Fast Prefix Search in Little Space, with Applications

Djamal Belazzougui[1], Paolo Boldi[2], Rasmus Pagh[3], and Sebastiano Vigna[2]

[1] Université Paris Diderot—Paris 7, France
[2] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
[3] IT University of Copenhagen, Denmark

Abstract. A *prefix search* returns the strings out of a given collection S that start with a given prefix. Traditionally, prefix search is solved by data structures that are also dictionaries, that is, they actually contain the strings in S. For very large collections stored in slow-access memory, we propose extremely compact data structures that solve *weak* prefix searches—they return the correct result only if *some* string in S starts with the given prefix. Our data structures for weak prefix search use $O(|S| \log \ell)$ bits in the worst case, where ℓ is the average string length, as opposed to $O(|S|\ell)$ bits for a dictionary. We show a lower bound implying that this space usage is optimal.

1 Introduction

In this paper we are interested in the following problem (hereafter referred to as *prefix search*): given a collection of n strings, find all the strings that start with a given prefix p. In particular, we will be interested in the space/time tradeoffs needed to do prefix search in a static context (i.e., when the collection does not change over time).

There is a large literature on indexing string collections. We refer to Ferragina et al. [11,4] for state-of-the-art results, with emphasis on the cache-oblivious model. Roughly speaking, results can be divided into two categories based on the power of queries allowed. As shown by Pătraşcu and Thorup [15] any data structure for bit strings that supports predecessor (or rank) queries must either use super-linear space, or use time $\Omega(\log |p|)$ for a query on a prefix p. On the other hand, it is known that prefix queries, and more generally range queries, can be answered in constant time using linear space [1].

Another distinction is between data structures (typically comparison-based) where the query time grows with the number of strings in the collection, versus those (typically some kind of trie) where the query time depends only on the length of the query string[1]. In this paper we fill a gap in the literature by considering data structures for *weak prefix search*, a relaxation of prefix search, with query time depending only on the length of the query string. In a weak prefix search we have the guarantee that the input p is a prefix of some string in

[1] Obviously, one can also combine the two in a single data structure.

M. de Berg and U. Meyer (Eds.): ESA 2010, Part I, LNCS 6346, pp. 427–438, 2010.
© Springer-Verlag Berlin Heidelberg 2010

the set, and we are only requested to output the ranks (in lexicographic order) of the strings that have p as prefix.

Our first result is that weak prefix search can be performed by accessing a data structure that uses just $O(n \log \ell)$ bits, where ℓ is the average string length. This is much less than the space of $n\ell$ bits used for the strings themselves. We also show that this is the minimum possible space usage for any such data structure, regardless of query time. We investigate different time/space tradeoffs: At one end of this spectrum we have constant-time queries (for prefixes that fit in $O(1)$ words), and still asymptotically vanishing space usage for the index. At the other end, space is optimal and the query time grows logarithmically with the length of the prefix. Precise statements can be found in the technical overview below.

Technical overview. For simplicity we consider strings over a binary alphabet, but our methods generalise to larger alphabets. Our main result is that weak prefix search needs just $O(|p|/w + \log |p|)$ time and $O(n \log \ell)$ space, where ℓ is the average length of the strings, p is the query string, and w is the machine word size. In the cache-oblivious model [12], we use $O(p/B + \log |p|)$ I/Os. For strings of fixed length w, this reduces to query time $O(\log w)$ and space $O(n \log w)$, and we show that the latter is *optimal* regardless of query time. Throughout the paper we strive to state all space results in terms of ℓ, and time results in terms of the length of the actual query string p, because in a realistic setting (e.g., term dictionaries of a search engine) string lengths might vary wildly, and queries might be issued that are significantly shorter than the average (let alone maximum) string length. Actually, the data structure size depends on the *hollow trie size* of the set S—a data-aware measure related to the trie size [13] that is much more precise than the bound $O(n \log \ell)$.

Building on ideas from [1], we then give an $O(|p|/w+1)$ solution (i.e., constant time for prefixes of length $O(w)$) that uses space $O(n\ell^{1/c} \log \ell)$ (for any $c > 0$). This structure shows that weak prefix search is possible in constant time using sublinear space; queries requires $O(|p|/B+1)$ I/Os in the cache-oblivious model.

Comparison to related results. If we study the same problem in the I/O model or in the cache-oblivious model, the nearest competitors are the String B-tree [10] and its cache-oblivious version [4], albeit they require access to the set S. The *static* String B-tree can be modified to use space $O(n \log n + n \log \ell)$; it has very good search performance with $O(|p|/B + \log_B n)$ I/Os per query (supporting all query types discussed in this paper), and its cache-oblivious version guarantees the same bounds with high probability. However, a search for p inside the String B-tree may involve $\Omega(|p| + \log n)$ RAM operations ($\Omega(|p|/w + \log n)$ for the cache-oblivious version), so it may be too expensive for intensive computations. Our first method, which achieves the optimal space usage of $O(n \log \ell)$ bits, uses $O(|p|/w + \log |p|)$ RAM operations and $O(|p|/B + \log |p|)$ I/Os instead. The number of RAM operations is a strict improvement over String B-trees, while the I/O bound is better for large enough sets. Our second method uses slightly more space ($O(n\ell^{1/c} \log \ell)$ bits for any $c > 0$) but features $O(|p|/w + 1)$ RAM operations and $O(|p|/B + 1)$ I/Os.

In [11], the authors discuss very succinct static data structures for the same purposes (on a generic alphabet), decreasing the space to a lower bound that is, in the binary case, the trie size. The search time is logarithmic in the number of strings. As in the previous case, we improve on RAM operations and on I/Os for large enough sets.

The first cache-oblivious dictionary supporting prefix search was devised by Brodal *et al.* [5] achieving $O(|p|)$ RAM operations and $O(|p|/B + \log_B n)$ I/Os. We note that the result in [5] is optimal in a comparison-based model, where we have a lower bound of $\Omega(\log_B n)$ I/Os per query. By contrast, our result, like those in [4,11], assumes an integer alphabet where there is no such lower bound.

Implicit in the paper of Alstrup *et al.* [1] on range queries is a linear-space structure for constant-time weak prefix search on fixed-length bit strings. Our constant-time data structure, instead, uses sublinear space and allows for variable-length strings.

Applications. Data structures that allow weak prefix search can be used to solve the non-weak version of the problem, provided that the original data is stored (typically, in some slow-access memory): a single probe is sufficient to determine if the result set is empty; if not, access to the string set is needed just to retrieve the strings that match the query. By the same means we can answer prefix counting queries. It is also possible to solve range queries with two additional probes to the original data (w.r.t. the output size), improving the results in [1]. We finally show that our results extend to the cache-oblivious model, where we provide an alternative to the results in [5,4,11] that removes the dependence on the data set size for prefix searches and range queries.

Our contributions. The main contribution of this paper is the identification of the weak prefix search problem, and the proposal of a solutions based on techniques developed in [2]. Optimality (in space or time) of the solution is also a central result of this research. The second interesting contribution is the description of *range locators* for variable-length strings; they are an essential building block in our weak prefix search algorithms, and can be used whenever it is necessary to recover in little space the range of leaves under a node of a trie.

2 Notation and Tools

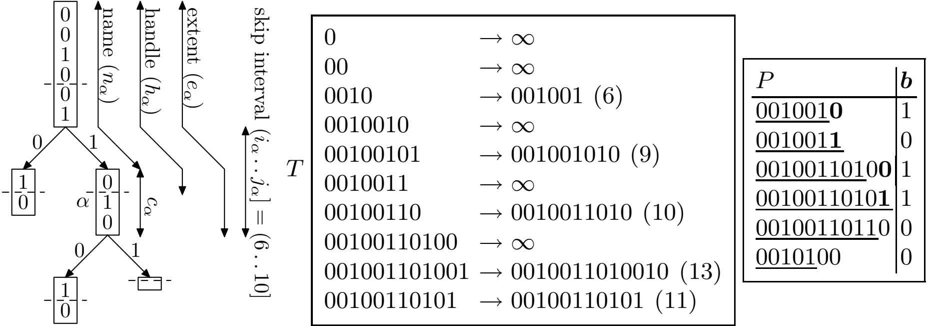
In the following sections, we will use the toy set of strings shown in Figure 1 to display examples of our constructions. We use von Neumann's definition and notation for natural numbers: $n = \{ 0, 1, \ldots, n-1 \}$, so $2 = \{ 0, 1 \}$ and 2^* is the set of all binary strings.

Weak prefix search. Given a prefix-free set of strings $S \subseteq 2^*$, the *weak prefix search* problem requires, given a prefix p of some string in S, to return the range of strings of S having p as prefix; this set is returned as the interval of integers that are the ranks (in lexicographic order) of the strings in S having p as prefix.

Model and assumptions. The model of computation considered in most of the paper is a unit-cost word RAM with word size w. We assume that $|S| = O(2^{cw})$

for some constant c, so that constant-time static data structures depending on $|S|$ can be used. We extend several results also to the cache-oblivious model [12].

Compacted tries. Consider the compacted trie built for a prefix-free set of strings $S \subseteq 2^*$. For a given node α of the trie, we define (see Figure 1):



0	$\to \infty$
00	$\to \infty$
0010	$\to$ 001001 (6)
0010010	$\to \infty$
00100101	$\to$ 001001010 (9)
0010011	$\to \infty$
00100110	$\to$ 0010011010 (10)
00100110100	$\to \infty$
001001101001	$\to$ 0010011010010 (13)
00100110101	$\to$ 00100110101 (11)

P	$\boldsymbol{b}$
001001**0**	1
001001**1**	0
0010011010**0**	1
0010011010**1**	1
00100110110	0
0010100	0

Fig. 1. The trie built on the sample set $\{001001010, 0010011010010, 00100110101\}$, and the associated map and range locator. T maps handles to extents; the corresponding *hollow* z-fast prefix trie just returns the *lengths* (shown in parentheses) of the extents. In the range locator table, we boldface the zeroes and ones appended to extents, and we underline the actual keys (as trailing zeroes are removed). The last two keys are 00100110101^+ and 0010011^+, respectively.

- e_α, the *extent of node* α, is the longest common prefix of the strings represented by the leaves that are descendants of α (this was called the "string represented by α" in [2]);
- c_α, the *compacted path of node* α, is the string stored at α;
- n_α, the *name of node* α, is the string e_α deprived of its suffix c_α (this was called the "path leading to α" in [2]);
- given a string x, we let $\text{exit}(x)$ be the exit node of x, that is, the only node α such that n_α is a prefix of x and e_α is not a proper prefix of x;
- the *skip interval* $(i_\alpha \,..\, j_\alpha]$ associated to α is $(0\,..\,|c_\alpha|]$ for the root, and $(|n_\alpha|-1\,..\,|e_\alpha|]$ for all other nodes.

Data-aware measures. Consider the compacted trie on a set $S \subseteq 2^*$. We define the *trie measure* of S [13] as

$$\mathrm{T}(S) = \sum_\alpha (j_\alpha - i_\alpha) = \sum_\alpha (|c_\alpha| + 1) - 1 = 2n - 2 + \sum_\alpha |c_\alpha| = O(n\ell),$$

where the summation ranges over all nodes of the trie. For the purpose of this paper, we will also use the *hollow*[2] *trie measure*

$$\mathrm{HT}(S) = \sum_{\alpha \text{ internal}} (\text{bitlength}(|c_\alpha|) + 1) - 1.$$

[2] A compacted trie is made *hollow* by replacing the compacted path at each node by its length *and then discarding all its leaves.* A recursive definition of hollow trie appears in [3].

Since bitlength$(x) = \lceil \log(x+1) \rceil$, we have $\mathrm{HT}(S) = O(n \log \ell)$.

Storing functions. The problem of storing statically an r-bit function $f : A \to 2^r$ from a given set of keys A has recently received renewed attention [7]. For the purposes of this paper, we simply recall that these methods allow us to store an r-bit function on n keys using $rn + cn + o(n)$ bits for some constant $c \geq 0$, with $O(|x|/w)$ access time for a query string x. Practical implementations are described in [3]. In some cases, we will store a *compressed* function using a minimal perfect function ($O(n)$ bits) followed by a compressed data representation (e.g., an Elias–Fano compressed list [3]). In that case, storing natural numbers x_0, x_1, $\ldots$, x_{n-1} requires space $\sum_i \lfloor \log(x_i + 1) \rfloor + n \log(\sum_i \lfloor \log(x_i + 1) \rfloor / n) + O(n)$.

Relative dictionaries. A *relative dictionary* stores a set E relatively to some set $S \supseteq E$. That is, the relative dictionary answers questions about membership to E, but its answers are required to be correct only if the query string is in S. It is possible to store such a dictionary in $|E| \log(|S|/|E|)$ bits of space with $O(|x|/w)$ access time [2].

Rank and select. We will use two basic blocks of several succinct data structures—rank and select. Given a bit array (or bit string) $\boldsymbol{b} \in 2^n$, whose positions are numbered starting from 0, $\mathrm{rank}_{\boldsymbol{b}}(p)$ is the number of ones up to position p, exclusive ($0 \leq p \leq n$), whereas $\mathrm{select}_{\boldsymbol{b}}(r)$ is the position of the r-th one in $\boldsymbol{b}$, with bits numbered starting from 0 ($0 \leq r < \mathrm{rank}_{\boldsymbol{b}}(n)$). It is well known that these operations can be performed in constant time on a string of n bits using additional $o(n)$ bits, see [14,16].

3 From Prefixes to Exit Nodes

We break the weak prefix search problem into two subproblems. Our first goal is to go from a given prefix of some string in S to its exit node.

Hollow z-fast prefix tries. We start by describing an improvement of the *z-fast trie*, a data structure first defined in [2]. The main idea behind a z-fast trie is that, instead of representing explicitly a binary tree structure containing compacted paths of the trie, we will store a function that maps a certain prefix of each extent to the extent itself. This mapping (which can be stored in linear space) will be sufficient to navigate the trie and obtain, given a string x, the *name of the exit node of* x and the exit behaviour (left, right, or possibly equality for leaves). The interesting point about the z-fast trie is that it provides such a name in time $O(|x|/w + \log |x|)$, and that it leads easily to a probabilistically relaxed version, or even to hollow variants.

To make the paper self-contained, we recall the main definitions from [2]. The *2-fattest* number in a nonempty interval of positive integers is the number in the interval whose binary representation has the largest number of trailing zeros. Consider the compacted trie on S, one of its nodes α, and the 2-fattest number f in its skip interval $(i_\alpha \,..\, j_\alpha]$; if the interval is empty, which can happen only at the root, we set $f = 0$. The *handle* h_α of α is $e_\alpha[0 \,..\, f)$, where $e_\alpha[0 \,..\, f)$ denotes the first f bits of e_α. A *(deterministic) z-fast trie* is a dictionary T mapping each

Algorithm 1
Input: a prefix p of some string in S.
Output: the name of $\operatorname{exit}(p)$.

```
a, b ← 0, |p|
while b − a > 1 do
   f ← the 2-fattest number in (a . . b)
   g ← T(p[0 . . f))
   if g ≥ |p| then
      b ← f
   else
      a ← g
   end if
end while
if a = 0 then
   return ε
else
   return p[0 . . a + 1)
end if
```

Algorithm 2
Input: the name x of a node.
Output: the interval $[i\mathinner{.\,.}j)$ of strings prefixed by x.

```
if x = ε then
   i ← 0, j ← n
else
   i ← rank_b h(x^←)
   if x = 111 · · · 11 then
      j ← n
   else
      j ← rank_b h((x^+)^←)
   end if
end if
return [i . . j)
```

Fig. 2. Algorithms for weak prefix search and range location

handle h_α to the corresponding extent e_α. In Figure 1, the part of the mapping T with non-∞ output is the z-fast trie built on the trie of Figure 1.

We now introduce a more powerful structure, the *(deterministic) z-fast prefix trie*. Consider again a node α of the compacted trie on S with notation as above. The *pseudohandles* of α are the strings $e_\alpha[0\mathinner{.\,.}f')$, where f' ranges among the 2-fattest numbers of the intervals $(i_\alpha\mathinner{.\,.}t]$, with $i_\alpha < t < f$. Essentially, pseudohandles play the same rôle as handles for every *prefix* of the handle that extends the node name. We note immediately that there are at most $\log(f-i_\alpha) \leq \log|c_\alpha|$ pseudohandles associated with α, so the overall number of handles and pseudohandles is bounded by $\mathrm{HT}(S) + \sum_{x\in S}\log|x| = O(n\log\ell)$. It is now easy to define a z-fast prefix trie: the dictionary providing the map from handles to extents is enlarged to pseudohandles, which are mapped to the special value ∞.

We are actually interested in a *hollow* version of a z-fast prefix trie—more precisely, a version implemented by a function T that maps handles of internal nodes to the length of their extents, and handles of leaves and pseudohandles to ∞. The function (see again Figure 1) can be stored in a very small amount of space; nonetheless, we will still be able to compute the name of the exit node of any string that is a prefix of some string in S using Algorithm 1:

Theorem 1. *Let p be a nonempty string that is a prefix of some string in S and $X = \{\, p_0 = \varepsilon, p_1, \ldots, p_t \,\}$, where p_1, p_2, …, p_t are the extents of the nodes of the trie that are proper prefixes of p, ordered by increasing length. Let $(a\mathinner{.\,.}b)$ be the interval maintained by Algorithm 1. Before and after each iteration the following invariant is satisfied: $a = |p_j|$ for some j, and $a \leq |p_t| < b$.*

Proof. We note that the invariant is trivially true at the start, as the initial interval is $(0\mathinner{.\,.}|p|)$. We now prove by induction that in the rest of execution the

invariant is true. At each step we pick the 2-fattest number $f \in (|p_i| \mathinner{..} b)$, and change interval. We have two cases (we follow the notation of Algorithm 1):

- If $f > |p_t|$, since $f < |p|$ then it is either the length of the handle of the exit node of p, or the length of a pseudohandle associated with the exit node of p, so we set $b = f$ and the invariant is preserved.
- Otherwise, f is 2-fattest in $(|p_i| \mathinner{..} |p_t|]$, so $p[0 \mathinner{..} f)$ must be the handle of an ancestor of $\operatorname{exit}(p)$ (as f is 2-fattest in every subinterval of $(|p_i| \mathinner{..} |p_t|]$ that contains it) which implies $g = |p_k|$ for some k. Thus, by setting $a = g$ the invariant is preserved. □

By the previous theorem, Algorithm 1 is correct and completes in at most $\log |p|$ iterations. We note that finding the 2-fattest number in an interval requires the computation of the most significant bit[3], but alternatively we can check that $(1 \ll i) \mathbin{\&} a \neq (1 \ll i) \mathbin{\&} b$ for decreasing i: if the test is satisfied, the number is $b \mathbin{\&} -1 \ll i$, otherwise we decrement i.

Space and time. The space needed for a hollow z-fast prefix trie depends on the component chosen for its implementation. The most trivial bound uses a function mapping handles and pseudohandles to one bit that makes it possible to recognise handles of internal nodes ($O(n \log \ell)$ bits), and a function mapping handles to extent lengths ($O(n \log L)$ bits, where L is the maximum string length).

These results, however, can be significantly improved. First of all, we can store handles of internal nodes in a relative dictionary. The dictionary will store $n-1$ strings out of $O(n \log \ell)$ strings, using $O(n \log((n \log \ell)/n)) = O(n \log \log \ell)$ bits. Then, the mapping from handles to extent lengths $h_\alpha \mapsto |e_\alpha|$ can actually be recast into a mapping $h_\alpha \mapsto |e_\alpha| - |h_\alpha|$. But since $|e_\alpha| - |h_\alpha| \leq |c_\alpha|$, by storing these data by means of a compressed function we will use space

$$\sum_\alpha \lfloor \log(|e_\alpha| - |h_\alpha| + 1) \rfloor + O(n \log \log \ell) + O(n)$$
$$\leq \sum_\alpha \lfloor \log(|c_\alpha| + 1) \rfloor + O(n \log \log \ell) \leq \mathrm{HT}(S) + O(n \log \log \ell),$$

where α ranges over internal nodes.

Algorithm 1 cannot iterate more than $\log |p|$ times; at each step, we query constant-time data structures using a prefix of p: using incremental hashing [6, Section 5], we can preprocess p in time $O(|p|/w)$ (and in $|p|/B$ I/Os) so that hashing prefixes of p requires constant time afterwards. We conclude that Algorithm 1 requires time $O(|p|/w + \log |p|)$.

Faster, faster, faster... We now describe a data structure mapping prefixes to exit nodes inspired by the techniques used in [1] that needs $O(n\ell^{1/2} \log \ell)$ bits of space and answers in time $O(|p|/w)$, thus providing a different space/time tradeoff. The basic idea is as follows: let $s = \lceil \ell^{1/2} \rceil$ and, for each node α of

[3] More precisely, the 2-fattest number in $(a..b]$ is $-1 \ll \operatorname{msb}(a \oplus b) \mathbin{\&} b$.

the compacted trie on the set S, consider the set of prefixes of e_α with length $t \in (i_\alpha \mathinner{..} j_\alpha]$ such that either t is a multiple of s or is smaller than the first such multiple. More precisely, we consider prefixes whose length is either of the form ks, where $ks \in (i_\alpha \mathinner{..} j_\alpha]$, or in $(i_\alpha \mathinner{..} \min\{\,\bar{k}s, j_\alpha\,\}]$, where $\bar{k}$ is the minimum k such that $ks > i_\alpha$.

We store a function F mapping each prefix p defined above to the length of the name of the corresponding node α (actually, we can map p to $|p| - |n_\alpha|$). Additionally, we store a mapping G from each node name to the length of its extent (again, we can just map $n_\alpha \mapsto |c_\alpha|$).

To retrieve the exit node of a string p that is a prefix of some string in S, we consider the string $q = p[0 \mathinner{..} |p| - |p| \bmod s)$ (i.e., the longest prefix of p whose length is a multiple of s). Then, we check whether $G(p[0 \mathinner{..} F(q))) \geq |p|$ (i.e., whether p is a prefix of the extent of the exit node of q). If this is the case, then clearly p has the same exit node as q (i.e., $p[0 \mathinner{..} F(q))$). Otherwise, the map F provides directly the length of the name of the exit node of p, which is thus $p[0 \mathinner{..} F(p))$. All operations are completed in time $O(|p|/w)$. The proof that this structure uses space $O(n\ell^{1/2} \log \ell)$ is deferred to the full paper.

4 Range Location

Our next problem is determining the range (of lexicographical ranks) of the leaves that appear under a certain node of a trie. Actually, this problem is pretty common in static data structures, and usually it is solved by associating with each node a pair of integers of $\log n \leq w$ bits. However, this means that the structure has, in the worst case, a linear ($O(nw)$) dependency on the data.

To work around this issue, we propose to use a *range locator*—an abstraction of a component used in [2]. Here we redefine range locators from scratch, and improve their space usage so that it is dependent on the average string length, rather than on the maximum string length. A range locator takes as input *the name of a node*, and returns the range of ranks of the leaves that appear under that node. For instance, in our toy example the answer to 0010011 would be $[1 \mathinner{..} 3)$. To build a range locator, we need to introduce *monotone minimal perfect hashing*.

Given a set of n strings T, a *monotone minimal perfect hash function* [2] is a bijection $T \to n$ that preserves lexicographical ordering. This means that each string of T is mapped to its rank in T (but strings not in T give random results). We use the following results from [3]:[4]

Theorem 2. *Let T be a set of n strings of average length ℓ and maximum length L, and $x \in 2^*$ be a string. Then, there are monotone minimal perfect hashing functions on T that:*

1. *use space $O(n \log \ell)$ and answer in time $O(|x|/w)$;*
2. *use space $O(n \log \log L)$ and answer in time $O(|x|/w + \log |x|)$.*

[4] Actually, results in [3] are stated for prefix-free sets, but it is trivial to make a set of strings prefix-free at the cost of doubling the average length.

We show how a reduction can relieve us from the dependency on L; this is essential to our goals, as we want to depend just on the average length:

Theorem 3. *There is a monotone minimal perfect hashing function on* T *using space* $O(n \log\log \ell)$ *that answers in time* $O(|x|/w + \log|x|)$ *on a query string* $x \in 2^*$.

Proof. We divide T into the set of strings T^- shorter then $\ell \log n$, and the remaining "long" strings T^+. Setting up a n-bit vector $\boldsymbol{b}$ that records the elements of T^- with select-one and select-zero structures ($n + o(n)$ bits), we can reduce the problem to hashing monotonically T^- and T^+. We note, however, that using Theorem 2 the set T^- can be hashed in space $O(|T^-| \log\log(\ell \log n)) = O(|T^-| \log\log \ell)$, as $2\ell \geq \log n$, and T^+ can be hashed explicitly using a $(\log n)$-bit function; since $|T^+| \leq n/\log n$ necessarily, the function requires $O(n)$ bits. Overall, we obtain the required bounds. □

We now describe in detail our range locator, using the notation of Section 2. Given a string x, let $x^{\leftarrow}$ be x with all its trailing zeroes removed. We build a set of strings P as follows: for each extent e of an internal node, we add to P the strings $e^{\leftarrow}$, $e1$, and, if $e \neq 111 \cdots 11$, we also add to P the string $(e1^+)^{\leftarrow}$, where $e1^+$ denotes the successor of length $|e1|$ of $e1$ in lexicographical order (numerically, it is $e1 + 1$). We build a monotone minimal perfect hashing function h on P, noting the following easily proven fact:

Proposition 1. *The average length of the strings in* P *is at most* 3ℓ.

The second component of the range locator is a bit vector $\boldsymbol{b}$ of length $|P|$, in which bits corresponding to the names of leaves are set to one. The vector is endowed with a ranking structure $\mathrm{rank}_{\boldsymbol{b}}$ (see Figure 1).

It is now immediate that given a node name x, by hashing $x^{\leftarrow}$ and ranking the bit position thus obtained in $\boldsymbol{b}$, we obtain the left extreme of the range of leaves under x. Moreover, performing the same operations on $(x^+)^{\leftarrow}$, we obtain the right extreme. All these strings are in P by construction, except for the case of a node name of the form $111 \cdots 11$; however, in that case the right extreme is just the number of leaves (see Algorithm 2 for the details).

A range locator uses at most $3n + o(n)$ bits for $\boldsymbol{b}$ and its selection structures. Thus, space usage is dominated by the monotone hashing component. Using the structures described above, we obtain:

Theorem 4. *There are structures implementing range location in time* $O(|x|/w)$ *using* $O(n \log \ell)$ *bits of space, and in* $O(|x|/w + \log|x|)$ *time using* $O(n \log\log \ell)$ *bits of space.*

We remark that other combinations of monotone minimal perfect hashing and succinct data structures can lead to similar results. Among several such asymptotically equivalent solutions, we believe ours is the most practical.

5 Putting It All Together

In this section we gather the main results about prefix search:

Theorem 5. *There are structures implementing weak prefix search in space* $\mathrm{HT}(S) + O(n \log\log \ell)$ *with query time* $O(|p|/w + \log|p|)$*, and in space* $O(n\ell^{1/2} \log \ell)$ *with query time* $O(|p|/w)$*.*

Proof. The first structure uses a hollow z-fast prefix trie followed by the range locator of Theorem 3: the first component provides the name n_α of exit node of $|p|$; given n_α, the range locator returns the correct range. For the second structure, we use the structure defined in Section 3 followed by the first range locator of Theorem 2. □

Actually, the second structure described in Theorem 5 can be made to occupy space $O(n\ell^{1/c} \log \ell)$ for any constant $c > 0$ (the proof will be given in the full version):

Theorem 6. *For any constant* $c > 0$*, there is a structure implementing weak prefix search in space* $O(n\ell^{1/c} \log \ell)$ *with query time* $O(|p|/w)$*.*

We note that all our time bounds can be translated into I/O bounds in the *cache-oblivious model* if we replace the $O(|p|/w)$ terms by $O(|p|/B)$ (where B is the I/O block size in bits). The $O(|p|/w)$ term appears in two places: first, in the phase of precalculation of a hash-vector of $\lceil |p|/w \rceil$ hash words on the prefix p which is later used to compute all the hash functions on prefixes of p; second, in the range location phase, where we need to compute $x^{\leftarrow}$ and $(x^{+})^{\leftarrow}$, where x is a prefix of p and subsequently compute the hash vectors on $x^{\leftarrow}$ and $(x^{+})^{\leftarrow}$. Observe that the above operations can be carried on using arithmetic operations only, without any additional I/O (we can use 2-wise independent hashing involving only multiplications and additions for computing the hash vectors and only basic arithmetic operations for computing $x^{\leftarrow}$ and $(x^{+})^{\leftarrow}$) except for the writing of the result of the computation which occupies $O(|p|/w)$ words of space and thus take $O(|p|/B)$ I/Os. Thus in both cases we need only $O(|p|/B)$ I/Os corresponding to the time needed to read the pattern and to write the result.

6 A Space Lower Bound

In this section we show that the space usage achieved by the weak prefix search data structure described in Theorem 5 is optimal up to a constant factor. In fact, we show a matching lower bound for the easier problem of prefix counting (i.e., counting how many strings start with a given prefix), and consider the more general case where the answer is only required to be correct up to an additive constant less than k. We note that any data structure supporting prefix counting can be used to achieve approximate prefix counting, by building the data structure for the set that contains every k-th element in sorted order.

Theorem 7. *Consider a data structure (possibly randomised) indexing a set S of n strings with average length $\ell > \log(n) + 1$, supporting k-approximate prefix count queries: Given a prefix of some key in S, the structure returns the number of elements in S that have this prefix with an additive error of less than k, where $k < n/2$. The data structure may return any number when given a string that is not a prefix of a key in S. Then the expected space usage on a worst-case set S is $\Omega((n/k)\log(\ell - \log n))$ bits. In particular, if no error is allowed and $\ell > (1+\varepsilon)\log n$, for constant $\varepsilon > 0$, the expected space usage is $\Omega(n \log \ell)$ bits.*

Proof. Let $u = 2^\ell$ be the number of possible keys of length ℓ. We show that there exists a probability distribution on key sets S such that the expected space usage is $\Omega((n/k)\log\log(u/n))$ bits. By the "easy directions of Yao's lemma," this implies that the expected space usage of any (possibly randomised) data structure on a worst case input is at least $\Omega((n/k)\log\log(u/n))$ bits. The bound for $\ell > (1+\varepsilon)\log n$ and $k = 1$ follows immediately.

Assume without loss of generality that $n/(k+1)$ and k are powers of 2. All strings in S will be of the form $abc \in 2^*$, where $|a| = \log_2(n/(k+1))$, $|b| = \ell - \log_2(n/(k+1)) - \log_2 k$, and $|c| = \log_2 k$. Let $t = \ell - \log_2(n/(k+1)) - \log_2 k$ denote the length of b. For every value of a the set will contain exactly $k+1$ elements: One where b and c are strings of 0s, and for b chosen uniformly at random among strings of Hamming weight 1 we have k strings for $c \in 2^{\log_2 k}$. Notice that the entropy of the set S is $n/(k+1)\log_2 t$, as we choose $n/(k+1)$ values of b independently from a set of t strings. To finish the argument we will need to show that any two such sets require different data structures, which means that the entropy of the bit string representing the data structure for S must also be at least $n/(k+1)\log_2 t$, and in particular this is a lower bound on the expected length of the bit string.

Consider two different sets S' and S''. There exists a value of a, and distinct values b', b'' of Hamming weight 1 such that S' contains all k ℓ-bits strings prefixed by ab', and S'' contains all k ℓ-bits strings prefixed by ab''. Assume without loss of generality that b' is lexicographically before b''. Now consider the query for a string of the form $a0^\ell$, which is a prefix of ab' but not ab'' – such a string exists since b' and b'' have Hamming weight 1. The number of keys with this prefix is $k+1$ and 1, respectively, for S' and S'', so the answers to the queries must be different (both in the multiplicative and additive case). Hence, different data structures are needed for S' and S''. □

Note that the trivial information-theoretical lower bound does not apply, as it is impossible to reconstruct S from the data structure.

It is interesting to note the connections with the lower and upper bounds presented in [11]. This paper shows a lower bound on the number of bits necessary to represent a set of strings S that, in the binary case, reduces to $\mathrm{T}(S) + \log \ell$, and provide a matching data structure. Theorem 5 provides a *hollow* data structure that is sized following the naturally associated measure: $\mathrm{HT}(S) + O(n \log\log \ell)$. Thus, Theorem 5 and 7 can be seen as the hollow version of the results presented in [11], albeit our lower bound is a match only asymptotically. Improving Theorem 7 to $\mathrm{HT}(S) + o(\mathrm{HT}(S))$ is an interesting open problem.

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: Optimal static range reporting in one dimension. In: STOC 2001, pp. 476–482 (2001)
2. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In: SODA 2009, pp. 785–794. ACM Press, New York (2009)
3. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Theory and practise of monotone minimal perfect hashing. In: ALENEX 2009. SIAM, Philadelphia (2009)
4. Bender, M.A., Farach-Colton, M., Kuszmaul, B.C.: Cache-oblivious string B-trees. In: PODS 2006, pp. 233–242. ACM, New York (2006)
5. Brodal, G.S., Fagerberg, R.: Cache-oblivious string dictionaries. In: SODA 2006, pp. 581–590 (2006)
6. Dietzfelbinger, M., Gil, J., Matias, Y., Pippenger, N.: Polynomial hash functions are reliable (extended abstract). In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 235–246. Springer, Heidelberg (1992)
7. Dietzfelbinger, M., Pagh, R.: Succinct data structures for retrieval and approximate membership (extended abstract). In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 385–396. Springer, Heidelberg (2008)
8. Elias, P.: Efficient storage and retrieval by content and address of static files. J. Assoc. Comput. Mach. 21(2), 246–260 (1974)
9. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. on Info. Theory 21, 194–203 (1975)
10. Ferragina, P., Grossi, R.: 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)
11. Ferragina, P., Grossi, R., Gupta, A., Shah, R., Vitter, J.S.: On searching compressed string collections cache-obliviously. In: PODS 2008, pp. 181–190 (2008)
12. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: FOCS 1999, pp. 285–297. IEEE Comput. Soc. Press, Los Alamitos (1999)
13. Gupta, A., Hon, W.-K., Shah, R., Vitter, J.S.: Compressed data structures: Dictionaries and data-aware measures. Theor. Comput. Sci. 387(3), 313–331 (2007)
14. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS 1989, pp. 549–554 (1989)
15. Pǎtraşcu, M., Thorup, M.: Randomization does not help searching predecessors. In: SODA 2007, pp. 555–564 (2007)
16. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA 2002, pp. 233–242. ACM Press, New York (2002)