous in the '80s-'90s due to its suffix-based version, known as the Suffix Tree, which dominated first the String-matching scene [1], and then Bio-Informatics [15]. Starting from the Oxford English Dictionary initiative [12], and the subsequent advent of the Web, the design of tries managing large sets of strings became mandatory. It turned immediately clear that laying out a trie in a disk memory with page size B words, requiring optimal space and path traversals in $O(|P|/B)$ I/Os was not an easy task. And indeed Demaine *et al.* [6] showed that any layout of an arbitrary tree (and thus a trie) in external memory needs a poor number of I/Os to traverse a downward path spelling the pattern P.

The turning point in disk-efficient prefix-searching was the design of the String B-tree data structure [9], which was able to achieve $O(\log_B K + \mathsf{Scan}(P))$ I/Os, where $\mathsf{Scan}(P) = O(1 + \frac{|P|}{B \cdot \log N})$ indicates the number of I/Os needed to examine the input pattern, given that each disk page consists of B memory-words each of $\Theta(\log N)$ bits, and $|P|$ denotes the length of the binary representation of the pattern P. String B-trees provided I/O-efficient analogues of tries and suffix trees, with the specialty of introducing some *redundancy* in the representation

⋆ This work was partially supported by MIUR of Italy under projects PRIN ARS Technomedia 2012 and FIRB Linguistica 2006, the Midas EU Project, and the eCloud EU Project.

of the classic trie, which allowed the author to surpass the lower bounds in [6]. The I/O-bound is optimal whenever the alphabet size is large and the data structure is required to support the search for the lexicographic position of P among the strings $\mathcal{S}$ too. The space usage is $O(K \log N + N)$ bits, which is uncompressed, given that strings and pointers are stored explicitly. The string B-tree is based upon a careful orchestration of a B-tree layout of string pointers, plus the use of a Patricia Trie [19] in each B-tree node which organizes its strings in optimal space and supports prefix searches in $O(1)$ string accesses. Additionally, the string B-tree is *dynamic* in that it allows the insertion/deletion of individual strings from $\mathcal{S}$. As for B-trees, the data structure needs to know B in advance so depending from this parameter crucially for its design. Brodal and Fagerberg [4] made one step further by removing the dependance on B, and thus designing a static *cache-oblivious* trie-like data structure [14]. Unlike string B-trees, this structure is independent of B and does store, basically, a trie over the indexed strings *plus* few paths which are *replicated multiple times.* This *redundancy* is the essential feature that gets around the lower bound in [6], and it comes essentially at no additional asymptotic space-cost. Overall this solution solves the prefix-search in $O(\log_B K + \mathsf{Scan}(P))$ I/Os by using $O(K \log N + N)$ bits of space, simultaneously over all values of B and thus cache-obliviously, as currently said in the literature. In order to reduce the space-occupancy, Bender *et al.* [3] designed the (randomized) cache-oblivious string B-tree (shortly, COSB). It achieves the improved space of $(1+\epsilon)\mathsf{FC}(\mathcal{S}) + O(K \log N)$ bits, where $\mathsf{FC}(\mathcal{S})$ is the space required by the Front-coded storage of the strings in $\mathcal{S}$ (see Section 2), and ϵ is a user-defined parameter that controls the trade-off between space occupancy and I/O-complexity of the query/update operations. COSB supports searches in $O(\log_B K + (1+\frac{1}{\epsilon})(\mathsf{Scan}(P) + \mathsf{Scan}(succ(P))))$ I/Os, where $succ(P)$ is the successor of P in the ordered $\mathcal{S}$.[1] The solution is randomized so I/O-bounds holds with high probability. Furthermore, observe that the term $O((1+\frac{1}{\epsilon})\mathsf{Scan}(succ(P)))$ may degenerate becoming $\Theta((1+\frac{1}{\epsilon})\sqrt{N}/B)$ I/Os for some sets of strings. Subsequently, Ferragina *et al.* [10] proposed an improved cache-oblivious solution for the static-version of the problem regarding the space occupancy. They showed that there exists a static data structure which takes $(1+\epsilon)\mathsf{LT}(S) + O(K)$ bits, where $\mathsf{LT}(\mathcal{S})$ is a lower-bound to the storage complexity of $\mathcal{S}$. Searches can be supported in $O(\log_2 K + \mathsf{Scan}(P))$ I/Os or in $O(\log_B K + (1+\frac{1}{\epsilon})(\mathsf{Scan}(P) + \mathsf{Scan}(succ(P))))$ I/Os. Even if this solution is deterministic, its query complexity still has the costly dependency on $\mathsf{Scan}(succ(P))$. For completeness, we notice that the literature proposes many other compressed solutions but their searching algorithms are not suitable for the Cache-oblivious model (see e.g., [11,21,17]).

Recently, Belazzougui *et al.* [2] introduced the *weak* variant of the problem that allows for a *one-side* answer, namely the answer is requested to be correct only in the case that P prefixes some of the strings in $\mathcal{S}$; otherwise, it leaves to

[1] This index can be also dynamized to support insertion and deletion of a string P in $O(\log_B K + (\log^2 N)\ (1+\frac{1}{\epsilon})\mathsf{Scan}(P))$ I/Os plus the cost of identifying P's rank within $\mathcal{S}$.

the algorithm the possibility to return a un-meaningful solution to the query. The *weak*-feature allowed the authors of [2] to reduce the space occupancy from $O(N)$, which occurs when strings are incompressible, to the surprisingly succinct bound of $O(K \log \frac{N}{K})$ bits which was indeed proved to be a space lower-bound, regardless of the query complexity. In the cache-oblivious model the query complexity of their solution is $O(\log_2 |P| + \mathsf{Scan}(P))$ I/Os. Their key contribution was to propose a solution which do not store the string set $\mathcal{S}$ but uses only $O(\log \frac{N}{K})$ bits per string. This improvement is significant for very-large string sets, and we refer the reader to [2] for a discussion on its applications. Subsequently, Ferragina [8] proposed a very simple (randomized) solution for the weak-prefix search problem which matches the best known solutions for the prefix-search and the weak prefix-search, by obtaining $O(\log_B N + \mathsf{Scan}(P))$ I/Os within $O(K \log \frac{N}{K})$ bits of space occupancy. The searching algorithm is randomized, and thus its answer is correct with high probability.

In this paper we attack three problems of increasing sophistication by posing ourselves the challenging question: *how much redundancy we have to add to the classic trie space occupancy in order to achieve $O(\log_B K + \mathsf{Scan}(P))$ I/Os in the supported search operations.*[2]

Weak-prefix search. Returns the (lexicographic) range of strings prefixed by P, or an arbitrary value whenever such strings do not exist.

Full-Prefix search. Returns the (lexicographic) range of strings prefixed by P, or $\perp$ if such strings do not exist.

Longest-Prefix search. Returns the (lexicographic) range of strings sharing the longest common prefix with P.

We get the above I/O-bound for Weak-Prefix Search Problem, for the other problems we achieve $O(\log_B K + (1 + \frac{1}{\epsilon})\mathsf{Scan}(P))$ I/Os, for any constant $\epsilon > 0$. The space complexities are asymptotically optimal, in that they match the space lower bound for tries up to constant factors. The query complexity is optimal for the latter problem. This means that for Weak-Prefix Search Problem we improve [8] via a *deterministic* solution (rather than randomized) with a better space occupancy and a better I/O-complexity; for Longest-Prefix Search Problem we improve both [3] and [10] via a space-I/O optimal *deterministic* solution (rather than randomized, space sub-optimal, or I/O-inefficient solutions in [3] and [10]). Technically speaking, our results are obtained by adopting few technicalities plus a *new storage scheme* that extends the *Locality Preserving Front-Coding* scheme, at the base of COSB, in such a way that prefixes of the compressed strings can be decompressed in optimal I/Os, rather than just the entire strings. This scheme is surprisingly simple and it can be looked as a compressed version of the Blind-trie, backbone of the String B-tree [9].

[2] We remark that this query bound can be looked at as nearly optimal for the first two problems because it has not been proved yet that the term $\log_B K$ is necessary within the space bounds obtained in this paper.

$\mathcal{S}$	The set of strings
N	Total length of the strings in $\mathcal{S}$
K	Number of strings in $\mathcal{S}$
$\mathcal{T}_\mathcal{S}$	The compact trie built on $\mathcal{S}$
t	Number of nodes in $\mathcal{T}_\mathcal{S}$, it is $t \leq 2K - 1$
$\mathsf{p}(u)$	The parent of the node u in $\mathcal{T}_\mathcal{S}$
$\mathsf{string}(u)$	The string spelled out by the path in $\mathcal{T}_\mathcal{S}$ reaching u from the root
$\mathsf{label}(u)$	The label of the edge $(\mathsf{p}(u), u)$
$\hat{\mathcal{S}}$	The set $\mathcal{S}$ augmented with all strings $\mathsf{string}(u)$
$\mathsf{Trie}(\mathcal{S})$	The sum of edge-label lengths in $\mathcal{T}_\mathcal{S}$
$\mathsf{LT}(\mathcal{S})$	Lower bound (in bits) to the storage complexity of the set of strings $\mathcal{S}$ (it is $\mathsf{LT}(\mathcal{S}) = \mathsf{Trie}(\mathcal{S}) + \log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$)

Table 1: A summary of our notation and terminology.

2 Notation and background

In order to simplify the following presentation of our results, we assume to deal with *binary* strings. In the case of a larger alphabet Σ, it is enough to transform the strings over Σ into binary strings, and then apply our algorithmic solutions. The I/O complexities do not change because they depend only on the number of strings K in $\mathcal{S}$ and on the number of bits that fit in a disk block (hence $\Theta(B \log N)$ bits). As a further simplifying assumption we take $\mathcal{S}$ to be *prefix free*, so that no string in the set is prefix of another string; condition that is satisfied in practice because of the `null`-character terminating each string.

Table 1 summarizes all our notation and terminology. Below we will briefly recall few algorithmic tools that we will deploy to design our solutions to the prefix-search problem. We start with Front Coding, a compression scheme for strings which represents $\mathcal{S}$ as the sequence $\mathsf{FC}(\mathcal{S}) = \langle n_1, L_1, n_2, L_2, \ldots, n_K, L_K \rangle$, where n_i is the length of longest common prefix between S_{i-1} and S_i, and L_i is the suffix of S_i remaining after the removal of its first n_i (shared) characters, hence $L_i = |S_i| - n_i$. The first string S_1 is represented in its entirety, so that $L_1 = S_1$ and $n_1 = 0$. FC is a well established practical method for encoding a string set [22], and we will use interchangeably FC to denote either the algorithmic scheme or its output size in bits.

In order to estimate the space-cost of $\mathsf{FC}(\mathcal{S})$ in bits, the authors of [10] introduced the so called *trie measure* of $\mathcal{S}$, defined as: $\mathsf{Trie}(\mathcal{S}) = \sum_{i=1}^{K} |L_i|$, which accounts for the number of characters outputted by $\mathsf{FC}(\mathcal{S})$. And then, they devised a lower-bound $\mathsf{LT}(\mathcal{S})$ to the storage complexity of $\mathcal{S}$ which adds to the trie measure the cost, in bits, of storing the lengths $|L_i|$s. We have $\mathsf{LT}(\mathcal{S}) = \mathsf{Trie}(\mathcal{S}) + \log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$ bits.

In the paper we will often obtain bounds in terms of $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$, so the following fact is helpful:

Fact 1 *For any dictionary of strings $\mathcal{S}$, we have $\log\binom{\mathsf{Trie}(\mathcal{S})}{t-1} = O(K\log\frac{N}{K})$. Nevertheless there exist dictionaries for which $K\log\frac{N}{K}$ may be up to $\log K$ times larger than $\log\binom{\mathsf{Trie}(\mathcal{S})}{t-1}$. Finally, it is $O(\log\binom{\mathsf{Trie}(\mathcal{S})}{t-1}) = o(\mathsf{Trie}(\mathcal{S})) + O(K)$.*

Despite its simplicity $\mathsf{FC}(\mathcal{S})$ is a good compressor for $\mathcal{S}$, and indeed [10] showed that the representation obtained via Front-Coding takes the following number of bits:

$$\mathsf{LT}(\mathcal{S}) \leq \mathsf{FC}(\mathcal{S}) \leq \mathsf{LT}(\mathcal{S}) + O(K\log\frac{N}{K}). \tag{1}$$

It is possible to show that there exist pathological cases in which Front Coding requires space close to that upper bound. The main drawback of Front-coding is that decoding a string S_j might require the decompression of the entire sequence $\langle 0, L_1, \ldots, n_j, L_j\rangle$. In order to overcome this drawback, Bender *et al.* [3] proposed a variant of FC, called *locality-preserving front coding* (shortly, LPFC), that, given a parameter ϵ, adaptively partitions $\mathcal{S}$ into blocks such that decoding any string S_j takes $O((1+\frac{1}{\epsilon})|S_j|/B)$ optimal time and I/Os, and requires $(1+\epsilon)\mathsf{FC}(\mathcal{S})$ bits of space. This adaptive scheme offers a clear space/time tradeoff in terms of the user-defined parameter ϵ and it is agnostic in the parameter B.

A different linearization, called *rear-coding* (RC), is a simple variation of FC which *implicitly* encodes the length n_i by specifying the length of the suffix of S_{i-1} to be removed from it in order to get the longest common prefix between S_{i-1} and S_i. This change is crucial to avoid *repetitive* encodings of the same longest common prefixes, the space inefficiency in FC. And indeed RC is able to come very close to LT, because we can encode the lengths of the suffixes to be dropped via a binary array of length $\mathsf{Trie}(\mathcal{S})$ with $K-1$ bits set to 1, as indeed those suffixes partition $\mathcal{T}_{\mathcal{S}}$ into K disjoint paths from the leaves to the root. So RC can be encoded in

$$\mathsf{RC}(\mathcal{S}) \leq \mathsf{Trie}(\mathcal{S}) + 2\log\binom{\mathsf{Trie}(\mathcal{S})}{K-1} + O(\log\mathsf{Trie}(\mathcal{S})) = (1+o(1))\mathsf{LT}(\mathcal{S}) \text{ bits}, \tag{2}$$

where the latter equality follows from the third statement in Fact 1. Comparing eqn. (2) and (1), the difference between RC and FC is in the encoding of the n_i, so $\mathsf{Trie}(\mathcal{S}) \leq N$ (in practice $\mathsf{Trie}(\mathcal{S}) \ll N$).

In the design of our algorithms and data structures we will need two other key tools which are nowadays the backbone of every compressed index: namely, $\mathsf{Rank}/\mathsf{Select}$ data structures for binary strings. Their complexities are recalled in the following theorems.

Theorem 1 ([7]). *A binary vector $B[1..m]$ with n bits set to 1 can be encoded within $\log\binom{m}{n}+O(n)$ bits so that we can solve in $O(1)$ time the query $\mathsf{Select}_1(B,i)$, with $1 \leq i \leq n$, which returns the position in B of the ith occurrence of 1.*

Theorem 2 ([20]). *A binary vector $B[1..m]$ with n bits set to 1 can be encoded within $m+o(m)$ bits so that we can solve in $O(1)$ time the queries $\mathsf{Rank}_1(B,i)$, with $1 \leq i \leq m$, which returns the number of 1s in the prefix $B[1..i]$, and $\mathsf{Select}_1(B,i)$, with $1 \leq i \leq n$, which returns the position in B of the ith occurrence of 1.*

3 A key tool: Cache-Oblivious Prefix Retrieval

The novelty of our paper consists of a surprisingly simple representation of $\mathcal{S}$ which is compressed and still supports the cache-oblivious retrieval of any prefix of any string in this set in optimal I/Os and space (up to constant factors). The striking news is that, despite its simplicity, this result will constitute the backbone of our improved algorithmic solutions.

In this section we instantiate our solution on tries even if it is sufficiently general to represent any (labeled) tree in compact form still guaranteeing optimal traversal in the cache-oblivious model of any root-to-a-node path. We assume that the trie nodes are numbered accordingly to the time of their DFS visit. Any node u in $\mathcal{T}_{\mathcal{S}}$ is associated with $\mathsf{label}(u)$ which is (the variable length) string on the edge $(p(u), u)$, where $p(u)$ is the parent of u in $\mathcal{T}_{\mathcal{S}}$. Observe that any node u identifies uniquely the string $\mathsf{string}(u)$ that prefixes all strings of $\mathcal{S}$ descending from u. Obviously, $\mathsf{string}(u)$ can be obtained by juxtaposing the labels of the nodes on the path from the root to u. Our goal is to design a storage scheme whose space occupancy is $(1+\epsilon)\mathsf{LT}(\mathcal{S}) + O(K)$ bits and supports in optimal time/IO the operation $\mathsf{Retrieval}(u, \ell)$ which asks for the prefix of the string $\mathsf{string}(u)$ having length $\ell \in (|\mathsf{string}(\mathsf{p}(u))|, |\mathsf{string}(u)|]$. Note that the returned prefix ends up in the edge $(\mathsf{p}(u), u)$. In other words, we want to be able to access the labels of the nodes in any root-to-a-node path and any of their prefixes. Formally, we aim to prove the following theorem.

Theorem 3. *Given a set $\mathcal{S}$ of K binary strings having total length N, there exists a storage scheme for $\mathcal{S}$ that occupies $(1+\epsilon)\mathsf{LT}(\mathcal{S}) + O(K)$ bits, where $\epsilon > 0$ is any fixed constant, and solves the query $\mathsf{Retrieval}(u, \ell)$ in $O(1 + (1 + \frac{1}{\epsilon})\frac{\ell}{B \log N})$ optimal I/Os.*

Before presenting a proof, let us discuss efficiency of two close relatives of our solution: *Giraffe tree decomposition* [4] and *Locality-preserving front Coding* (LPFC) [3]. The former solution has the same time complexity of our solution but has a space occupancy of at least $3 \cdot \mathsf{LT}(\mathcal{S}) + O(K)$ bits. The latter approach has (almost) the same space occupancy of our solution but provides no guarantee on the number of I/Os required to access *prefixes* of the strings in $\mathcal{S}$.

Our goal is to accurately lay out the labels of nodes of $\mathcal{T}_{\mathcal{S}}$ so that any $\mathsf{string}(u)$ can be retrieved in optimal $O((1 + \frac{1}{\epsilon})\mathsf{Scan}(\mathsf{string}(u)))$ I/Os. This suffices for obtaining the bound claimed in Theorem 3 because, once we have reconstructed $\mathsf{string}(p(u))$, $\mathsf{Retrieval}(u, \ell)$ is completed by accessing the prefix of $\mathsf{label}(u)$ of length $j = \ell - |\mathsf{string}(p(u))|$ which is written consecutively in memory. One key feature of our solution is a proper replication of some labels in the lay out, whose space is bounded by $\epsilon \cdot \mathsf{LT}(\mathcal{S})$ bits.

The basis of our scheme is the amortization argument in LPFC [3] which represents $\mathcal{S}$ by means of a variant of the classic front-coding in which some strings are stored explicitly rather than front-coded. More precisely, LPFC writes the string S_1 explicitly, whereas all subsequent strings are encoded in accordance with the following argument. Suppose that the scheme already compressed $i-1$ strings and has to compress string S_i. It scans back $c|S_i|$ characters in the current representation to check if it is possible to decode S_i, where $c = 2 + 2/\epsilon$. If this is the case, S_i is compressed by writing its suffix L_i, otherwise S_i is fully written. A sophisticated amortization argument in [3] proves that LPFC requires $(1+\epsilon)\mathsf{LT}+O(K\log(N/K))$ bits. This bound can be improved to $(1+\epsilon)\mathsf{LT}+O(K)$ bits by replacing front-coding with rear-coding (see eqn. 2). By construction, this scheme guarantees an optimal decompression time/IO of any string $S_i \in \mathcal{S}$, namely $O((1+\frac{1}{\epsilon})\mathsf{Scan}(S_i))$ I/Os. But unfortunately, this property does not suffice to guarantee an optimal decompression for prefixes of the strings: the decompression of *a prefix* of a string S_i may cost up to $\Theta((1+\frac{1}{\epsilon})\mathsf{Scan}(S_i))$ I/Os.

In order to circumvent this limitation, we modify LPFC as follows. We define the superset $\hat{\mathcal{S}}$ of $\mathcal{S}$ which contains, for each node u in $\mathcal{T}_{\mathcal{S}}$ (possibly a leaf), the string $\mathsf{string}(u)$. This string is a prefix of $\mathsf{string}(v)$, for any descendant v of u in $\mathcal{T}_{\mathcal{S}}$, so it is lexicographically smaller than $\mathsf{string}(v)$. The lexicographically ordered $\hat{\mathcal{S}}$ can thus be obtained by visiting the nodes of $\mathcal{T}_{\mathcal{S}}$ according to a DFS visit. In our solution we require that all the strings emitted by $\mathsf{LPFC}(\hat{\mathcal{S}})$ are self-delimited. Thus, we prefix each of them with its length coded with Elias' Gamma coding. Now, let R be the compressed output obtained by computing $\mathsf{LPFC}(\hat{\mathcal{S}})$ with rear-coding. We augment R with two additional data structures:

– The binary array $E[1..|R|]$ which sets to 1 the positions in R where the encoding of some $\mathsf{string}(u)$ starts. E contains $t-1$ bits set to 1. Array E is enriched with the data structure in Theorem 1 so that Select_1 queries can be computed in constant time.
– The binary array $V[1..t]$ that has an entry for each node in $\mathcal{T}_{\mathcal{S}}$ according to their (DFS-)order. The entry $V[u]$ is sets to 1 whenever $\mathsf{string}(u)$ has been copied in R, 0 otherwise. We augment V with the data structure of Theorem 2 to support Rank and Select queries.

In order to answer $\mathsf{Retrieval}(u,\ell)$ we first implement the retrieval of $\mathsf{string}(u)$. The query $\mathsf{Select}_1(E,u)$ gives in constant time the position in R where the encoding of $\mathsf{string}(u)$ starts. Now, if this string has been fully copied in R then we are done; otherwise we have to reconstruct it. This has some subtle issues that have to be addressed efficiently, for example, we do not even know the length of $\mathsf{string}(u)$ since the array E encodes the individual edge-labels and not their lengths from the root of $\mathcal{T}_{\mathcal{S}}$. We reconstruct $\mathsf{string}(u)$ forward by starting from the first copied string (say, $\mathsf{string}(v)$) that precedes $\mathsf{string}(u)$ in R. The node index v is obtained by computing $\mathsf{Select}_1(V,\mathsf{Rank}_1(V,u))$ which identifies the position of the first 1 in V that precedes the entry corresponding to u (i.e., the closer copied strings preceding u in the DFS-visit of $\mathcal{T}_{\mathcal{S}}$).

Assume that the copy of $\mathsf{string}(v)$ starts at position p_v, which is computed by selecting the v-th 1 in the E. By the DFS-order in processing $\mathcal{T}_{\mathcal{S}}$ and by the fact that $\mathsf{string}(u)$ is not copied, it follows that $\mathsf{string}(u)$ can be reconstructed by copying characters in R starting from position p_v up to the occurrence of $\mathsf{string}(u)$. We recall that rear-coding augments each emitted string with a value: let w and w' two nodes consecutive in the DFS-visit of $\mathcal{T}_{\mathcal{S}}$, rear-coding writes the value $|\mathsf{string}(w)| - \mathsf{lcp}(\mathsf{string}(w),\mathsf{string}(w'))$ (namely, the length of the suffix of $\mathsf{string}(w)$ that we have to remove from w in order to obtain the length of its longest common prefix with $\mathsf{string}(w'))$. This information is exploited in reconstructing $\mathsf{string}(u)$. We start by copying $\mathsf{string}(v)$ in a buffer by scanning R forward from position p_v. At the end, the buffer will contain $\mathsf{string}(u)$. For every value m written by rear-coding, we overwrite the last m characters of the buffer with the characters in R of the suffix of the current string (delimited by E's bits set to 1). By LPFC-properties, we are guaranteed that this scan takes $O((1 + \frac{1}{\epsilon})\mathsf{Scan}(\mathsf{string}(u))$ I/Os.

Let us now come back to the solution of $\mathsf{Retrieval}(u, \ell)$. First of all we reconstruct $\mathsf{string}(\mathsf{p}(u))$, then determine the edge-label $(\mathsf{p}(u), u)$ in E given the DFS-numbering of u and a select operation over E. We thus take from this string its prefix of length $\ell - |\mathsf{string}(\mathsf{p}(u))|$, the latter is known because we have indeed reconstructed that string.

To conclude the proof of Theorem 3 we are left with bounding the space occupancy of our storage scheme. We know that R requires no more than $(1 + \epsilon)\mathsf{LT}(\hat{\mathcal{S}}) + O(K)$ bits, since we are using rear-coding. The key observation is then the trie measure of $\hat{\mathcal{S}}$ coincides with the one of $\mathcal{S}$ (i.e., $\mathsf{Trie}(\hat{\mathcal{S}}) = \mathsf{Trie}(\mathcal{S})$), so that $|R| = (1+\epsilon)\mathsf{LT}(\hat{\mathcal{S}}) + O(K) = (1+\epsilon)\mathsf{LT}(\mathcal{S}) + O(K)$. The space occupancy of E is $\log \binom{|R|}{t-1}$ bits (Theorem 1), therefore $|E| \leq t \log(|R|/t) + O(t) = o(\mathsf{Trie}(\mathcal{S})) + O(K)$ bits. It is easy to see that the cost of self-delimiting the strings emitted by LPFC with Elias' Gamma coding is within the same space bound. The vector V requires just $O(K)$ bits, by Theorem 2.

The query $\mathsf{Retrieval}(u, \ell)$ suffices for the aims of this paper. However, it is more natural an operation that, given a string index i and a length ℓ, returns the prefix of $S_i[1..\ell]$. This can be supported by using a variant of the ideas presented later for the Weak-prefix Search problem, which adds $O(\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1} + K) = o(\mathsf{Trie}(\mathcal{S})) + O(K)$ bits to space complexity (hidden by the other terms) and a term $O(\log_B K)$ I/Os to query time. Alternatively, it is possible to keep an I/O-optimal query time by adding $O(K \log \frac{N}{K})$ bits of space.

4 Searching strings: three problems

In this section we address the three problems introduced in the Introduction, they allow us to frame the wide spectrum of algorithmic difficulties and solutions related with the search for a pattern within a string set.

Problem 1 (Weak-Prefix Search Problem) Let $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ be a set of K binary strings of total length N. We wish to preprocess $\mathcal{S}$ in such a way that, given a pattern P, we can efficiently answer the query $\mathsf{weakPrefix}(P)$ which asks for the range of strings in $\mathcal{S}$ prefixed by P. An arbitrary answer could be returned whenever P is not a prefix of any string in $\mathcal{S}$. □

The lower bound in [2] states that $\Omega(K \log \frac{N}{K})$ bits are necessary regardless the query time. We show the following theorem.

Theorem 4. *Given a set of $\mathcal{S}$ of K binary strings of total length N, there exists a deterministic data structure requiring $2 \cdot \log \binom{\mathsf{Trie}(\mathcal{S})}{t-1} + O(K)$ bits of space that* *solves the* Weak-Prefix Search Problem *for any pattern P with $O(\log_B K + \mathsf{Scan}(P))$ I/Os.*

The space occupancy is optimal up to constant factor since $\log \binom{\mathsf{Trie}(\mathcal{S})}{t-1}$ is always at most $K \log \frac{N}{K}$ (see Fact 1). Moreover, our refined estimate of the space occupancy, by means of $\mathsf{Trie}(\mathcal{S})$, shows that it can go below the lower bound in [2] even by a factor $\Theta(\log K)$ depending on the characteristics of the indexed dictionary (see Fact 1). The query time instead is almost optimal, because it is not clear whether the term $\log_B K$ is necessary within this space bound. Summarizing, our data structure is smaller, deterministic and faster than previously known solutions.

We follow the solution in [8] by using a two-level indexing. We start by partitioning $\mathcal{S}$ into $s = K/\log N$ groups of (contiguous) strings defined as follows: $\mathcal{S}_i = \{S_{1+i\log N}, S_{2+i\log N}, \ldots, S_{(i+1)\log N}\}$ for $i = 0, 1, 2, \ldots, s-1$. We then construct a subset $\mathcal{S}_{\mathsf{top}}$ of $\mathcal{S}$ consisting of $2s = \Theta(\frac{n}{\log n})$ representative strings obtained by selecting the smallest and the largest string in each of these groups. The index in the first level is responsible for searching the pattern P within the set $\mathcal{S}_{\mathsf{top}}$, in order to identify an approximated range. This range is guaranteed to contain the range of strings prefixed by P. A search on the second level suffices to identify the correct range of strings prefixed by P. We have two crucial differences w.r.t. the solution in [8]: 1) our index is deterministic; 2) our space-optimal solution for the second level is the key for achieving Theorem 4.
First level. As in [8] we build the Patricia Trie $\mathsf{PT}_{\mathsf{top}}$ over the strings in $\mathcal{S}_{\mathsf{top}}$ with the speciality that we store in each node u of $\mathsf{PT}_{\mathsf{top}}$ a fingerprint of $O(\log N)$ bits computed for $\mathsf{string}(u)$ according to Karp-Rabin fingerprinting [18]. The crucial difference w.r.t. [8] is the use of a (deterministic) injective instance of Karp-Rabin that maps any prefix of any string in $\mathcal{S}$ into a distinct value in a interval of size $O(N^2)$.[3] Given a string $S[1..s]$, the Karp-Rabin fingerprinting $\mathsf{rk}(S)$ is equal to $\sum_{i=1}^{s} S[i] \cdot t^i \pmod M$, where M is a prime number and t is a randomly chosen integer in $[1, M-1]$. Given the set of strings $\mathcal{S}$, we can obtain an instance $\mathsf{rk}()$ of the Karp-Rabin fingerprinting that maps all the prefixes of all the strings in $\mathcal{S}$ to the first $[M]$ integers without collisions, with M chosen among the first $O(N^2)$ integers. It is known that a value of t that guarantees no collisions can be found in expected $O(1)$ attempts. In the cache-oblivious setting, this implies that finding a suitable function requires $O(\mathsf{Sort(N)} + N/B)$ I/Os in expectation, where $\mathsf{Sort(N)}$ is the number of I/Os required to sort N integers.

Given $\mathsf{PT}_{\mathsf{top}}$ and the pattern P, our goal is that of finding the lowest edge $e = (v, w)$ such that $\mathsf{string}(v)$ is a prefix of P and $\mathsf{string}(w)$ is not. This edge can be found with a standard blind search on $\mathsf{PT}_{\mathsf{top}}$ and by also comparing fingerprints of P with the ones stored in the traversed nodes (see [8] for more details). A cache-oblivious efficient solution is obtained by laying out $\mathsf{PT}_{\mathsf{top}}$ via the centroid trie decomposition [3]. This layout guarantees that the above search requires $O(\log_B K + \mathsf{Scan}(P))$ I/Os. However, in [8] the edge e is correctly identified only

[3] Notice that we require the function to be injective for prefixes of strings in $\mathcal{S}$ not $\mathcal{S}_{\mathsf{top}}$.

with high probability. The reason is that a prefix of P and a prefix of a string in $\mathcal{S}$ may have the same fingerprint even if they are different. Our use of the injective Karp-Rabin fingerprints avoids this situation guaranteeing that the search is always correct[4].

Second level. For each edge $e = (u, v)$ of $\mathsf{PT}_{\mathsf{top}}$ we define the set of strings $\mathcal{S}_e$ as follows. Assume that each node v of $\mathsf{PT}_{\mathsf{top}}$ points to its leftmost/rightmost descending leaves, denoted by $L(v)$ and $R(v)$ respectively. We call $\mathcal{S}_{L(v)}$ and $\mathcal{S}_{R(v)}$ the two groups of strings, from the grouping above, that contain $S_{L(v)}$ and $S_{R(v)}$. Then $\mathcal{S}_e = \mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$. We have a total of $O(K/\log n)$ sets, each having $O(\log N)$ strings. The latter is the key feature that we exploit in order to index these small sets efficiently by resorting to the following lemma. We remark that $\mathcal{S}_e$ will not be constructed and indexed explicitly, rather we will index the sets $\mathcal{S}_{L(v)}$ and $\mathcal{S}_{R(v)}$ individually, and keep two pointers to each of them for every edge e. This avoids duplication of information and some subtle issues in the storage complexities. But poses the problem of how to weak-prefix search in $\mathcal{S}_e$ which is only virtually available. The idea is to search in $\mathcal{S}_{L(v)}$ and $\mathcal{S}_{R(v)}$ individually, three cases may occur. Either we find that the range is totally within one of the two sets, and in this case we return that range; or we find that the range includes the rightmost string in $\mathcal{S}_{L(v)}$ and the leftmost string in $\mathcal{S}_{R(v)}$, and in this case we merge them. The correctness comes from the properties of trie's structure and the first-level search, as one can prove by observing that the trie built over $\mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$ is equivalent to the two tries built over the two individual sets except for the rightmost path of $\mathcal{S}_{L(v)}$ and the leftmost path of $\mathcal{S}_{R(v)}$ which are merged in the trie for $\mathcal{S}_e$. This merging is not a problem because if the range is totally within $\mathcal{S}_{R(v)}$, then the dominating node is within the trie for this set and thus the search for P would find it by searching both $\mathcal{S}_{R(v)}$ or $\mathcal{S}_e$. Similarly this holds for a range totally within $\mathcal{S}_{L(v)}$. The other case comes by exclusion, so the following lemma allows to establish the claimed I/O and space bounds.

Lemma 1. *Let $\mathcal{S}_i$ be a set of $K_i = O(\log N)$ strings of total length at most N. The Patricia trie of $\mathcal{S}_i$ can be represented by requiring $\log \binom{\mathsf{Trie}(\mathcal{S}_i)}{t_i - 1} + O(K_i)$ bits of space so that the blind search of any pattern P with $O((\log K_i)/B + \mathsf{Scan}(P))$ I/Os, where t_i is the number of nodes in the trie of the set $\mathcal{S}_i$.*

To conclude the proof of Theorem 4, we distinguish two cases based on the value of K. If $K = O(\log N)$, we do not use the first level since Lemma 1 with $K_i = K$ already matches the bounds in Theorem 4. Otherwise $K = \Omega(\log N)$, and so searching P requires $O(\log_B K + \mathsf{Scan}(P))$ I/Os on the first level and $O((\log\log N)/B + \mathsf{Scan}(P)) = O(\log_B K + \mathsf{Scan}(P))$ I/Os on the second level. For the space occupancy, we observe that the first level requires $O(K)$ bits, and the second level requires $\sum_i (\log \binom{\mathsf{Trie}(\mathcal{S}_i)}{t_i - 1} + K_i)$ bits (Lemma 1). Note that $\sum_i t_i \leq t = O(K)$ because each string of $\mathcal{S}$ belongs to at most one $\mathcal{S}_i$.

Problem 2 (Full-Prefix Search Problem) Let $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ be a set of K binary strings of total length N. We wish to preprocess $\mathcal{S}$ in such a way that,

[4] Recall that in the Weak-Prefix Search Problem we are searching under the assumption that P is a prefix of at least a string in $\mathcal{S}$.

given a pattern P, we can efficiently answer the query $\mathsf{Prefix}(P)$ which asks for the range of strings in $\mathcal{S}$ prefixed by P, the value $\perp$ is returned whenever P is not a prefix of any string in $\mathcal{S}$. $\square$

This the classic prefix-search which requires to recognize whether P is or is not the prefix of any string in $\mathcal{S}$. By combining Theorems 3 and 4 we get:

Theorem 5. *Given a set of $\mathcal{S}$ of binary strings of size K of total length N, there exists a data structure requiring $(1+\epsilon)\mathsf{LT}(\mathcal{S}) + O(K)$ bits of space that solves the* Full-Prefix Search Problem *for any pattern P with $O(\log_B K + (1+\frac{1}{\epsilon})\mathsf{Scan}(P))$ I/Os, where $\epsilon > 0$ is any constant.*

We use the solution of Theorem 4 to identify the highest node u from which descends the largest range of strings that are prefixed by P. Then, we use Theorem 3 to check I/O-optimally whether $\mathsf{Retrieval}(u, |P|)$ equals P. The space occupancy of this solution is optimal up to a constant factor; the query complexity is almost optimal being unclear whether it is possible to remove the $\log_B K$ term and still maintain optimal space.

Problem 3 (Longest-Prefix Search Problem) Let $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ be a set of K binary strings of total length N. We wish to preprocess $\mathcal{S}$ in such a way that, given a pattern P, we can efficiently answer the query $\mathsf{LPrefix}(P)$ which asks for the range of strings in $\mathcal{S}$ sharing the longest common prefix with P. $\square$

This problem waives the requirement that P is a prefix of some string in $\mathcal{S}$, and thus searches for the longest common prefix between P and $\mathcal{S}$'s strings. If P prefixes some strings in $\mathcal{S}$, then this problem coincides with the classic prefix-search. Possibly the identified lcp is the *null* string, and thus the returned range of strings is the whole set $\mathcal{S}$. We will prove the following result.

Theorem 6. *Given a set of $\mathcal{S}$ of K binary strings of total length N, there exists a data structure requiring $(1+\epsilon)\mathsf{LT}(\mathcal{S}) + O(K)$ bits of space that solves the* Longest-Prefix Search Problem *for any pattern P with $O(\log_B K + (1+\frac{1}{\epsilon})\mathsf{Scan}(P))$ I/Os, where $\epsilon > 0$ is any constant.*

First we build the data structures of Theorem 3 with a constant ϵ' to be fixed later, in order to efficiently access prefixes of strings in $\mathcal{S}$ but also as a basis to partition the strings. It is convenient to observe this process on $\mathcal{T}_\mathcal{S}$. Recall that the data structure of Theorem 3 processes nodes of $\mathcal{T}_\mathcal{S}$ in DFS-order. For each visited node u, it encodes $\mathsf{string}(u)$ either by copying $\mathsf{string}(u)$ or by writing $\mathsf{label}(u)$. In the former case we will say that u is marked. Let $\mathcal{S}_{\mathsf{copied}}$ be the set formed by the $\mathsf{string}(u)$ of any marked node u. The goal of a query $\mathsf{LPrefix}(P)$ is to identify the lowest node w in $\mathcal{T}_\mathcal{S}$ sharing the longest common prefix with P. We identify the node w in two phases. In the first phase we solve the query $\mathsf{LPrefix}(P)$ on the set $\mathcal{S}_{\mathsf{copied}}$ in order to identify the range of all the (consecutive) marked nodes $[v_l, v_r]$ sharing the longest common prefix with P. Armed with this information, we start a second phase that scans appropriate portions of the compressed representation R of Theorem 3 to identify our target node w. (For space reasons we defer the description of our solution to the full paper.)

References

1. A. Apostolico. The myriad virtues of subword trees. *Combinatorial Algorithms on Words*, pages 85–96, 1985.
2. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *ESA*, pages 427–438. Springer, 2010.
3. M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *PODS*, pages 233–242. ACM, 2006.
4. G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *SODA*, pages 581–590. ACM Press, 2006.
5. A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.
6. E. D. Demaine, J. Iacono, and S. Langerman. Worst-case optimal tree layout in a memory hierarchy. *CoRR*, cs.DS/0410048, 2004.
7. P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.
8. P. Ferragina. On the weak prefix-search problem. In *CPM*, LNCS vol. 6661, pages 261–272. Springer, 2011.
9. P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
10. P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *PODS*, pages 181–190, 2008.
11. P. Ferragina, and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms* 7(1): 10, 2010.
12. W. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.
13. E. Fredkin. Trie memory. *Communication of the ACM*, 3(9):490–499, 1960.
14. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms* 8(1): 4, 2012.
15. D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.
16. M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *ICALP*, volume 4596 of *LNCS*, pages 509–520, 2007.
17. W.-K. Hon, R. Shah, and J. S. Vitter. Compression, Indexing, and Retrieval for Massive String Data. In *CPM*, pages 260–274, 2010
18. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
19. D. R. Morrison. PATRICIA - practical algorithm to retrieve coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
20. J. I. Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.
21. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.* 39(1), 2007.
22. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, 1999.