On the Weak Prefix-Search Problem ⋆

Paolo Ferragina

Dipartimento di Informatica, University of Pisa, Italy

Abstract

The weak-prefix search problem consists of searching for the strings in a dictionary $\mathcal{S}$ that are prefixed by a pattern $P[1,p]$; if no such string does occur, any answer can be returned. Strings in $\mathcal{S}$ have average length ℓ, are n in number, and are given in advance to be preprocessed, whereas the pattern P is provided on-line. In this paper we solve this problem in the external-memory and in the cache-oblivious models by using the optimal $O(n \log \ell)$ bits of space and requiring $O(p/B + \log_B n)$ I/Os. The searching algorithm is of Monte-Carlo type, so its answer is correct with high probability. We also discuss some variants of the problem concerning with a distribution over the queried patterns, a deterministic solution, and foresee applications in the design of energy-efficient data structures.

Key words: Prefix Search, Compressed Indexes for String Dictionaries, Weighted Data Structures, Energy-Efficient Data Structures

1 Introduction

Searching for a pattern $P[1,p]$ as a prefix of a set $\mathcal{S}$ of n strings of average length ℓ is a classic string-matching problem. All known solutions require $O(n\ell)$ bits of space in the worst case, because they store the strings of $\mathcal{S}$ explicitly. Recently Bellazougui 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 the algorithm the possibility to return an un-meaningful solution to the problem. Formally,

⋆ This work has been partially supported by a Yahoo! Research grant, PRIN MadWeb 2008 and FIRB Linguistica 2006. The author address is Dipartimento di Informatica, Largo B. Pontecorvo 3, 56127 Pisa, Italy. Email address: `ferragina@di.unipi.it`.

Problem 1 (Bellazougui *et al.*, 2010) *Let $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ be a prefix-free set of binary strings of average length ℓ, sorted alphabetically. The* weak *prefix-search problem asks, given a binary query-string $P[1,p]$, for the range of strings of $\mathcal{S}$ prefixed by P. The range is represented as a pair of string-ids, because $\mathcal{S}$ is sorted. In the case that P does not prefix any string of $\mathcal{S}$, the returned range can be* any *pair.*

The *weak*-feature allowed the authors of [2] to reduce the space occupancy from $O(n\ell)$ to $O(n \log \ell)$ bits, and indeed they proved that this is a space lower-bound. Their solution takes $O(p/w + \log_2 p)$ memory transfers for searching in the RAM model with a memory word of w bits. The key issue here is not to store the string set $\mathcal{S}$ but design an index that uses $O(\log \ell)$ bits per string. This improvement is significant for very-large string sets, and we refer the reader to [2] for a discussion on the possible applications.

But if the set $\mathcal{S}$ is large, either for the number of strings or for their total length, it is more appropriate to evaluate the algorithmic solutions in the External-Memory model in which B is the disk-page size and M is the size of the available internal memory [15], and both parameters are known to the algorithm. In this context the Bellazougui's result can be rephrased as $O(p/B + \log_2 p)$ I/Os, by setting $w = B$, still within the optimal $O(n \log \ell)$ bits of space occupancy. Noteworthy the authors of [2] showed that this I/O-bound holds also in the more powerful Cache-Oblivious model where the two model's parameters M and B are unknown to the running algorithm (see e.g. [11]).

On the other hand, the best known result for the *classic* prefix-search problem (see e.g. [4,9,10]) uses $O(n\ell)$ bits of space and $O(p/B + \log_B n)$ I/Os. This is always worse in terms of space occupancy than what is obtained for the weak prefix-search, but it is faster whenever $\log_B n < \log_2 p$. This latter inequality is realistic in practice even for moderate pattern's lengths, given that it corresponds to $n < p^{\log_2 B} \approx p^{15}$.

Starting from these observations, we have studied and designed a (randomized) solution for the weak-prefix search problem which matches the best of the two solutions above by obtaining $O(p/B + \log_B n)$ I/Os within $O(n \log \ell)$ bits of space occupancy. The searching algorithm is of Monte-Carlo type, so its answer is correct with high probability. Our solution has the nice feature of being algorithmically simpler than the one proposed in [2], and thus worth to be implemented. The paper actually details two different approaches that achieve the claimed bounds based on a 2-level indexing scheme (Section 3) and a suitable re-design and I/O-efficient mapping of the Patricia Trie data structure and the blind search procedure of [9] (Section 4). One of these approaches works on the external-memory model and boils down onto the careful orchestration of the disk-mapping of a trie devised in [13] with the I/O-optimal

access to unary paths in trees (Section 4.1). The other approach works in the cache-oblivious model and applies the centroid decomposition of a binary tree [3] to our 2-level indexing data structure (Section 4.2). As further contributions, we will also address a variation of the problem in which a *probability distribution* over the queried patterns is known (Section 5), and finally we will comment onto the design of a deterministic solution as well as other scenarios which would be worth to deal with in the near future (Section 6).

We conclude this section by noticing that our results are stated in terms of binary strings, for simplicity of exposition, but they generalize easily to strings drawn from an arbitrary alphabet Σ. This is obvious for a constant-sized alphabet, whereas the case of a larger alphabet can be managed by re-mapping its symbols occurring in $\mathcal{S}$'s strings to the range $[0, n\ell - 1]$, the second extreme denoting the total length of the strings in $\mathcal{S}$.

2 Background

A *compacted trie* $\mathcal{T}_\mathcal{S}$ built on the set $\mathcal{S}$ of binary strings is a tree in which the root may be unary, but all other internal nodes are binary and thus have exactly two children. $\mathcal{T}_\mathcal{S}$ consists of n leaves (one per string in $\mathcal{S}$) and no more than n internal nodes, hence $O(n)$ nodes in total. Each edge $e = (u, v)$ is labeled with a substring $s(e)$ of the strings in $\mathcal{S}$, each leaf l is labeled with an integer in the range $[1, n]$ denoting the rank of its associated string in $\mathcal{S}$, and each internal node u is labeled with an integer denoting the length of the string spelled out by the root-to-u path in $\mathcal{T}_\mathcal{S}$. Without abusing of the notation, we will use $s(u)$ to denote that string, and observe that in the case u is a leaf it is $s(u) \in \mathcal{S}$.

A *Patricia trie* $\mathcal{PT}_\mathcal{S}$ is derived from the compacted trie $\mathcal{T}_\mathcal{S}$ by substituting each substring labeling a tree edge with its first bit, commonly called *branching bit*. Figure 1 shows an example. In the following we will use $\mathcal{PT}_\mathcal{S}$ in combination with a special prefix-search procedure introduced in [9] and called *blind search*. The key idea is to search for the lexicographic position of P in $\mathcal{S}$ by percolating a root-to-leaf path in $\mathcal{PT}_\mathcal{S}$, matching only the available branching bits. Two cases may occur: (i) either P is exhausted and a node u is reached; or (ii) we end up into a leaf l.[1] In the first case, the authors of [9] suggest to take any leaf l descending from u, and they prove that $s(l)$ is one of the strings in $\mathcal{S}$ sharing the *longest common prefix* (lcp) with P [9, Lemma 1, pag 253]. This lcp-value together with another percolation of $\mathcal{PT}_\mathcal{S}$ was used in [9] to find P's

[1] The algorithm of [9] is formulated on a general alphabet Σ and considers also the case in which P mismatches all branching characters out of a node u. In our setting this cannot occur because strings and nodes are binary.

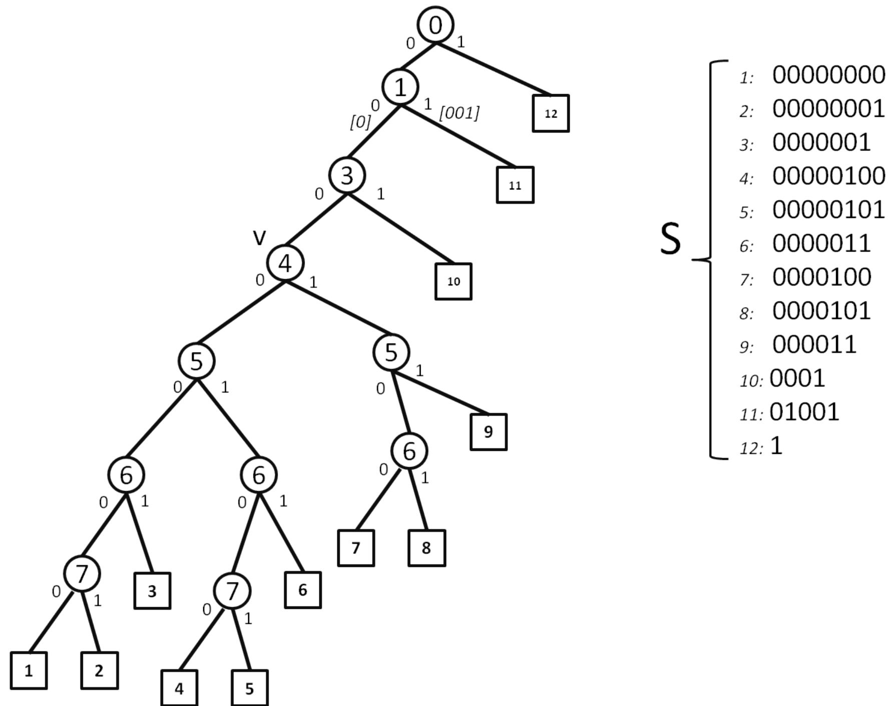

Fig. 1. *An example of compacted trie and its Patricia's variant built over a set $\mathcal{S}$ of 12 binary strings. Bits between brackets are the part of an edge-label that has been dropped when turning the compacted trie into the corresponding Patricia trie. In this example, this dropping occurs just twice.*

position in $\mathcal{S}$. The nice fact was that just one string of $\mathcal{S}$ was compared.

This single string-comparison is not possible in our weak-prefix search problem because $\mathcal{S}$ is not available. Nevertheless that result can be rephrased for our problem as follows: either the string $s(l)$ has prefix P or P does not prefix any string in $\mathcal{S}$. As an example take $P = 0010$, the blind search over the Patricia Trie in Figure 1 reaches the node v which is *not* the correct locus for the string P because there is a mismatch at the third bit that was not catched by the blind search since that bit was missing in the corresponding edge label. Anyway, this is not a problem because P is not a prefix of any string in $\mathcal{S}$, and so the answer to the weak-prefix search can be arbitrary. On the other hand if $P = 0000$ then the blind search stops again at the node v, and this is now correct since P prefixes all 9 strings descending from v.

The advantage of using the combination between blind search *and* $\mathcal{PT}_\mathcal{S}$, with respect to the classical approach over compacted tries, is that the blind search identifies one single node in $\mathcal{PT}_\mathcal{S}$ whose spelled string is one of the candidates for being prefixed by P, and this node can be found by deploying only the information available in $\mathcal{PT}_\mathcal{S}$, thus without any access to the string set $\mathcal{S}$.[2]

[2] The compacted trie needs to access $\mathcal{S}$'s strings to resolve the edge labels during a string search.

As a result we would be tempted to conclude that the weak prefix-search problem admits the simple solution sketched above. But there are two subtle shortcomings. The first one is that $\mathcal{PT}_\mathcal{S}$ takes $\Omega(n \log n)$ bits of space, which may be $\omega(n \log \ell)$; the second one is that the packing of $\mathcal{PT}_\mathcal{S}$ into pages of size B is a difficult problem in its generality, especially when dealing with the cache-oblivious model [7]. We address the first issue in the next Section 3, and we will solve the second issue by proposing two different solutions, detailed in Section 4, which are tailored to deploy the specialities of the blind-search procedure over Patricia tries.

3 A 2-level indexing scheme for RAM

Our first step is to propose a 2-level indexing scheme whose goal is to reduce the space occupancy of the classic Patricia trie from $O(n \log n)$ to $O(n \log \ell)$ bits. Of course it is $\ell = \Omega(\log_2 n)$ because all n binary strings are distinct.

A proper sampling of $\mathcal{S}$. Recall that the strings in $\mathcal{S}$ are lexicographically sorted. For the sake of presentation we assume that $s_1 = 0^+$ and $s_n = 1^+$, so that P is lexicographically included in $\mathcal{S}$, and let n be a multiple of $\log n$.

We partition $\mathcal{S}$ into $g = n/\log n$ 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 $i = 0, 1, 2, \ldots, g-1$. We then construct a subset of $\mathcal{S}$, call it $\mathcal{S}'$, that consists of $2g = \Theta(\frac{n}{\log n})$ strings obtained by taking the smallest (first) and the largest (last) string in each of these groups. In some sense this *sampling* process recalls the one adopted to design the String B-tree [9], but it is restricted to just two levels, it assumes that the block size is $\log n$, and it requires to design a different prefix-search because the strings of $\mathcal{S}$ are not available.

The data structures. For the sake of presentation, let us forget the I/O-issues and concentrate on the design of a solution for the RAM model. We denote by $[s_l, s_r]$ the two strings in $\mathcal{S}$ which delimit the range of strings prefixed by P and thus the integer-pair (l, r) is the solution to our weak-prefix search problem. We recall that these two strings may be arbitrary in the case that P does not prefix any string of $\mathcal{S}$. Given our notation we construct two types of Patricia Tries:

- $\mathcal{PT}'$ is the Patricia Trie built over the strings in $\mathcal{S}'$ with the speciality that we store in each node u of $\mathcal{PT}'$ a fingerprint of $O(\log n)$ bits computed for the string-prefix $s(u)$ according to the Karp-Rabin's scheme [6].
- For each edge $e = (u, v)$ of $\mathcal{PT}'$ we build the Patricia trie $\mathcal{PT}_e$ which indexes a group of $O(\log n)$ strings defined as follows. Assume that each node v of

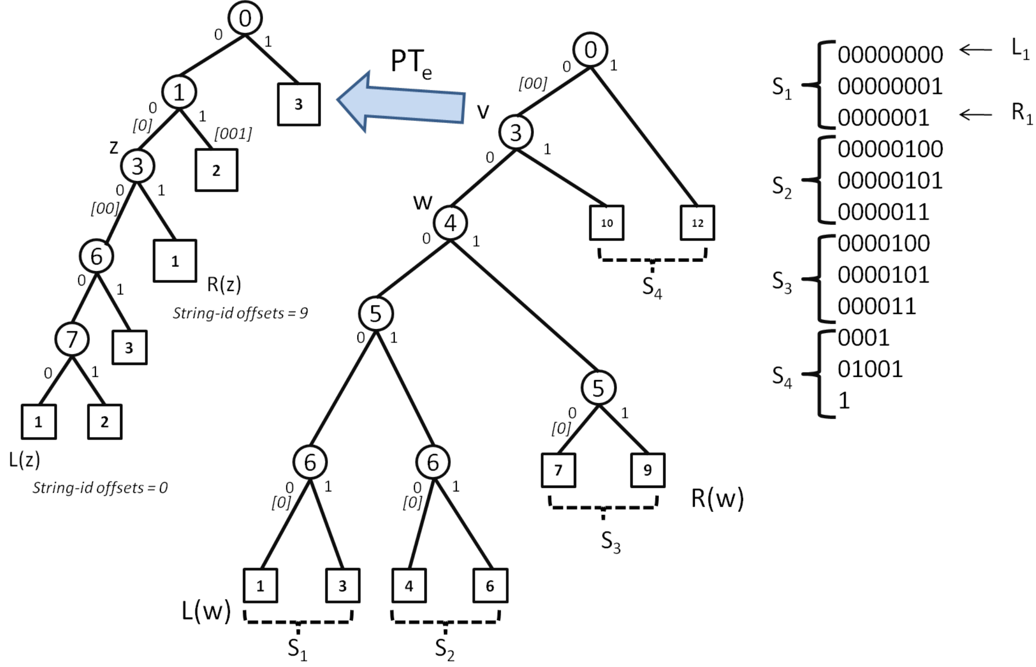

Fig. 2. *The Patricia Trie built on the subset $\mathcal{S}'$, in which each group is formed by* 3 *strings. On the left it is shown $\mathcal{PT}_e$, where e is the edge incident into v, with its leaves encoded* relatively *to the first two string-ids of groups $\mathcal{S}_1$ and $\mathcal{S}_4$, namely* 1 *and* 10*. The figure also indicates the leftmost $s_{L(w)}$ and the rightmost $s_{R(w)}$ strings descending from node w.*

$\mathcal{PT}'$ 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 the Patricia trie $\mathcal{PT}_e$ is built on the string set $\mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$. We have a total of $O(n/\log n)$ such Patricia tries.

An example of this two-level indexing data structure is given in Figure 2, where it is illustrated the Patricia trie $\mathcal{PT}'$ and the Patricia trie $\mathcal{PT}_e$ corresponding to the edge incident into the node v and indexing the groups $\mathcal{S}_1$ and $\mathcal{S}_4$.

The following Lemma estimates the space occupancy of the proposed data structure, showing that it matches the lower bound proved in [2]. Hence it is space optimal.

Lemma 1 *The data structures $\mathcal{PT}'$ and $\mathcal{PT}_e$, for all edges e in $\mathcal{PT}'$, occupy a total of $O(n \log \ell)$ bits.*

Proof: Since we have $O(n/\log n)$ nodes in $\mathcal{PT}'$, and hence $O(n/\log n)$ potential string-prefixes to be compared against P, we design a KR-fingerprint that takes $O(\log n)$ bits and ensures distinct substrings to collide with polynomially small probability. The Patricia trie $\mathcal{PT}'$ consists of $O(n/\log n)$ nodes, each storing: one KR-fingerprint of $O(\log n)$ bits, one string length encodable

in $O(\log \ell)$ bits on average, and two child pointers of $O(\log n)$ bits each. Since $\ell = \Omega(\log_2 n)$ (see above), the Patricia trie $\mathcal{PT}'$ takes $O(n)$ bits overall.

As far as the Patricia trie $\mathcal{PT}_e$ is concerned, we notice that the strings in $\mathcal{S}_{L(v)}$ or $\mathcal{S}_{R(v)}$ are contiguous in $\mathcal{S}$, so we can encode their string-ids relatively to the first string-id of each group, thus using $O(\log\log n)$ bits for each of them. $\mathcal{PT}_e$ indexes $O(\log n)$ strings, so it occupies $O((\log n) \times (\log \ell + \log\log n))$ bits on average. Since the number of edges in $\mathcal{PT}'$ is $O(n/\log n)$, we have such number of Patricia tries $\mathcal{PT}_e$s, which therefore take $O(n \log \ell)$ bits overall. $\square$

To design our algorithm for supporting the weak prefix-search of P in $\mathcal{S}$, and prove its correctness, we need some few useful properties. The following one can be easily derived from [9], its correctness has been sketched in Section 2.

Fact 1 (Ferragina-Grossi, 1999) *Let $\mathcal{PT}_X$ be the Patricia trie built on a string set $\mathcal{X}$. The execution of the blind search for a pattern P over $\mathcal{PT}_X$ stops at a node u such that either the string $s(u)$ is prefixed by P, or P does not prefix any string in $\mathcal{X}$.*

The impact of this Fact cannot be overestimated, in the sense that it can be used to perform a weak-prefix search over the (contiguous) groups of strings indexed by the $\mathcal{PT}_e$s; but it cannot be used over the Patricia trie $\mathcal{PT}'$ to determine the lexicographic position of P in $\mathcal{S}'$, as it was done originally in [9]. This would be crucial to determine the subgroup of (contiguous) strings where the prefix-search should continue. But the set $\mathcal{S}'$ is a sampling of $\mathcal{S}$ and these strings are unavailable, so we cannot compute the lcp between P and the string $s(u)$ to drive the selection of the lexicographic position of P in that set. Even the KR-fingerprints, which are available for each edge of $\mathcal{PT}'$, cannot be deployed for this purpose: they allow to check the (mis-)match of two substrings but not their lexicographic order. An example is given in Figure 2 for the pattern $P = 01001$. The blind-search reaches the right child of the node w, thus we would be driven to take $\mathcal{S}_3$ as the group of strings potentially prefixed by P. This is clearly wrong because $P = s_{11}$, so P lies in $\mathcal{S}_4$ and does not prefix any string of $\mathcal{S}_3$.

We circumvent this drawback by devising a new structural property of Patricia Tries. This property is tricky because it concentrates only on the two strings delimiting the result range, which can in turn be derived as a whole by deploying the ordering of $\mathcal{S}$'s strings. Of course the ordering allows also to efficiently count the number of strings answering the prefix-search query by just one arithmetic operation on the string-ids of those range extremes.

Fact 2 *If P prefixes some strings in $\mathcal{S}$, then it does exist an edge $e = (u, v)$ in $\mathcal{PT}'$ such that s_l and s_r can be identified by looking at the string set $\mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$.*

Proof: For ease of exposition we denote by $s_{L(i)}$ and $s_{R(i)}$ the leftmost and

the rightmost strings sampled from the group $\mathcal{S}_i$. Given that P prefixes some of the strings in $\mathcal{S}$, we consider two cases: (a) P does not prefix anyone of the strings in $\mathcal{S}'$, and (b) P prefixes at least one string of $\mathcal{S}'$. Case (a) implies that the solution-range $[s_l, s_r]$ is totally included in one group $\mathcal{S}_i$, so $s_{L(i)} < s_l \leq s_r < s_{R(i)}$. Case (b) implies that the solution-range $[s_l, s_r]$ spans one or more groups, say $\mathcal{S}_x, \mathcal{S}_{x+1}, \ldots, \mathcal{S}_y$.

The Fact is trivially true in Case (a) because it is enough to take as edge e the one incident into the leaf $v = s_{L(i)}$ (or equivalently, $v = s_{R(i)}$) so that $\mathcal{S}_{L(v)} = \mathcal{S}_{R(v)}$. Recall that the first and the last strings of each group $\mathcal{S}_i$ belong to $\mathcal{S}'$, and thus they are indexed in $\mathcal{PT}'$.

The proof for the Case (b) is more elaborate. We assumed that $s_l \in \mathcal{S}_x$ and $s_r \in \mathcal{S}_y$, with $x \leq y$, so we need to prove that there is an edge $e = (u, v)$ in $\mathcal{PT}'$ such that $\mathcal{S}_{L(v)} = \mathcal{S}_x$ and $\mathcal{S}_{R(v)} = \mathcal{S}_y$. Given that P prefixes all strings in $[s_l, s_r]$, it prefixes all strings of $\mathcal{S}_{x+1}, \ldots, \mathcal{S}_{y-1}$. It also prefixes $s_{R(x)}$, which lies on the right of s_l (possibly it is $s_l = s_{R(x)}$), and it prefixes $s_{L(y)}$, which lies on the left of s_r (possibly it is $s_{L(y)} = s_r$).

We now concentrate on s_l because the proof for s_r is symmetric, and we prove that $s_l \in \mathcal{S}_{L(v)}$ for some node v in $\mathcal{PT}'$. This implies that $\mathcal{S}_{L(v)} = \mathcal{S}_x$, as claimed above. The proof distinguishes two sub-cases: (b.1) $s_{L(x)} < s_l$ and so P does not prefix $s_{L(x)}$ which is on the left of s_l, (b.2) $s_{L(x)} = s_l$ and thus P prefixes $s_{L(x)}$.

Case (b.1) implies that $lcp(s_{L(x)}, s_l) < p$. We know that P prefixes the rightmost string of $\mathcal{S}_x$, hence $s_{R(x)}$, which occurs in the set $\mathcal{S}'$ as well as $s_{L(x)}$ does. So the blind search over $\mathcal{PT}'$, which indexes the strings in $\mathcal{S}'$, will exhaust P stopping at a node v such that $|s(v)| \geq p$. It is not difficult to convince yourself that $s_{R(x)}$ descends from v but its adjacent string on-the-left in $\mathcal{S}'$, namely $s_{L(x)}$, does not. Therefore the leftmost string in $\mathcal{S}'$ descending from v is $s_{R(x)}$. So we can conclude that $\mathcal{S}_{L(v)} = \mathcal{S}_x$, and thus $s_l \in \mathcal{S}_{L(v)}$.

Case (b.2) follows a similar argument to prove that $s_l = s_{L(x)}$ descends from v but its adjacent string on-the-left in $\mathcal{S}'$, namely $s_{R(x-1)}$ (because of the sampling), does not descend from v given that P does not prefix it. So the leftmost string in $\mathcal{S}'$ descending from v is $s_{L(x)}$, and thus again $\mathcal{S}_{L(v)} = \mathcal{S}_x$, and hence $s_l \in \mathcal{S}_{L(v)}$. □

In other words, Fact 2 relaxes the requirement that $\mathcal{PT}'$ can identify the lexicographic position of P in $\mathcal{S}'$, and aims for an *approximate solution* of it: it shows that the position of P in the original set $\mathcal{S}$ can be determined within a distance $O(\log n)$. It is the second round of the weak-prefix search procedure, executed on the Patricia trie $\mathcal{PT}_e$ attached to the edge e, that will complete correctly the prefix-search operation.

Before detailing the algorithm that implements the whole weak prefix-search, we refer the reader to Figure 2 for an illustrative example of the statement of Fact 2. For the Case (a) of the above proof, take the pattern $P = 0100$ which prefixes the single string s_{11} internal into $\mathcal{S}_4$. The edge identified by Fact 2 is the one incident in $s_{L(4)} = s_{10}$, so the candidate group of strings to be searched for identifying s_l and s_r is $\mathcal{S}_4$. For the Case (b), take $P = 00$ which prefixes $[s_1, s_{10}]$. In this case the edge e is the one connecting the root of $\mathcal{PT}'$ to v. It identifies the groups $\mathcal{S}_1 \cup \mathcal{S}_4$ and indeed $s_1 \in \mathcal{S}_1$ and $s_{10} \in \mathcal{S}_4$. Both groups are indexed by the Patricia Trie $\mathcal{PT}_e$ shown in the figure.

The weak prefix-search in the RAM model. Now we are ready to show how the Patricia Trie $\mathcal{PT}'$ can be used to efficiently identify the edge e characterized by Fact 2, and how the search proceeds in $\mathcal{PT}_e$ to detect s_l and s_r among the strings of $\mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$.

Subtly enough, the identification of edge e specified in Fact 2 is not immediate and cannot proceed by just applying verbatim the blind-search procedure of Fact 1. As an example, assume that P does not prefix any string in $\mathcal{S}'$ but it prefixes one string in some sub-group $\mathcal{S}_i$. Given that the Patricia trie $\mathcal{PT}'$ is built over $\mathcal{S}'$, the blind search could lead to a node which is completely far away from the leaf $s_{L(i)}$ or $s_{R(i)}$ we are searching for. Figure 2 shows this dangerous case for $P = 0100$. Here P prefixes $s_{11} \in \mathcal{S}_4$ but no string in $\mathcal{S}'$ is prefixed by P. The path followed by the blind search in $\mathcal{PT}'$ leads to the node w and thus the application of Fact 2 to the edge $e = (v, w)$ would lead to search for s_l and s_r in $\mathcal{S}_1 \cup \mathcal{S}_3$. This is incorrect. The problem here is that the mismatch-bit between P and the first traversed edge resides at position $lcp(P, s(v)) = 1$ which is internal into the edge label and thus has not been compared by the blind search. The next compared bit, namely $P[4]$, matches the branching bit leading to w and thus drives the search far from the node v, hence far from $\mathcal{S}_4$.

The problem here arises because $\mathcal{PT}'$ contains a "reduced" set of branching nodes (wrt $\mathcal{PT}_{\mathcal{S}}$), which are indeed the lowest-common-ancestors of the leaves associated to the sampled strings of $\mathcal{S}'$. So the path leading to $s_{L(i)}$ is surely followed by the blind-search procedure up to the first $lcp(P, s_{L(i)}) < p$ bits, but then it may diverge from that path when matching the pattern's bits following $lcp(P, s_{L(i)})$.

In order to circumvent this problem we have to empower the blind-search procedure for detecting the edge of the mismatch bit. The pseudo-code in Figure 3 details our approach that hinges on the deployment of the fingerprints $f(u)$ available at each node u of $\mathcal{PT}'$. These fingerprints maintain a succinct encoding of the substring $s(u)$, taking $O(\log n)$ bits each. We highlight that, for efficiency reasons, $f(u)$ does not represent the label of the edge leading to

(1) Compute the Karp-Rabin's fingerprint of all prefixes of the pattern P, according to the function $f()$ used for the nodes in $\mathcal{PT}'$.

(2) Execute a blind search for P over the Patricia trie $\mathcal{PT}'$. Each time a node v is reached, we check whether $f(v)$ equals the fingerprint of the corresponding pattern prefix $f(P[1, |s(v)|])$. If the two fingerprints match and $|s(v)| < p$, the blind search proceeds branching out of v with the bit $P[|s(v)| + 1]$, otherwise we stop at v and call e the edge of $\mathcal{PT}'$ entering into v.

(3) Repeat the blind-search procedure over $\mathcal{PT}_e$ and call v' the node where it stops. Return $s_l = L(v')$ and $s_r = R(v')$.

Fig. 3. *The weak-prefix search algorithm.*

u, but it rather represents the entire string-prefix $s(u)$, as spelled out by the root-to-u path. This way, we need to compute and store only $O(p)$ fingerprints for P, which can be done in linear time according to the Karp-Rabin's scheme.

It is evident in the pseudo-code of Figure 3 that fingerprints will be equal for all the ancestors of the node u (and thus for all pattern prefixes of length $\leq |s(u)|$), after that they'll be different with high probability. The crucial twist here is that we are not able to identify the position of the mismatching bit, we are only able to identify the edge $e = (u, v)$ containing that mismatching bit. This is enough to conclude that P lies lexicographically either to the left of the subtree descending from v (hence to the left of $L(v)$) or to its right (hence to the right of $R(v)$). This is what has been stated in Fact 2 and implemented in Step 2. Finally, the fact that $\mathcal{PT}_e$ indexes the two contiguous ranges of strings $\mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$, together with Fact 1 guarantee that Step 3 correctly identifies s_l and s_r among the strings of $\mathcal{S}_{L(v)} \cup \mathcal{S}_{R(v)}$ as the leftmost/rightmost descendants of node v' in $\mathcal{PT}_e$.

Overall the search for $P[1, p]$ percolates two downward paths of length at most p each, one in $\mathcal{PT}'$ and the other in $\mathcal{PT}_e$. We have therefore proved the following

Theorem 2 *The combination of $\mathcal{PT}'$ and the set of $\mathcal{PT}_e$ solves the weak-prefix search problem in $O(p)$ time and $O(n \log \ell)$ bits of space. The search is correct with high probability because of the use of the Karp-Rabin's fingerprints.*

A note is in order at this point. The search uses some local information stored at each node and taking $O(\log n + \log \ell) = O(w)$ bits. This node information can thus be stuffed into one single machine word. This consideration is crucial when partitioning the nodes of the Patricia tries among the disk pages (next section), because it allows to conclude that the node information can be safely packed with the node itself independently of the length of the string it represents.

4 I/O-packing of the Patricia Tries

Packing trees of t nodes into pages of size B is difficult if there is no additional restriction either on the tree structure or on the type of tree traversals. Surprisingly enough, even if we restrict the tree traversals to root-to-leaf paths of length L, the type of I/O-bounds we are aiming for— namely $O(L/B + \log_B t)$— cannot be guaranteed in general [7]. But when the path lengths are short do exist efficient disk-mappings:

Fact 3 *Each Patricia trie $\mathcal{PT}_e$ can be mapped onto disk pages so that any root-to-leaf path can be traversed in $O(\log_B n)$ I/Os.*

Proof: It is enough to observe that $\mathcal{PT}_e$ consists of $O(\log n)$ nodes, so that any root-to-leaf path has length $L = O(\log n)$. From the result in [7], there exist a disk-mapping taking $\frac{L}{\log B} = O(\log_B n)$ I/Os to traverse these paths. □

Extending this I/O-bound to all other path lengths, and thus to all possible pattern lengths, is not easy and needs to dig into the structural properties of tries and prefix searches as we do next. Specifically, we will discuss two approaches to the I/O-packing of Patricia tries on disk memories. The first approach is based on the concept of *skeleton trie* introduced in [13] and leads to a solution working in the external-memory model, where the parameters B, M are known at time the data structure is built; the second approach is based on the concept of *centroid trie* introduced in [3] and leads to a cache-oblivious solution achieving the same I/O-bounds as above, but working in the case that the parameters B, M are unknown at time the data structure is built. Although the latter result subsumes the former in terms of performance, we describe both of them because the technicalities are novel and thus interesting in themselves. There is also a subtlety that led us to state both results: the cache-oblivious solution is restricted to work only in the (significant) case that $p \leq M$; the external-memory solution works everywhere. [3]

4.1 On the skeleton tree

In [13] the authors observe that a tree can be decomposed into subtrees whose total number of nodes may be larger than the disk-page size B, but whose non-unary nodes are $O(B)$ and thus can be packed into one disk page. This

[3] This is due to the way fingerprints are checked. In the external-memory approach they are computed and checked from the empty prefix to the entire P; in the cache-oblivious approach the checking depends on the way the traversal of the centroid tree proceeds.

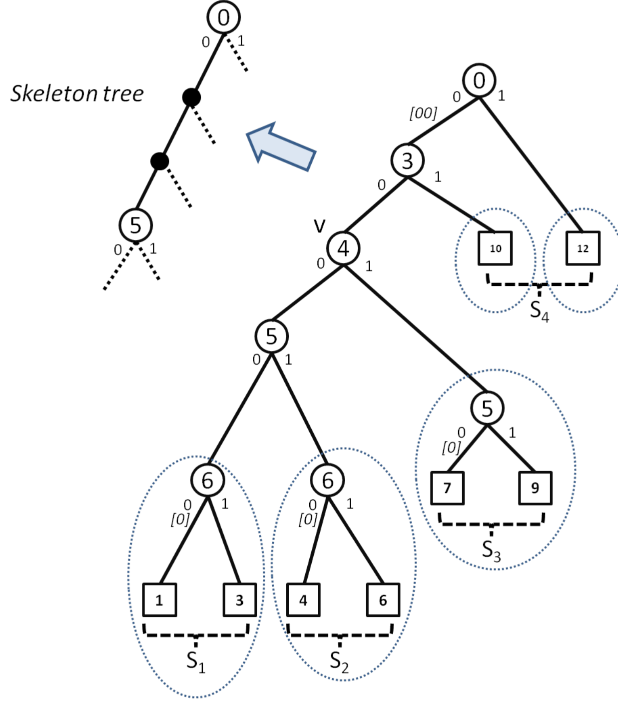

Fig. 4. *The Skeleton Trie derived for the sampled Patricia Trie of Figure 2, assuming a disk-page size of* $B = 3$ *nodes. Dotted circles denote the single components mapped to one single page. To be noticed that the topmost subtree would be larger than* B *if stored as is. Instead its skeleton tree consists of only two nodes (as shown in the top-left part of the figure); the other nodes (here denoted as bullets) and their edges are stored in auxiliary pages according to the scheme described in the text.*

means that the remaining $\Omega(B)$ nodes form (unary) paths that need to be managed properly, anyway differently from what has been proposed in that paper where the indexed strings were available. For the sake of completeness we recall the partitioning scheme of [13], and adapt its description from the suffix tree (which was the tree to be partitioned in that paper) to our Patricia Trie $\mathcal{PT}'$.

For a node v in $\mathcal{PT}'$, the number of leaves in the subtree under v is referred to as $size(v)$. If v is a leaf node then $size(v) = 1$. The rank of node v, denoted $rank(v)$, is i if and only if $C^i \leq size(v) < C^{i+1}$, for some integer constant C fixed in advance. We partition $\mathcal{PT}'$ into subtrees such that the nodes u and v belong to the same partition if all nodes on the undirected path between u and v have the same rank. The authors of [13] called this subtree the *skeleton tree*, see Figure 4 for an example.

The rank of a component Q is the same as the rank of the nodes in Q, i.e. $rank(Q) = rank(v)$ for any $v \in Q$. In particular we say that node $v \in Q$ is a *leaf* of Q if and only if none of v's children in $\mathcal{PT}'$ belongs to Q; a node u is termed the *root* of Q if and only if u's parent is not a node of Q. Let us denote by $\mathcal{T}_Q$ the subtree obtained by restricting the component Q to its leaves and

to its non-unary nodes.

Fact 4 (Ko-Aluru, 2006) *The subtree $\mathcal{T}_Q$ consists of at most $C-1$ leaves and at most $2C-2$ non-unary nodes.*

Although the size of $\mathcal{T}_Q$ is upper bounded by $O(C)$, this bound does not hold for the number of non-unary nodes lying in Q, which can be up to $C^{i+1}-C^i = \Omega(C)$. By setting $C = \Theta(B)$, we can stuff $\mathcal{T}_Q$ in one disk page, but we need possibly many more pages to pack these unary nodes. Nevertheless they form unary paths that can thus be lied down contiguously and linearly on disk, so that their traversal takes an optimal number of I/Os. By orchestrating skeleton trees, unary paths and the special properties of the blind-search procedure over Patricia tries we will obtain the result stated in Theorem 3 below.

As proved in [13], any root-to-leaf path in $\mathcal{PT}'$ is decomposed in no more than $O(\log_B n)$ disk pages. In that paper authors explain how to prefix search for P by using skeleton tries and the strings of $\mathcal{S}'$. In our context we do not have $\mathcal{S}'$. So our algorithmic idea is to proceed in two phases, which alternate at each component Q: first we prefix-search P in the skeleton trie $\mathcal{T}_Q$, matching its branching bits and the KR-fingerprints attached to its nodes; this identifies an edge where either a mismatch occurs or that leads to another component. Since each edge represents a *single* unary-path, it is then accessed I/O-optimally, and allows to move from Q to the next component in $\mathcal{PT}'$.

In detail, the first phase determines an edge (z, w) of $\mathcal{T}_Q$ such that $s(z)$ prefixes P (and this has been checked by means of the KR-fingerprint $f(z)$) and either w lies outside the current skeleton tree $\mathcal{T}_Q$, or w belongs to the tree but $s(w)$ and P mismatch somewhere after their $(|s(z)|+1)$-th bit. So the mismatch is on the unary path from z to w; this path is lied down contiguously on disk with all the information (namely branching bits and fingerprints) needed to perform a prefix-search over it, as the one described in Step 2 of Figure 3. This means that the traversal of this path takes $O(1 + L/B)$ I/Os where L is the number of scanned nodes. It is clear that bits matched in P are never re-scanned, so the only I/O-waste consists of the additional $O(1)$ I/Os paid to retrieve from disk the skeleton trie $\mathcal{T}_Q$ and the first page of the unary path where the mismatch lies.

At most $O(\log_B n)$ components are traversed in a prefix search [13], so the following result holds. [4]

Theorem 3 *The weak-prefix search problem can be solved in the external-memory model by using $O(n \log \ell)$ bits of space and $O(p/B + \log_B n)$ I/Os. The correctness holds with high probability.*

[4] Recall Fact 3 which addressed the disk-mapping of small Patricia tries $\mathcal{PT}_e$.

We resort a known decomposition of trees proposed in [3] for the static cache-oblivious String B-tree. We apply it to our Patricia Trie $\mathcal{PT}'$, and then modify accordingly the blind-search procedure of Figure 3.

Let us recall how to re-shape $\mathcal{PT}'$ via the centroid decomposition. The key idea is that there is a *centroid node* z in $\mathcal{PT}'$ that has at least $t/3$ and at most $2t/3$ descendants, where $t = |\mathcal{PT}'| = O(n/\log n)$. The centroid tree of $\mathcal{PT}'$ is obtained by making z the root and attaching as z's children the recursively defined centroid trees of z's up and down tries. At every level of the recursion we eliminate a constant fraction of nodes from consideration, so the centroid tree has depth $O(\log n)$. Thus, as observed for the small tries $\mathcal{PT}_e$ (see Fact 3), there exists a cache-oblivious packing for which any root-to-leaf path traverses $O(\log_B n)$ disk pages (hence I/Os). The packing is executed not only on the tree-structure of the centroid tree, but also on the fingerprinting information that is associated to the nodes of $\mathcal{PT}'$.

Figure 5 shows an example of centroid tree computed for the Patricia Trie of Figure 1, indexing the whole set of strings $\mathcal{S}$. Each internal node z is associated with its string $s(z)$, which is shown explicitly only for the sake of presentation but that is stored on disk in terms of its KR-fingerprint $f(z)$ using one single machine word. Each internal node has a *solid* edge linking it to the root of its down trie (the leaves of which have $s(z)$ as a prefix), and a *dotted* edge linking it to the root of its up trie (the leaves of which do not have $s(z)$ as a prefix). Moreover it maintains a pointer to its leftmost and its rightmost descending leaves within the entire $\mathcal{PT}'$ (not shown in the picture). In Figure 5 we use circles to denote internal nodes of $\mathcal{PT}'$ and squares to denote leaves of $\mathcal{PT}'$. Notice that leaves of $\mathcal{PT}'$ can become internal nodes of the centroid tree, and vice versa.

Prefix-searching over the centroid tree. We are left with showing how Step 2 of Figure 3 is implemented (recall that Step 3 acts on $\mathcal{PT}_e$ which takes $O(\log_B n)$ I/Os to be percolated, according to Fact 3). Let z be the node of the centroid tree currently visited, initially z is the root. We determine whether $s(z)$ is a prefix of P by comparing the corresponding fingerprints. If it does, then we follow the solid edge, otherwise we follow the dotted edge. The ratio is that, in the case of a match, the prefix search must proceed in the subtrie of $\mathcal{PT}'$ that descends from z (previously called down-trie) and containing all strings of $\mathcal{S}'$ that have $s(z)$ as a prefix; otherwise, the prefix search does not pass through z but lies into its up-trie. It should be evident that we are doing a sort of *binary search* over the entire structure of $\mathcal{PT}'$ which eventually identifies the deepest node u whose fingerprint $f(u)$ equals the fingerprint of the corresponding pattern prefix $f(P[1, |s(u)|])$. Now if $|s(u)| < p$, we take v

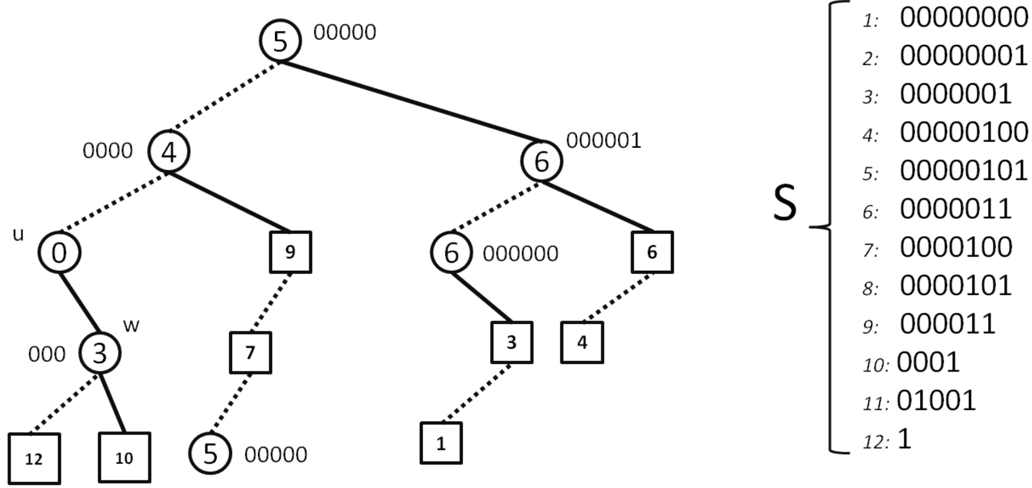

Fig. 5. *The centroid tree of the entire Patricia Trie $\mathcal{PT}_{\mathcal{S}}$ of Fig.1.*

as the child of u in $\mathcal{PT}'$ whose branching bit equals $P[|s(u)|+1]$; otherwise it is $|s(u)| = p$ and thus we can set $v = u$. This node v is exactly the one identified in Step 2 of the pseudo-code of Figure 3.

As an illustrative example, take $P = 0100$ and start matching the centroid-tree's nodes in Figure 5 from the root. We percolate the leftmost downward path up to node u, since its ancestors have longer labeling substrings which therefore do not prefix-match P. This means that we will follow the dotted edges until we reach node u whose string $s(u)$ is empty and thus prefix-matches P. At this point we follow the solid child w of u that does not prefix-match P. Its dotted child is a leaf which does not prefix-match P too. Given that u is the deepest node in $\mathcal{PT}'$ whose label prefix-matches P, we jump[5] on u's "copy" in $\mathcal{PT}'$ and take as child v the one whose branching bit is $P[1] = 0$. We have therefore identified the edge $e = (u, v)$ of Fact 2, so that the prefix-search for P can be continued in the proper $\mathcal{PT}_e$ according to Fact 3.

Theorem 4 *The weak-prefix search problem can be solved in the cache-oblivious model by using $O(n \log \ell)$ bits of space and $O(p/B + \log_B n)$ I/Os. The correctness holds with high probability.*

5 A distribution-aware solution

Our two-level indexing scheme can be turned into one that guarantees *query-distribution* awareness, so changing the term $O(\log_B n)$ into a term that depends on the probability of querying the pattern P, such as $O(\log_B \frac{1}{\wp(P)})$. This

[5] The "jump" is logical since v can bring the two pointers within its occurrence in the centroid tree, thus saving one I/O, and the duplicate storage of $\mathcal{PT}'$.

improvement can be significant in the case that some patterns are queried more frequently than others.

To define correctly the scenario, let us start from a probability distribution over the strings of $\mathcal{S}$ (and thus over the leaves of $\mathcal{PT}_{\mathcal{S}}$), and then derive a distribution over their prefixes: for every node $v \in \mathcal{PT}_{\mathcal{S}}$, we define $\wp(v)$ as the sum of the probabilities of the descending leaves (i.e. strings having prefix $s(v)$). We can adapt the centroid decomposition of the previous section in order to work on the probability mass of the nodes in $\mathcal{PT}_{\mathcal{S}}$ rather than on the number of their descending leaves. This guarantees that every prefix-search for a pattern $P[1,p]$ in $\mathcal{PT}_{\mathcal{S}}$ takes $O(p/B + \log_B \frac{1}{\wp(P)})$ I/Os.

But we have to be careful with the very rare strings s such that $\wp(s) < 1/n$. For these strings the above I/O-bound is worse than the one we achieved in Theorems 3 and 4. So we need a different approach which consists of dividing the nodes of $\mathcal{PT}_{\mathcal{S}}$ into two subsets: one composed by rare nodes, whose probability is $< 1/n$, and the other set formed by the rest. It is clear that the rare nodes form subtrees of $\mathcal{PT}_{\mathcal{S}}$ that lead to trie leaves; whereas the un-rare nodes form a single trie rooted in the root of $\mathcal{PT}_{\mathcal{S}}$. We store the former subtries by using the scheme of Theorems 3 and 4, which is UN-aware of the underlying distribution. Conversely, we store the single trie of un-rare nodes by using the distribution-aware approach sketched above.

The prefix-search for P starts in this latter trie and stops, as usual, at an edge (u, v) such that $s(u)$ prefixes P but either v belongs to the top trie (and thus is un-rare as u) or it roots a rare subtrie (and thus it is rare too). In the former case the search stops at this edge, and since it has acted on the distribution-aware scheme, it takes $O(|s(u)|/B + \log_B 1/\wp(u))$ I/Os. Since $s(u)$ prefixes P it is $|s(u)| \leq p$ and $\wp(u) \geq \wp(P)$. Moreover we know that u is an un-rare node, hence $\wp(u) \geq 1/n$. Consequently this case takes $O(p/B + \log_B \min\{n, 1/\wp(P)\})$ I/Os.

In the other case that v is rare, the search must continue in the sub-trie of rare nodes rooted in v. Since this trie is lied down on disk by using the distribution UN-aware scheme, this second prefix-search takes $O(p/B + \log_B n)$ I/Os. It is clear that this second case may occur only if P is a rare string, so the total I/O-bound for the prefix search can be again written as $O(p/B + \log_B \min\{n, 1/\wp(P)\})$ I/Os.

A final note is in order at this point. The above query-distribution aware approach must be applied to both $\mathcal{PT}'$ and to all $\mathcal{PT}_e$ in our two-level indexing scheme to get the space bound of $O(n \log \ell)$ bits.

Theorem 5 *The* weighted *weak-prefix search problem, for which a query-distribution is known in advance, can be solved in the cache-oblivious model*

using $O(n \log \ell)$ bits of space and $O(p/B + \log_B \min\{n, \frac{1}{\wp(P)}\})$ I/Os. The correctness holds with high probability.

6 Conclusions

Our algorithms can be made *deterministic* in the reasonable applicative scenario (mentioned in [2]) that the strings can be stored on disk. We need only to store on disk the sampled set $\mathcal{S}'$ and use just one additional disk I/O to determine v in Step 2 (in practice we argue that $p/B \leq 1$). Compared to the String B-tree [9], we are performing the same number of I/Os but with a smaller storage cost of $o(n \log n)$ bits in addition to the indexed strings, and without the need to keep the whole string set $\mathcal{S}$ on disk.

Following [3] we could further squeeze the space occupancy of our solutions by using the *locality preserving front-coding* scheme introduced in that paper. This allows to achieve space close to the *front-coding* of the set of strings $\mathcal{S}'$, but without penalizing asymptotically the cost of one string decoding. The combination of our results with the LPFC-scheme would give a win-win situation because we would save space both for the storage of the indexing data structure and for the storage of the indexed strings.

We conclude this paper by foreseeing a challenging scenario that actually goes beyond the weak prefix-search problem investigated in these sections. Energy efficiency has now become a key issue for servers and data center operations because "The cost of power and cooling is likely to exceed that of hardware" [1]. It is commonly believed that improvements in the energy efficiency of IT devices will be much more dramatic, and eventually have much greater impact, than in other areas of technology. Much prior work was concerned with electrical and systems engineering, with a relatively smaller amount of investigation in the core areas of Computer Science. Recently [5,12] proposed the study of a "Science of Power Management" which should develop theoretical models of power-performance tradeoffs at multiple levels of granularity, and design canonical algorithmic techniques to attain better results than the ones obtainable via engineering system knobs and parameters. Following this proposal, we addressed in [8] the energy-issues that arise in the design of compressed data structures, and illustrated some preliminary experimental results showing that a careful orchestration of patterns of memory accesses and computation can lead to "dramatic improvements" in the energy consumption of IT systems, as the ones aimed for by [5,12]. We believe that the data structures presented in this paper are concrete examples of this new line of research: they could be used for implementing a prefix-search black-box in modern smart-phones. Experiments over real applicative scenarios are needed to validate this guess.

Acknowledgements

I thank Djamal Belazzougui and the anonymous referees of CPM '11 for their useful comments.

References

[1] L.A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[2] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *Procs 18th European Symposium on Algorithms (ESA)*, volume 6346 (part I), Lecture Notes in Computer Science, Springer, pages 427–438, 2010.

[3] M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious String B-trees. In *Procs 25th ACM Symposium on Principles of Database Systems (PODS)*, pages 233–242, 2006.

[4] G. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Procs 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 581-590, 2006.

[5] K.W. Cameron, K. Pruhs, S. Irani, P. Ranganathan, and D. Brooks. Report of the science of power management workshop. NSF Report, August 2009.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[7] E.D. Demaine, J. Iacono, and S. Langerman. Worst-case optimal tree layout in a memory hierarchy. Manuscript, available on `arXiv:cs.DS/0410048`, 2004.

[8] P. Ferragina. Data structures: Time, I/Os, entropy, joules! In *Procs 18th European Symposium on Algorithms (ESA)*, volume 6347 (part II), Lecture Notes in Computer Science, Springer, pages 1–16, 2010.

[9] P. Ferragina and R. Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *Journal of the 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 *Procs 27th ACM Symposium on Principles of Database Systems (PODS)*, pages 181–190, 2008.

[11] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Procs 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[12] K. Kant. Toward a science of power management. *IEEE Computer*, 42(9):99–101, 2009.

[13] P. Ko and S. Aluru. Obtaining provably good performance from suffix trees in secondary storage. In *Procs 17th Symposium on Combinatorial Pattern Matching (CPM)*, volume 4009, Lecture Notes in Computer Science, Springer, pages 72–83, 2006.

[14] A. Maheshwari and N. Zeh. A survey of techniques for designing i/o-efficient algorithms. In *Algorithms for Memory Hierarchies*, volume 2625, Lecture Notes in Computer Science, Springer, pages 36–61, 2003.

[15] J.S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.