Fast Compressed Tries through Path Decompositions

ROBERTO GROSSI and GIUSEPPE OTTAVIANO, Università di Pisa

Tries are popular data structures for storing a set of strings, where common prefixes are represented by common root-to-node paths. More than 50 years of usage have produced many variants and implementations to overcome some of their limitations. We explore new succinct representations of path-decomposed tries and experimentally evaluate the corresponding reduction in space usage and memory latency, comparing with the state of the art. We study the following applications: compressed string dictionary and monotone minimal perfect hash for strings.

In compressed string dictionary, we obtain data structures that outperform other state-of-the-art compressed dictionaries in space efficiency while obtaining predictable query times that are competitive with data structures preferred by the practitioners. On real-world datasets, our compressed tries obtain the smallest space (except for one case) and have the fastest lookup times, whereas access times are within 20% slower than the best-known solutions.

In monotone minimal perfect hash for strings, our compressed tries perform several times faster than other trie-based monotone perfect hash functions while occupying nearly the same space. On real-world datasets, our tries are approximately 2 to 5 times faster than previous solutions, with a space occupancy less than 10% larger.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; E.1 [**Data Structures**]: Trees

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Succinct data structures, string dictionaries, monotone perfect hash functions, tries

ACM Reference Format:
Roberto Grossi and Giuseppe Ottaviano. 2014. Fast compressed tries through path decompositions. ACM J. Exp. Algor. 19, 1, Article 1.8 (October 2014), 20 pages.
DOI: http://dx.doi.org/10.1145/2656332

1. INTRODUCTION

Tries are widely used data structures that turn a string set into a digital search tree. Several operations can be supported, such as mapping the strings to integers, re*trie*ving a string from the trie, performing prefix searches, and many others. Thanks to their simplicity and functionality, they have enjoyed a remarkable popularity in a number of fields—computational biology, data compression, data mining, information retrieval, natural language processing, network routing, pattern matching, text processing, and Web applications, to name a few—motivating the significant effort spent in the variety of their implementations over the past 50 years [Knuth 1998].

Part of the work was done while G. Ottaviano was an intern at Microsoft Research, Cambridge.
Authors' address: R. Grossi and G. Ottaviano, Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo, 3, 56127 Pisa, Italy; emails: {grossi, ottavian}@di.unipi.it.

DOI: http://dx.doi.org/10.1145/2656332

However, their simplicity comes at a cost: as most tree structures, they generally suffer poor locality of reference due to pointer chasing. This effect is amplified when using space-efficient representations of tries, where performing any basic navigational operation, such as visiting a child, requires accessing possibly several directories, usually with unpredictable memory access patterns. Tries are particularly affected, as they are unbalanced structures: the height can be in the order of the number of strings in the set. Furthermore, space savings are achieved only by exploiting the common prefixes in the string set, although it is not clear how to compress their nodes and their labels without incurring an unreasonable overhead in the running time.

In this article, we experiment with how *path decompositions* of tries help on both of the issues mentioned previously, inspired by the work presented in Ferragina et al. [2008]. By using a *centroid* path decomposition, the height is guaranteed to be logarithmic in the number of strings, reducing dramatically the number of cache misses in a traversal; in addition, for any path decomposition, the labels can be laid out in a way that enables efficient compression and decompression of a label in a sequential fashion.

We keep two main goals in mind: reduce the space requirement, and guarantee fast query times using algorithms that exploit the memory hierarchy. In our algorithm engineering design, we follow some guidelines. First, proposed algorithms and data structures should be as simple as possible to ensure reproducibility of the results, whereas the performance should be similar to or better than what is available in the state of the art. Second, the proposed techniques should as much as possible lay on a theoretical ground. Third, the theoretical complexity of some operations is allowed to be worse than that known for the best solutions when there is a clear experimental benefit,[1] since we seek the best performance in practice.

The literature about space-efficient and cache-efficient tries is vast. Several papers address the issue of a cache-friendly access to a set of strings supporting prefix search [Acharya et al. 1999; Bender et al. 2006; Brodal and Fagerberg 2006; Ferragina and Grossi 1999], but they do not deal with space issues except Bender et al. [2006], which introduces an elegant variant of front coding. Other papers aiming at succinct labeled trees and compressed data structures for strings [Arroyuelo et al. 2010; Belazzougui et al. 2011; Benoit et al. 2005; Blandford and Blelloch 2008; Ferragina et al. 2009; Munro and Raman 2001; Sadakane and Navarro 2010], support powerful operations—such as substring queries—and are very good in compressing data, but they do not exploit the memory hierarchy. Few papers [Chiu et al. 2010; Ferragina et al. 2008] combine (nearly) optimal information theoretic bounds for space occupancy with good cache-efficient bounds, but no experimental analysis is performed. More references on compressed string dictionaries can be found in Brisaboa et al. [2011].

The article is organized as follows. After describing some of the known tools to represent sequences and trees succinctly, we apply our path decomposition ideas to string dictionaries in Section 3 and to monotone perfect hash functions (hollow tries) in Section 4, showing that it is possible to improve their performance with a very small space overhead. In Section 5, we present some optimizations to the range min-max tree [Sadakane and Navarro 2010; Arroyuelo et al. 2010] that we use to support fast operations on balanced parentheses, improving both in space and time on the existing implementations [Arroyuelo et al. 2010]. Our experimental results are discussed in Section 6, where our implementations compare very favorably to some of the best implementations. We provide the source code at http://github.com/ot/path_decomposed_tries for the reader interested in further comparisons.

[1]For example, it is folklore that a sequential scan of a *small* sorted set of keys is faster than a binary search, because the former method is very friendly with branch prediction and cache prefetching of modern machines.

2. BACKGROUND AND TOOLS

2.1. Basic Tools

In the following, we make extensive use of compacted tries and basic succinct data structures, which we shortly summarize in this section.

Compacted tries. To fix the notation, we quickly recall the definition of compacted tries. We build recursively the trie in the following way:

—The compacted trie of a single string is a node whose label is the string.
—Given a nonempty string set $\mathcal{S}$, the root of the tree is labeled with the longest common prefix α (possibly empty) of the strings in $\mathcal{S}$. For each character b such that the set $\mathcal{S}_b = \{\beta | \alpha b \beta \in \mathcal{S}\}$ is nonempty, the compacted trie built on $\mathcal{S}_b$ is attached to the root as a child. The edge is labeled with the *branching character* b. The number of characters in the label α is called the *skip* and denoted with δ.

Unless otherwise specified, we will use *trie* to indicate a *compacted trie* in the rest of the article.

Rank and select operations. Given a bit vector X with n bits, we can define the following operations: $\text{Rank}_b(i)$ returns the number of occurrences of bit $b \in \{\mathtt{0}, \mathtt{1}\}$ in the first i positions in X; $\text{Select}_b(i)$ returns the position of the i-th occurrence of bit b in X. These operations can be supported in constant time by adding $o(n)$ bits of redundancy to the bit vector [Clark 1998; Jacobson 1989].

Elias-Fano encoding. The Elias-Fano representation [Elias 1974; Fano 1971] is an encoding scheme to represent a nondecreasing sequence of m integers in $[0, n)$ occupying $2m + m\lceil \log \frac{n}{m} \rceil + o(m)$ bits while supporting constant-time access to the i-th integer. The scheme is very simple and elegant, and efficient implementations are described in Grossi and Vitter [2005], Okanohara and Sadakane [2007], and Vigna [2008].

Balanced parentheses. In a sequence of n balanced parentheses (BP), each open parenthesis `(` can be associated to its *mate* `)`. Operations FindClose and FindOpen can be defined, which find the position of the mate of respectively an open or close parenthesis. The sequence can be represented as a bit vector, where `1` represents `(` and `0` represents `)`, and by adding $o(n)$ bits of redundancy it is possible to support the previously defined operations in constant or nearly constant time [Jacobson 1989; Munro and Raman 2001]. A tree T can be recursively encoded as a sequence of balanced parentheses $\text{BP}(T)$ by encoding a leaf node as `()` and a node with children $T_1, \ldots, T_k$ as `(` $\text{BP}(T_1) \cdots \text{BP}(T_k)$ `)`. Note that each node is represented by a pair of parentheses, so a tree with m nodes can be represented with $2m$ parentheses. Several traversal operations can be implemented using the preceding FindClose and FindOpen operations.

2.2. Representing Binary Trees

We define a *binary tree* as a tree where each node is either an *internal node* that has *exactly* two children or a *leaf*. It follows immediately that a binary tree with n internal nodes has $n + 1$ leaves. An example of binary trees is given by binary compacted tries. Note that there is another popular definition of binary trees, such as is used in Munro and Raman [2001], where each node is allowed to have *at most* two children, but a distinction is made between degree-1 nodes based on whether their child is left or right. The two definitions are equivalent if we consider the internal nodes in the first definition to be the nodes in the second definition, and the leaves to be the *absent children*. Hence, a tree with $2n + 1$ nodes in the first definition is a tree with n nodes in the second. Choosing between the two definitions usually depends on whether the leaves have some significance for the application.

A simple way to represent a binary tree would be to see it as a generic tree and encode it with BP, taking $4n + o(n)$ bits for a tree with $2n + 1$ nodes. However, it is possible to do better: as noted in Munro and Raman [2001], binary trees with $2n + 1$ nodes can be bijectively mapped to general trees of n nodes, by considering the n internal nodes as the first-child next-sibling representation of a tree of n nodes; operations on the binary tree can be performed directly on the obtained tree. Such a tree can be encoded with BP, occupying just $2n + o(n)$ bits. This technique is used, for example, for hollow tries in Belazzougui et al. [2011]. Given a tree T, we call such encoding $\mathrm{FCNS}(T)$.

We show here how the same representation can be obtained directly from the tree, without the conceptual step through the first-child next-sibling mapping. This makes reasoning about the operations considerably simpler. Furthermore, with this derivation, the leaves have an explicit correspondence with positions in the parentheses sequence.

Definition 2.1. Let T be a binary tree. We define its depth-first binary sequence representation $\mathrm{DFBS}(T)$ recursively:

—If T is a leaf, $\mathrm{DFBS}(T) = \texttt{)}$.
—If T is a node with subtrees T_{left} and T_{right}, $\mathrm{DFBS}(T) = \texttt{(}\,\mathrm{DFBS}(T_{\mathrm{left}})\,\mathrm{DFBS}(T_{\mathrm{right}})$.

Although in FCNS only internal nodes are mapped to `(`s, in this representation all $2n+1$ nodes in depth-first order are in explicit correspondence to the $2n + 1$ parentheses in the sequence, with `(`s corresponding to internal nodes and `)`s corresponding to leaves. This property is especially useful if satellite data need to be associated to the leaves.

The following lemma proves that the obtained sequence is isomorphic to $\mathrm{FCNS}(T)$.

LEMMA 2.2. *For any binary tree T, it holds that* $\mathrm{DFBS}(T) = \mathrm{FCNS}(T)\,\texttt{)}$.

PROOF. If T is a leaf, $\mathrm{FCNS}(T)$ is the empty sequence and $\mathrm{DFBS}(T)$ is the single parenthesis `)`, so the lemma follows trivially. Otherwise, let $T_1, T_2, \ldots, T_i$ be the subtrees hanging off the right-most root-to-leaf path in T; then, by definition,

$$\mathrm{FCNS}(T) = \texttt{(}\,\mathrm{FCNS}(T_1)\texttt{)}\texttt{(}\,\mathrm{FCNS}(T_2)\texttt{)} \ldots \texttt{(}\,\mathrm{FCNS}(T_i)\texttt{)}.$$

By induction, we can rewrite it in the following way:

$$\mathrm{FCNS}(T) = \texttt{(}\,\mathrm{DFBS}(T_1)\texttt{(}\,\mathrm{DFBS}(T_2) \ldots \texttt{(}\,\mathrm{DFBS}(T_i).$$

Note that T_1 is just T_{left}, whereas $\mathrm{DFBS}(T_{\mathrm{right}}) = \texttt{(}\,\mathrm{DFBS}(T_2) \ldots \texttt{(}\,\mathrm{DFBS}(T_i)\texttt{)}$, where the last `)` is given by the final leaf in the right-most root-to-leaf path. Hence,

$$\mathrm{DFBS}(T) = \texttt{(}\,\mathrm{DFBS}(T_1)\texttt{(}\,\mathrm{DFBS}(T_2) \ldots \texttt{(}\,\mathrm{DFBS}(T_i)\texttt{)} = \mathrm{FCNS}(T)\texttt{)},$$

proving the inductive step. □

The preceding lemma implies that $\texttt{(}\,\mathrm{DFBS}(T)$ is a sequence of balanced parentheses. To make the construction self-contained, we prove it directly in the following lemma. The proof will also make clear how to perform navigational operations.

LEMMA 2.3. *The sequence* $\texttt{(}\,\mathrm{DFBS}(T)$ *is a sequence of balanced parentheses.*

PROOF. If T is a leaf, then $\texttt{(}\,\mathrm{DFBS}(T) = \texttt{()}$. Otherwise, let $\texttt{(}\,\mathrm{DFBS}(T) = \texttt{((}\,\mathrm{DFBS}(T_{\mathrm{left}})\,\mathrm{DFBS}(T_{\mathrm{right}})$. By induction, both $\texttt{(}\,\mathrm{DFBS}(T_{\mathrm{left}})$ and $\texttt{(}\,\mathrm{DFBS}(T_{\mathrm{right}})$ are sequences of balanced parentheses. Inserting the first into the second yields a sequence of balanced parentheses. □

Note that internal nodes are of the form $\texttt{(}\,\mathrm{DFBS}(T_{\mathrm{left}})\,\mathrm{DFBS}(T_{\mathrm{right}})$, where T_{left} and T_{right} are the children. The previous lemma allows us to skip the $\texttt{(}\,\mathrm{DFBS}(T_{\mathrm{left}})$ subsequence with a single FindClose, and hence the following corollary follows immediately.

COROLLARY 2.4. *If position i in* DFBS(T) *corresponds to an internal node, then* LeftChild(i) = $i + 1$ *and* RightChild(i) = FindClose(i) + 1.

The DFBS encoding is discussed also by Jansson et al. [2012], but without mentioning that it is possible to perform traversal operations on it. They obtain the n-bit bound instead using a compressed version of DFUDS, which is more powerful but also significantly more complex. Davoodi et al. [2012] note that the encoding yields a sequence of balanced parentheses, but they do not mention the equivalence with FCNS.

Using this encoding, it is also possible to generate random binary trees of a given size in a very intuitive way: to generate a tree of size $2n + 1$, choose an odd number m between 1 and $2n - 1$ from some distribution to be the size of the left subtree. Then, recursively generate trees of sizes m and $2n - m$, obtaining the sequences α and β. The resulting tree is $(\alpha\beta$. We use this technique to benchmark BP implementations in Section 5.1.

2.3. The *Succinct* Library

We implemented the data structures described earlier in the Succinct C++ library [SUCCINCT 2012]. The library is tested under Linux, Mac OS X, and Microsoft Windows, and released with a permissive license, in the hope that it will be useful both in research and applications.

Although similar in functionality to other existing libraries [LIBCDS 2011; SDSL 2010; SUX4J 2011], we made some radically different architectural choices. The most prominent is that data structures serialized to disk are not *deserialized* into memory but rather are *memory mapped*; although being slightly less flexible, memory mapping has several advantages over loading. For example, for short-running processes, it is often not necessary to load the full data structure in memory; instead, the kernel will load only the relevant pages. If such pages were accessed recently, they are likely to be still in the kernel's page cache, thus making the startup even faster. If several processes access the same structure, the memory pages that hold it are shared among all of the processes; with loading, instead, each process keeps its own copy of the data. Last, if the system runs out of memory, it can just unmap unused pages; with loading, it has to *swap* them to disk, thus increasing the I/O pressure.

We chose to avoid dynamic polymorphism and make extensive use of C++ templates instead. This allowed us to write idiomatic and modular C++ code without the overhead of virtual functions. Manual inspection of the generated code confirmed that, thanks to the ability to inline functions, there is virtually no abstraction penalty on all of the compilers on which we tested the code.

The library also implements the range min tree described in Section 5; as shown in Section 5.1, at the time of this writing, this implementation is the fastest among the publicly available range min-max–based BP data structures.

3. STRING DICTIONARIES

In this section, we describe an implementation of string dictionaries using path-decomposed tries. A *string dictionary* is a data structure on a string set $\mathcal{S} \subset \Sigma^*$ that supports the following operations:

—Lookup(s) returns -1 if $s \notin \mathcal{S}$ or a unique identifier in $[0, |\mathcal{S}|)$ otherwise.
—Access(i) retrieves the string with identifier i; note that Access(Lookup(s)) = s if $s \in \mathcal{S}$.

Path decomposition. Our string dictionaries, inspired by the approach described in Ferragina et al. [2008], are based on *path decompositions* of the trie built on $\mathcal{S}$ (recall that we use *trie* to indicate *compacted trie* in the rest of the article). A path

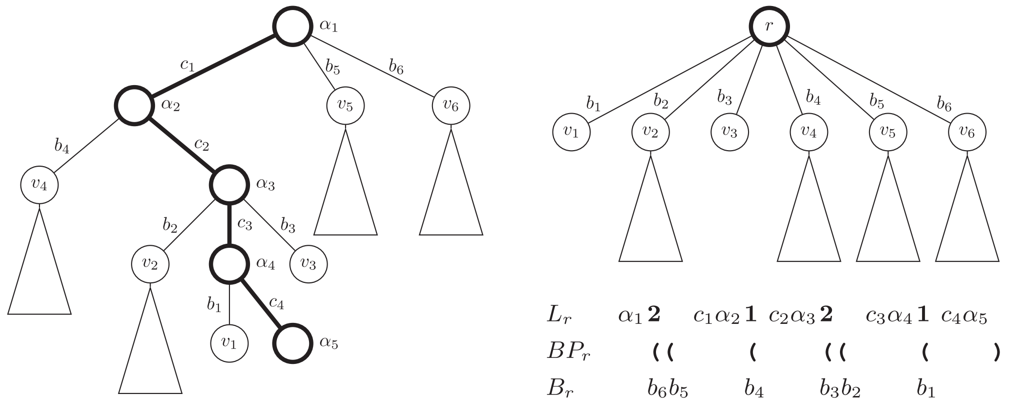

α_1
c_1
b_5
b_6
α_2
v_5
v_6
b_4
c_2
v_4
α_3
b_2
c_3
b_3
v_2
α_4
v_3
b_1
c_4
v_1
α_5
r
b_1 b_2 b_3 b_4 b_5 b_6
v_1 v_2 v_3 v_4 v_5 v_6
L_r α_1 **2** $c_1\alpha_2$ **1** $c_2\alpha_3$ **2** $c_3\alpha_4$ **1** $c_4\alpha_5$
BP_r (((((( )
B_r b_6b_5 b_4 b_3b_2 b_1

Fig. 1. Path decomposition of a trie. The α_i denote the labels of the trie nodes, and c_i and b_i denote the branching characters (depending on whether they are on the path or not).

decomposition $\mathcal{T}^c$ of a trie $\mathcal{T}$ is a tree where each node in $\mathcal{T}^c$ represents a path in $\mathcal{T}$. It is defined recursively in the following way: a root-to-leaf path in $\mathcal{T}$ is chosen and represented by the root node in $\mathcal{T}^c$. The same procedure is applied recursively to the subtries hanging off the chosen path, and the obtained trees become the children of the root, labeled with their corresponding branching character. Note that in the preceding procedure, the order of the decomposed subtries as children of the root is arbitrary. Unlike Ferragina et al. [2008], who arrange the subtries in *lexicographic order*, we arrange them in *bottom-to-top left-to-right* order; when using our succinct representation, this simplifies the traversal. Figure 1 shows a root-to-leaf path in $\mathcal{T}$ and its resulting node in $\mathcal{T}^c$.

There is a one-to-one correspondence between the paths of the two trees: root-to-*node* paths in $\mathcal{T}^c$ correspond to root-to-*leaf* paths in the trie $\mathcal{T}$, and hence to strings in $\mathcal{S}$. This also implies that $\mathcal{T}^c$ has exactly $|\mathcal{S}|$ nodes, and that the height of $\mathcal{T}^c$ cannot be larger than that of $\mathcal{T}$. Different strategies in choosing the paths in the decomposition give rise to different properties. We describe two such strategies:

—*Left-most path*: Always choose the left-most child.
—*Heavy path*: Always choose the *heavy* child—that is, the one whose subtrie has the most leaves (arbitrarily breaking ties). This is the strategy adopted in Ferragina et al. [2008] and borrowed from Sleator and Tarjan [1981].

OBSERVATION 3.1. *If the left-most path is used in the path decomposition, the depth-first order of the nodes in $\mathcal{T}^c$ is equal to the left-to-right order of their corresponding leaves in $\mathcal{T}$. Hence, if $\mathcal{T}$ is lexicographically ordered, so is $\mathcal{T}^c$. We call it a* lexicographic path decomposition.

OBSERVATION 3.2. *If the heavy path is used in the path decomposition, the height of the resulting tree is bounded by $O(\log|\mathcal{S}|)$. We call such a decomposition a* centroid path decomposition.

The two strategies enable a time/functionality trade-off: a lexicographic path decomposition guarantees that the indices returned by the Lookup are in lexicographic order, at the cost of a potentially linear height of the tree (but never higher than the trie $\mathcal{T}$). On the other hand, if the order of the indices is irrelevant, the centroid path decomposition gives logarithmic guarantees.[2]

[2]Ferragina et al. [2008] show how to have lexicographic indices in a centroid path-decomposed trie using secondary support structures and arranging the nodes in a different order. The navigational operations are

We exploit a crucial property of path decompositions: since each node in $\mathcal{T}^c$ corresponds to a node-to-leaf path π in $\mathcal{T}$, the concatenation of the labels in π corresponds to a suffix of a string in $\mathcal{S}$. To simulate a traversal of $\mathcal{T}$ using $\mathcal{T}^c$, we only need to scan sequentially, character-by-character, the label of each node until we find the needed child node. Hence, any representation of the labels that supports *sequential access* (simpler than random access) is sufficient. Besides being cache-friendly, as we will see in the next section, this allows an efficient compression of the labels.

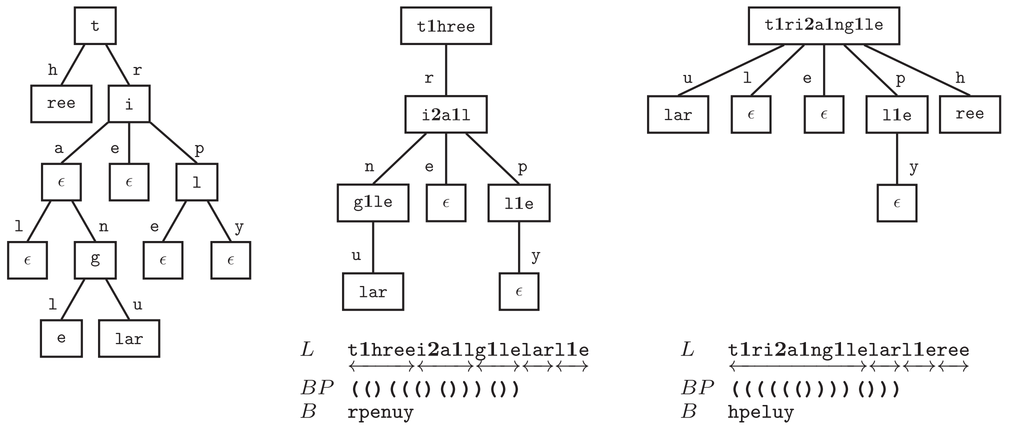
Tree representation. Each node v in the path-decomposed trie $\mathcal{T}^c$ is encoded with three sequences: BP_v, B_v, and L_v. Figure 1 shows an example for the root node.

—The bit vector BP_v is a run of d_v open parentheses followed by a close parenthesis, where d_v is the degree of node v.
—The string B_v is the concatenation of the branching characters b_i of node v, written in reverse order—that is, $B_v = b_{d_v} \cdots b_1$. Note that they are in one-to-one correspondence with the (s in BP.
—The string L_v is the *label* of node v. We recall that each node in $\mathcal{T}^c$ represents a *path* in $\mathcal{T}$. To encode the path, we augment the alphabet Σ with $|\Sigma| - 1$ special characters, $\Sigma' = \Sigma \cup \{\mathbf{1}, \mathbf{2}, \ldots, |\Sigma| - \mathbf{1}\}$, and alternate the label and the branching character of each node in the trie path with the number of subtries hanging off that node, encoded with the new special characters. More precisely, if the path in $\mathcal{T}$ corresponding to v is $w_1, c_1, \ldots, w_{k-1}, c_{k-1}, w_k$ where w_i are the trie nodes and c_i the edge labels, then $L = \alpha_{w_1}\tilde{d}_{w_1}c_1 \cdots \alpha_{w_{k-1}}\tilde{d}_{w_{k-1}}c_{k-1}\alpha_{w_k}$, where α_{w_i} is the node label of w_i and $\tilde{d}_{w_i}$ is the special character that represents $d_{w_i} - 1$ (all degrees in $\mathcal{T}$ are at least 2). Note that L_v is drawn from the larger alphabet Σ'; we will describe later how to encode it.

The sequences are then concatenated in depth-first order to obtain BP, B, and L: if $v_1, \ldots, v_n$ are the nodes of $\mathcal{T}^c$ in depth-first order, then $BP = BP_{v_1} \cdots BP_{v_n}$, $B = B_{v_1} \cdots B_{v_n}$, and $L = L_{v_1} \cdots L_{v_n}$. Note that BP represents the topology of $\mathcal{T}^c$ with the DFUDS encoding, and hence it is possible to support fast traversal operations on the tree (we refer the reader to Benoit et al. [2005] for the definition of the DFUDS encoding). Since the labels L_v are variable sized, to enable random access to the beginning of each label we encode their endpoints in L using an Elias-Fano monotone sequence. Figure 2 shows an example of the three sequences for both lexicographic and centroid path decompositions on a small string set.

Trie operations. Lookup is implemented recursively in a top-down fashion, starting from the root. To search for the string s in node v, we simultaneously scan its label L_v and s. If the current character in the label is a special character, we add its value to an accumulator m (initially set to zero). Otherwise, we check if the current character in the label and the one in s match, in which case we proceed the scan with the next position in both. If instead they mismatch, the previous character in the label must be a special character $\tilde{d}$, and otherwise the string s is not in $\mathcal{S}$; in this case, we return -1 (likewise, if the string exceeds the label). The range between the accumulator m and $m + \tilde{d}$ indicates the children of v that correspond to the nodes in $\mathcal{T}$ branching from the prefix of s traversed up to the mismatch. To find the child, it is then sufficient to find the matching branching character in B_v between in m and $m + \tilde{d}$. Since B is in correspondence with the open parentheses of BP, by performing a $\mathrm{Rank}_($ of the current position in BP, it is possible to find the starting point of B_v in B; also, because the characters in that range are sorted, a binary search can be performed. Again, -1 is returned if no matching character is found. The search proceeds recursively in the

noticeably more complex and require more powerful primitives on the underlying succinct tree, particularly for Access.



(a) Compacted trie (b) Lexicographic decomposition (c) Centroid decomposition

Fig. 2. Example of a compacted trie and its lexicographic and centroid path decomposition trees for the set {`three`, `trial`, `triangle`, `triangular`, `trie`, `triple`, `triply`}. Note that the children in the path decomposition trees are in bottom-to-top left-to-right order.

found node with the suffix of s, which starts immediately after the mismatch, until the string is fully traversed. The index of the `)` in *BP* corresponding to the found node is returned—that is, the depth-first index of that node, which can be found with a $\text{Rank}_{)}$ of the current position in *BP*. Note that it is possible to avoid all of the Rank calls by using the standard trick of double-counting—in other words, exploiting the observation that between any two mates there is an equal number of `(`s and `)`s; this implies that if $j = \text{FindClose}(i)$, then $\text{Rank}_{)}(j) = \text{Rank}_{)}(i) + (j - i - 1)/2$, so it is possible to keep track of the rank during the traversal. Likewise, $\text{Rank}_{(}(i) = i - \text{Rank}_{)}(i)$.

Access is performed similarly but in a bottom-up fashion. The initial position in *BP* is obtained by performing a $\text{Select}_{)}$ of the node index returned by Lookup. Then the path is reconstructed by jumping recursively from the leaf thus found in $\mathcal{T}^c$ to the parent until the root is reached. During the recursion, we maintain the invariant that, at node v, the suffix of the string to be returned corresponding to the path below v has already been decoded. When visiting node v, we know from which child of v we are jumping; hence, we scan its label L_v from the beginning until the position corresponding to that child is reached. The nonspecial characters seen during the scan are prepended to the string to be returned, and the recursion is carried on to the parent.

Time complexity. For the Lookup, for each node in the traversal, we perform a sequential scan of the labels and a binary search on the branching character. If the string has length p, we can never see more than p special characters during the scan. Hence, if we assume constant-time FindClose and Elias-Fano retrieval, the total number of operations is $O(p+h\log|\Sigma|)$, whereas the number of random memory accesses is bounded by $O(h)$, where h is the height of the path decomposition tree. The Access is symmetric, except the binary search is not needed and $p \geq h$, so the number of operations is bounded by $O(p)$, where p is the length of the returned string. Again, the number of random memory accesses is bounded by $O(h)$.

Labels encoding and compression. As mentioned previously, we need only to sequentially *scan* the label of each node from the beginning, so we can use any encoding that supports sequential scan with a constant amount of work per character. In the uncompressed trie, as a baseline, we simply use a vbyte encoding [Williams and Zobel 1999].

Since most bytes in the datasets do not exceed 127 in value, there is no noticeable space overhead. For a less sparse alphabet, more sophisticated encodings can be used.

The freedom in choosing the encoding allows us to explore other trade-offs. We take advantage of this to *compress the labels*, with an almost negligible overhead in the operations runtime.

We adopt a simple dictionary compression scheme for the labels: we choose a static dictionary of variable-sized words (that can be drawn from any alphabet) that will be stored along with the the tree explicitly, such that the overall size of the dictionary is bounded by a given parameter (constant) D. The node labels are then parsed into words of the dictionary, and the words are sorted according to their frequency in the parsing: a code is assigned to each word in decreasing order of frequency so that more frequent words have smaller codes. The codes are then encoded using some variable-length integer encoding; we use vbyte to favor performance. To decompress the label we scan the codes, and for each code we scan the word in the dictionary; hence, each character requires a constant amount of work.

We remark that the decompression algorithm is completely agnostic of how the dictionary was chosen and how the strings are parsed. For example, domain knowledge about the data could be exploited; in texts, the most frequent words would probably be a good choice.

Since we are looking for a general-purpose scheme, we adopt a modified version of the approximate re-pair [Larsson and Moffat 1999] described in Claude and Navarro [2010]. We initialize the dictionary to the alphabet Σ and scan the string to find the k most frequent pairs of codes. Then, we select all pairs whose corresponding substrings fit in the dictionary and substitute them in the sequence. We then iterate until the dictionary is filled (or there are no more repeated pairs). From this, we simultaneously obtain the dictionary and the parsing. To allow the labels to be accessed independently, we take care that no pairs are formed on label boundaries, as done in Brisaboa et al. [2011].

Note that whereas re-pair represents the words recursively as pairing rules, our dictionary stores the words literally, thus losing some space efficiency but fulfilling our requirement of constant amount of work per decoded character. If we used re-pair instead, accessing a single character from a recursive rule would have had a cost dependent on the recursion depth.

Implementation notes. For the *BP* vector, we use the range min tree described in Section 5 with a block size of 256 bits. Rank is supported using the `rank9` structure described in Vigna [2008], whereas Select is implemented through a one-level hinted binary search. The search for the branching character is replaced by a linear search, which for the cardinalities considered (few tens of distinct symbols) is actually *faster* in practice. The dictionary is represented as the concatenation of the words encoded in 16-bit characters to fit the larger alphabet $\Sigma' = [0, 511)$. The dictionary size bound D is chosen to be 2^{16} so that the word endpoints can be encoded in 16-bit pointers. The small size of the dictionary also makes it more likely that (at least the most frequently accessed part of) it is kept in cache.

4. MONOTONE MINIMAL PERFECT HASH FOR STRINGS

Minimal perfect hash functions map a set of strings $\mathcal{S}$ bijectively into $[0, |\mathcal{S}|)$. *Monotone* minimal perfect hash functions [Belazzougui et al. 2009] (or *monotone hashes*) also require that the mapping preserves the lexicographic order of the strings (not to be confused with generic order-preserving hashing, where the order to be preserved is arbitrary, thus incurring a $\Omega(|\mathcal{S}| \log |\mathcal{S}|)$ space lower bound). We remark that as with standard minimal hash functions, the Lookup can return any number on strings outside of $\mathcal{S}$, and hence the data structure does not have to *store* the string set.

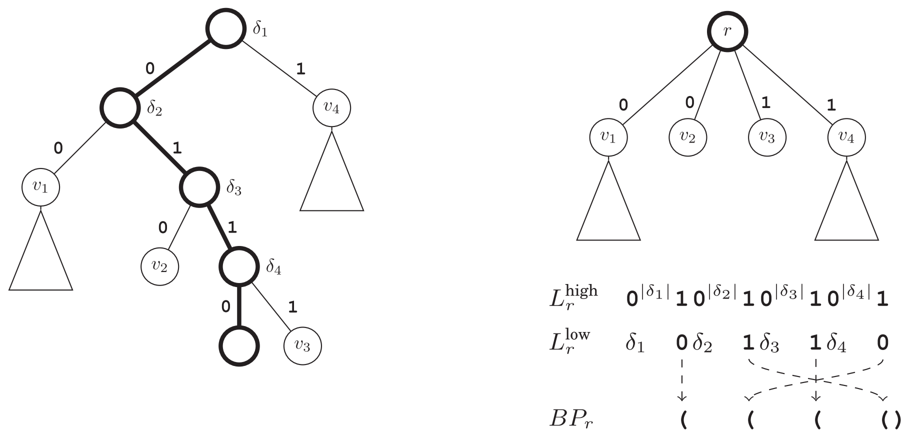

δ_1
δ_2
δ_3
δ_4
0
1
v_1
v_2
v_3
v_4
r
L_r^{high} $0^{|\delta_1|}10^{|\delta_2|}10^{|\delta_3|}10^{|\delta_4|}1$
L_r^{low} δ_1 $0\delta_2$ $1\delta_3$ $1\delta_4$ 0
BP_r (((()

Fig. 3. Path decomposition of a hollow trie. The δ_i denote the skips.

The hollow trie [Belazzougui et al. 2011] is a particular instance of monotone hash. It consists of a binary trie on $\mathcal{S}$, of which only the trie topology and the skips of the internal nodes are stored. The topology is encoded succinctly with FCNS; through the equivalence with DFBS described in Section 2.2, it is easy to see that the obtained sequence of parentheses is the same as the DFUDS encoding of the lexicographic path decomposition of the trie. To compute the hash value of a string x, a *blind search* is performed: the trie is traversed matching only the branching characters (bits, in this case) of x. If $x \in \mathcal{S}$, the leaf reached is the correct one, and its depth-first index is returned; otherwise, it has the longest prefix match with x, useful in some applications [Ferragina and Grossi 1999].

The cost of unbalancedness for hollow tries is even larger than that for normal tries: since the strings over Σ have to be converted to a binary alphabet, the height is potentially multiplied by $O(\log|\Sigma|)$ with respect to that of a trie on Σ. The experiments in Belazzougui et al. [2011] show indeed that the data structure has worse performance than the other monotone hashes, although it is among the most space-efficient.

Path decomposition with lexicographic order. To tackle their unbalancedness, we apply the centroid path decomposition idea to hollow tries. The construction presented in Section 3 cannot be used directly, because we want to both preserve the lexicographic ordering of the strings *and* guarantee the logarithmic height. However, both the binary alphabet and the fact that we do not need the Access operation come to the aid. First, inspired again by Ferragina et al. [2008], we arrange the subtries in *lexicographic* order. This means that the subtries on the *left* of the path are arranged from top to bottom and precede all those on the *right* that are arranged from bottom to top. In the path decomposition tree, we call *left* children the ones corresponding to subtries hanging off the left side of the path and *right* children the ones corresponding to those hanging on the right side. Figure 3 shows the new ordering.

We now need a small change in the heavy path strategy: instead of breaking ties arbitrarily, we choose the *left* child. We call this strategy *left-biased heavy path*, which gives the following.

OBSERVATION 4.1. *Every node-to-leaf left-biased heavy path in a binary trie ends with a left turn. Hence, every internal node of the resulting path decomposition has at least one right child.*

PROOF. Suppose by contradiction that the path leaf is a right child. Then, either its left sibling is not a leaf, in which case the path is not heavy, or it is a leaf, in which case the tie would be resolved by choosing the left child. □

Tree representation. The bit vector *BP* is defined as in Section 3. The label associated with each node is the sequence of skips interleaved with directions taken in the centroid path, excluding the leaf skip, as in Figure 3. Two aligned bit vectors L^{high} and L^{low} are used to represent the labels using an encoding inspired by γ codes [Elias 1975]: the skips are incremented by one (to exclude 0 from the domain), and their binary representations (without the leading `1`) are interleaved with the path directions and concatenated in L^{low}. L^{high} consists of `0` runs of length corresponding to the lengths of the binary representations of the skips, followed by `1`s, so that the endpoints of (skip, direction) pair encodings in L^{low} are aligned to the `1`s in L^{high}. Thus, a Select directory on L^{high} enables random access to the (skip, direction) pairs sequence. The labels of the node are concatenated in depth-first order: the `(`s in *BP* are in one-to-one correspondence with the (skip, direction) pairs.

Trie operations. As in Section 3, a trie traversal is simulated on the path decomposition tree. In the root node, the (skip, direction) pairs sequence is scanned (through L^{high} and L^{low}). During the scan the number of left and right children passed by is kept; when a mismatch in the string is found, the search proceeds in the corresponding child. Because of the ordering of the children, if the mismatch leads to a left child, then the child index is the number of left children seen in the scan, whereas if it leads to a right child, then it is the node degrees minus the number of right children seen (because the latter are represented from right to left). This correspondence between (skip, direction) pairs and child nodes (represented by `(`s in BP_v) is shown with dashed arrows in Figure 3. The search proceeds recursively until the string is fully traversed.

When the search ends, the depth-first order of the node found is not yet the number that we are looking for: all ancestors where we turned *left* come before the found node in depth-first but *after* it in the lexicographic order. Besides, if the found node is not a leaf, all strings in the left subtries of the corresponding path are lexicographically smaller than the current string. It is easy to fix these issues: during the traversal, we can count the number of left turns and subtract that from the final index. To account for the left subtries, using Observation 4.1, we can count the number of their leaves by jumping to the first right child with a FindClose: the number of nodes skipped in the jump is equal to the number of leaves in the left subtries of the node.

Time complexity. The running time of Lookup can be analyzed with a similar argument to that of the Lookup of Section 3. During the scan, there cannot be more skips than the string length; besides, there is no binary search. Hence, the number of operations is $O(\min(p, h))$, whereas the number of random memory accesses is bounded by $O(h)$.

Implementation notes. To support the Select on L^{high}, we use a variant of the *darray* [Okanohara and Sadakane 2007]. Since the `1`s in the sequence are at most 64 bits apart, we can bound the size of the blocks so that we do not need the overflow vector (called S_l in Okanohara and Sadakane [2007]).

5. BALANCED PARENTHESES: THE RANGE MIN TREE

In this section, we describe our variant of the data structure to support FindClose and FindOpen. As it is not restricted to tries, we believe that it is of independent interest.

We begin by discussing the range min-max tree [Sadakane and Navarro 2010], which is a succinct data structure to support operations on balanced parentheses in $O(\log n)$

time. It was shown in Arroyuelo et al. [2010] that it is very efficient in practice. Specifically, it is a data structure on $\{-1, 0, +1\}$ sequences that supports the *forward search* FwdSearch(i, x): given a position i and a target value x, return the least position $j > i$ such that the cumulative sum of the sequence at position j is equal to x.

The application to balanced parentheses is straightforward. If the sequence takes value $+1$ on open parentheses on -1 on close parentheses, the cumulative sum of the sequence at position i is called the *excess*, and it represents the difference between the number of open and close parentheses up to the position i.

Then, FindClose(i) = FwdSearch$(i$, Excess$(i)) - 1$. In other words, it is the left-most position j following i such that between i and j there is the same number of open and close parentheses.

Backward search is defined symmetrically, returning the right-most position preceding i, which has the desired cumulative sum; it can be used likewise to implement FindOpen.

The data structure is defined as follows. The sequence is divided into *blocks* of the same size; for each block, i are stored the minimum m_i and the maximum M_i of the cumulative sum of the sequence (for balanced parentheses, the excess) within the block. A *tree* is formed over the blocks, which become the leaves, and each node stores the minimum and the maximum cumulative sum among the leaves of its subtree.

To perform the forward search, we define the *target value* y as the cumulative sum of the sequence at i, plus x; the result of the forward search is of the left-most occurrence of the target value y following i in the sequence of the cumulative sums. To find its position, we traverse the tree to find the first block k following i where the value x plus the cumulative sum of the sequence at i is between m_k and M_k. Since the sequence has values in $\{-1, 0, +1\}$, the block k contains all intermediate values between m_k and M_k, and so it must contain y. A linear search is then performed within the block.

We fit the preceding data structure to support only FindOpen and FindClose, thus reducing both the space requirements and the running times. We list our two modifications.

Halving the tree space occupancy. We discard the maxima and store only the minima, and call the resulting tree *range min tree*. During the block search, we only check that the target value is greater than the block minimum. The following lemma guarantees that the forward search is correct. A symmetric argument holds for the backward search.

LEMMA 5.1. *Let* j = FwdSearch(i, x) *for* $x \leq 0$. *Then, the range min tree block search finds the block that contains* j.

PROOF. Since $x \leq 0$ and the sequence has values in $\{-1, 0, +1\}$, the values of the cumulative sums following i and preceding j must all be greater than the cumulative sum at j. Hence, the left-most block k that has minimum smaller than y must also contain y. □

Discarding the node maxima effectively halves the space occupancy of the tree, compared to the range min-max tree. Although we expected that the reduction in space would also be beneficial for cache locality, the experiments did not show any measurable improvement in query times.

Broadword in-block search. The in-block forward and backward search performance is crucial, as it is the inner loop of the search. In practical implementations [Arroyuelo et al. 2010; SDSL 2010], it is usually performed byte by byte with a lookup table that contains the solution for each possible combination of byte and excess (hence, $256 * 8 = 2{,}048$ bytes). This algorithm, which we will call *Lookup/Loop* in the following, involves many branches and accesses to a fairly big lookup tables for each byte. Suppose, instead, that we know which byte contains the closing parenthesis; we can then use the lookup table only on that byte.

To find that byte, we can use the same trick as in the range min tree: the first byte with min-excess smaller than the target excess must contain the closing parenthesis. This byte can be found by using a precomputed lookup table with the min excess of every possible byte (hence, 256 bytes) and checking all bytes in the word with a loop while updating the current excess at each loop iteration by using a second lookup table. We call this algorithm *RangeMin/Loop*.

The preceding algorithm still involves some hard-to-predict branches inside the loop. To get rid of them, we use a hybrid lookup table/broadword approach, which we call *RangeMin/Broadword*.

We divide the block into machine words. For each word w, we compute the word m_8 where the i-th byte contains the min excess of the i-th byte in w with inverted sign, so that it is nonnegative, by using the same precomputed lookup table used in RangeMin/Loop. At the same time, we compute the byte counts c_8 of w, where the i-th byte contains the number of `1`s of the i-th byte of w, using a broadword algorithm [Knuth 2009].

Using the equality $\text{Excess}(i) = 2 \cdot \text{Rank}_{(}(i) - i$, we can easily compute the excess for each byte of w: if e_w is the excess at the starting position of w, the word e_8 whose i-th byte contains the excess of the i-th byte of w can be obtained through the following formula:[3]

$$e_8 = (e_w + ((2 * c_8 - \texttt{0x...08080808} \ \texttt{<<}\ 8)) * \texttt{0x...01010101}.$$

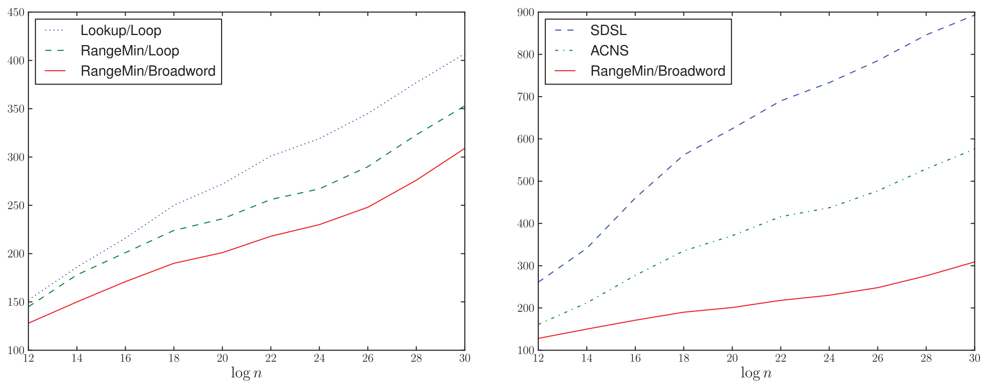
Now we have all we need: the closing parenthesis is in the byte where the excess function crosses the target excess—in other words, in the byte whose excess added to the min excess is smaller than the target. Hence, we are looking for the first byte position in which e_8 is smaller than m_8 (recall that the bytes in m_8 are negated). This can be done using the $\leq_8$ operation described in Knuth [2009] to compute a mask $l_8 = e_8 \leq_8 m_8$, where the i-th byte is 1 if and only if the i-th byte of e_8 is smaller than the i-th byte of m_8. If the l_8 is zero, the word does not contain the closing parenthesis; otherwise, an LSB operation quickly returns the index of the byte containing the solution. The same algorithm can be applied symmetrically for the FindOpen.

Overall, for 64-bit words, we perform eight lookups from a very small table, a few tens of arithmetic operations, and one single branch (to check whether the word contains the solution or not). In the following section, we show that RangeMin/Broadword is significantly faster than Lookup/Loop.

5.1. Range Min Tree Implementations

To evaluate the performance of different BP implementations, we generate a set of random binary trees of several sizes and measure the average FindClose time while performing random root-to-leaf traversals, mimicking the typical operations performed while traversing tries. Both the random trees and the random path directions are equal across benchmarks of different implementations. Each result is averaged over 20 runs on different random trees, and the lengths of the random paths sum up to 10 millions per run. In our range min tree, we use a block size of 512 bits.

[3]A subtlety is needed here to prove that the search is correct: the excess can be negative, and hence the carry in the subtraction corrupts the bytes after the first byte that contains the zero. However, this means that the word *contains* the solution, and the closing parenthesis is in the byte that precedes the one where the sampled excess becomes negative.



(a) Comparison of Lookup/Loop, RangeMin/Loop, and RangeMin/Broadword

(b) Comparison against other implementations

Fig. 4. Average FindClose time in nanoseconds for increasing sequence length.

Table I. Average Space Redundancy and FindClose Time in Nanoseconds for Different Implementations

		Average FindClose time (ns)									
log n	Redundancy	12	14	16	18	20	22	24	26	28	30
SDSL [SDSL 2010]	42%	261	341	460	562	624	690	733	785	846	893
ACNS [Arroyuelo et al. 2010]	15%	161	212	277	335	371	416	437	477	529	576
Lookup/Loop	32%	152	186	216	250	272	301	319	345	377	407
RangeMin/Loop	32%	145	178	201	224	236	256	267	290	323	353
RangeMin/Broadword	32%	128	150	171	190	201	218	230	248	276	309

Note: Each implementation was tested on sequences of different lengths, with log n being the logarithm of the sequence length.

We benchmarked our implementations of Lookup/Loop, RangeMin/Loop, and RangeMin/Broadword, the implementation of Arroyuelo et al. [2010], which we refer to as ACNS, and the implementation of SDSL [2010], which we refer to as SDSL. We could not compile ACNS and SDSL under Windows, so these experiments were run on an Intel Core i5 1.6Ghz running Mac OS X Lion, compiled with the stock compiler GCC 4.2.1. The results are shown in Figure 4 and Table I.

On the longest sequences, RangeMin/Loop nearly 20% faster than Lookup/Loop, whereas RangeMin/Broadword is nearly 15% faster than RangeMin/Loop. Overall, RangeMin/Broadword approximately 30% faster than RangeMin/Loop. We also tested our implementations with newer versions of GCC and MSVC, and found that the performance difference between RangeMin/Loop and RangeMin/Broadword vanishes, still being 20% to 30% faster than Lookup/Loop, due to better optimizations. This suggests that the choice between RangeMin/Loop and RangeMin/Broadword should depend on the particular compiler/architecture combination, with RangeMin/Broadword being a safe default.

When compared with other implementations, our implementation with RangeMin/Broadword is both faster and smaller than SDSL, and faster than ACNS, while occupying slightly more space. It should be noted, however, that the difference in space is entirely caused by the data structure used to support Rank (and thus Excess): whereas ACNS uses an implementation with an 8% overhead, we use the broadword data structure of Vigna [2008], which has a 25% overhead. If we account for this, the space used by the two implementations for supporting BP operations is roughly the same.

Table II. Average Height

	`enwiki-titles`	`aol-queries`	`uk-2002`	`webbase-2001`	`synthetic`
Compacted trie avg. height	9.8	11.0	16.5	18.1	504.4
Lex. avg. height	8.7	9.9	14.0	15.2	503.5
Centroid avg. height	5.2	5.2	5.9	6.2	2.8
Hollow avg. height	49.7	50.8	55.3	67.3	1,005.3
Centroid hollow avg. height	7.9	8.0	8.4	9.2	2.8

Note: For tries, the average height of the *leaves* is considered, whereas for path-decomposed tries, all nodes are considered (see the comments after Observation 3.2).

6. EXPERIMENTAL ANALYSIS

In this section, we discuss a series of experiments that we performed on both real-world and synthetic data. We performed several tests both to collect statistics that show how our path decompositions give an algorithmic advantage over standard tries and to benchmark the implementations comparing them with other practical data structures.

Setting. The experiments were run on a 64-bit 2.53GHz Core i7 processor with 8MB L3 cache and 24GB RAM, running Windows Server 2008 R2. All of the C++ code was compiled with MSVC 10, whereas for Java we used the Sun JVM 6.

Datasets. The tests were run on the following datasets:

—`enwiki-titles` (163MiB, 8.5M strings): All page titles from English Wikipedia.
—`aol-queries` (224MiB, 10.2M strings): Queries in the AOL 2006 query log [AOL 2006].
—`uk-2002` (1.3GiB, 18.5M strings): URLs of a 2002 crawl of the `.uk` domain [Boldi et al. 2004].
—`webbase-2001` (6.6GiB, 114.3M strings): URLs in the Stanford WebBase from LAW [2011].
—`synthetic` (1.4GiB, 2.5M strings): Set of strings $\mathtt{d}^i \mathtt{c}^j \mathtt{b}^t \sigma_1 \ldots \sigma_k$, where i and j range in $[0, 500)$, t ranges in $[0, 10)$, σ_i are all distinct (but equal for each string), and $k = 100$. The resulting tries are very unbalanced, while at the same time, the strings are extremely compressible. Furthermore, the constant suffix $\sigma_1 \ldots \sigma_k$ stresses the linear search in data structures based on front coding such as HTFC, and the redundancy is not exploited by front coding and noncompressed tries.

Average height. Table II compares the average height of plain tries with their path decomposition trees. In all of the real-world datasets, the centroid path decomposition cause nearly equal to a two to three times reduction in height compared to the standard compacted trie. The gap is even more dramatic in hollow tries, where the binarization of the strings causes a blow up in height close to $\log |\Sigma|$, whereas the centroid path-decomposed tree height is very small, actually much smaller than $\log |\mathcal{S}|$. It is interesting to note that even if the lexicographic path decomposition is unbalanced, it still improves on the trie, due to the higher fan-out of the internal nodes.

The synthetic dataset is a pathological case for tries, but the centroid path-decomposition still maintains an extremely low average height.

String dictionary data structures. We compared the performance of our implementations of path-decomposed tries to other data structures. *Centroid* and *Centroid compr.* implement the centroid path-decomposed trie described in Section 3, in the versions without and with labels compression. Likewise, *Lex.* and *Lex. compr.* implement the lexicographic version.

Re-Pair and *HTFC* are the re-pair and Hu-Tucker compressed front coding, respectively, from Brisaboa et al. [2011]. For HTFC, we chose bucket size 8 as the best space/time trade-off. Comparison with front coding is of particular interest, as it is one of the data structures generally preferred by the practitioners.

TX [TX 2010] is a popular open-source straightforward implementation of a (non-compacted) trie that uses LOUDS [Jacobson 1989] to represent the tree. We made some small changes to avoid keeping the whole string set in memory during construction.

To measure the running times, we chose 1 million random (and randomly shuffled) strings from each dataset for the *Lookup* and 1 million random indices for the Access. Each test was averaged on 10 runs. The construction time was averaged on 3 runs.

Re-Pair, HTFC, and TX do not support files bigger than 2GiB, so we could not run the tests on `webbase-2001`. Furthermore, Re-Pair did not complete the construction on `synthetic` in 6 hours, so we had to kill the process.

String dictionaries results. The results of the tests can be seen in Table III. On all datasets, our compressed tries obtain the smallest space, except on `uk-2002` where they come a close second. The centroid versions have also the fastest Lookup times, whereas the Access time is better for Re-Pair and occasionally HTFC, whose time is although within 20% of that of the centroid trie. TX is consistently the largest and slowest on all datasets.

Maybe surprisingly, the lexicographic trie is not much slower than the centroid trie for both Lookup and Access on real-world datasets. However, on the synthetic dataset, the unbalanced tries are more than 20 times slower than the balanced ones. HTFC exhibits a less dramatic slowdown but still in the order of 5x on lookup compared to the centroid trie. Although this behavior does not occur on our real-world datasets, it shows that no assumptions can be made for unbalanced tries. For example, in an adversarial environment, an attacker could exploit this weakness to perform a denial of service attack.

We remark that the labels compression adds an almost negligible overhead in both Lookup and Access, due to the extremely simple dictionary scheme, while obtaining a very good compression. Hence, unless the construction time is a concern (in which case other dictionary selection strategies can also be explored), it is always convenient to compress the labels.

Monotone hash data structures. For monotone hashes, we compared our data structures with the implementations in Belazzougui et al. [2011]. *Centroid hollow* implements the centroid path-decomposed hollow trie described in Section 4. *Hollow* is a reimplementation of the hollow trie of Belazzougui et al. [2011], using a range min tree in place of a pioneer-based representation and the encoding described in Section 2.2. *Hollow (Sux)* and *PaCo (Sux)* are two implementations from Belazzougui et al. [2011]; the first is the hollow trie, the second a hybrid scheme: a partially compacted trie is used to partition the keys into buckets, then each bucket is hashed with an MWHC function. Among the structures in Belazzougui et al. [2011], PaCo gives the best trade-off between space and lookup time. The implementations are freely available as part of the Sux project [SUX4J 2011].[4]

To measure construction time and lookup time, we adopted the same strategy as for string dictionaries. For Sux, as suggested in Belazzougui et al. [2011], we performed three runs of lookups before measuring the lookup time to let the JIT warm up and optimize the generated code.

[4]To be fair, we need to say that Sux is implemented in Java, whereas our structures are implemented in C++. However, the recent developments of the Java Virtual Machine have made the abstraction penalty gap smaller and smaller. Low-level optimized Java (as the one in Sux) can be on par of C++ for some tasks, and no slower than 50% with respect to C++ for most other tasks. We remark that the hollow trie construction is actually *faster* in the Sux version than in ours, although the algorithm is very similar.

Table III. Experimental Results

	`enwiki-titles`				`aol-queries`				`uk-2002`				`webbase-2001`				`synthetic`			
	161 bps				185 bps				621 bps				497 bps				4836 bps			
	String dictionaries																			
	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs
Centroid compr.	6.1	**32.1%**	**2.5**	2.6	7.9	**31.5%**	2.7	2.7	8.5	**13.6%**	3.8	4.9	7.8	**13.5%**	4.8	**5.4**	13.7	**0.4%**	**4.2**	**13.5**
Lex. compr.	6.4	**31.9%**	3.2	3.1	8.0	**31.2%**	3.8	3.6	8.4	**13.5%**	5.9	6.6	8.5	**13.3%**	7.3	7.7	109.2	**0.4%**	90.9	96.3
Centroid	1.8	53.6%	**2.4**	2.4	2.0	55.6%	**2.4**	2.6	2.3	22.4%	**3.4**	4.2	**2.2**	24.3%	**4.3**	**5.0**	8.4	17.9%	5.1	**13.4**
Lex.	2.0	52.8%	3.1	3.2	2.2	55.0%	3.5	3.5	2.7	22.3%	5.5	6.2	2.6	24.3%	7.0	7.4	102.8	17.9%	119.8	114.6
Re-Pair [Brisaboa et al. 2011]	60.0	41.5%	6.6	**1.2**	115.4	38.8%	7.3	**1.3**	326.4	**12.4%**	25.7	**3.1**	—	—	—	—	—	—	—	—
HTFC [Brisaboa et al. 2011]	**0.4**	43.2%	3.7	2.2	**0.4**	40.9%	3.8	2.2	**0.9**	24.4%	7.0	4.7	—	—	—	—	**5.0**	19.1%	22.0	18.0
TX [TX 2010]	2.7	64.0%	9.7	9.1	3.3	69.4%	11.9	11.3	5.7	30.0%	42.1	42.0	—	—	—	—	44.6	25.3%	284.3	275.9

	Monotone hashes														
	ctps	bps	lkp	ctps	bps	lkp	ctps	bps	lkp	ctps	bps	lkp	ctps	bps	lkp
Centroid hollow	1.1	**8.40**	2.7	1.2	**8.73**	**2.8**	1.5	8.17	**3.3**	1.5	8.02	**4.4**	8.6	9.96	**11.1**
Hollow	1.3	**7.72**	6.8	1.3	**8.05**	7.2	1.7	**7.48**	9.3	1.7	**7.33**	13.9	9.5	9.02	137.1
Hollow [SUX4J 2011]	**0.9**	**7.66**	14.6	**1.0**	**7.99**	16.6	**1.1**	**7.42**	18.5	**0.9**	**7.27**	22.4	**4.3**	**6.77**	462.7
PaCo [SUX4J 2011]	2.6	8.85	**2.4**	2.9	8.91	3.1	4.7	10.65	4.3	18.4	9.99	4.9	21.3	13.37	51.1

Note: *bps* is *bits per string*, *ctps* is the average *construction time* per string, *c.ratio* is the *compression ratio* between the data structure and the original file sizes, *lkp* is the average Lookup time, and *acs* the average Access time. All times are expressed in microseconds. The results within 10% of the best are in bold.

Monotone hash results. Table III shows the results for monotone hashes. On all real-world datasets, the centroid hollow trie is nearly equal to 2 to 3 times faster than our implementation of the hollow trie and nearly equal to 5 times faster than the Sux implementation. The centroid hollow trie is competitive with PaCo on all datasets while taking less space and with a substantially simpler construction. The synthetic dataset in particular triggers the pathological behavior on all of the unbalanced structures, with *Hollow*, *Hollow (Sux)* and *PaCo* being respectively 13, 41, and 5 times slower than *Centroid hollow*. Such a large performance gap suggests the same conclusion reached for string dictionaries: if *predictable* performance is needed, unbalanced structures should be avoided.

7. CONCLUSION AND FUTURE WORK

We have presented new succinct representations for tries that guarantee low average height and enable the compression of the labels. Our experimental analysis has shown that they obtain the best space occupancy when compared to the state of the art while maintaining competitive access times. Moreover, they give the most *consistent* behavior among different (possibly synthetic) datasets. We have not considered alternatives for the dictionary selection algorithm in labels compression; any improvement in that direction would be beneficial to the space occupancy of our tries.

We considered other kinds of path decompositions (longest path, ladder, etc.) as candidates for improvement; from preliminary experiments, such path decompositions did not seem to provide promising alternatives on our time/space/functionality trade-offs while retaining simplicity in the implementation, giving rise to tall decomposition trees or not preserving the lexicographic order. An alternative strategy, the *max-score* path decomposition, was recently shown to enable fast *top-k completion queries* for scored string sets [Hsu and Ottaviano 2013].

In principle, it is possible to enforce the lexicographic order on the centroid path decomposition (as presented theoretically in Ferragina et al. [2008]), but this does not seem to be of practical value. It is an interesting problem to find a path decomposition that combines the simplicity and performance of both lexicographic and centroid path decompositions.

ACKNOWLEDGMENTS

We would like to thank the authors of Brisaboa et al. [2011] for kindly providing the source code for their algorithms.

REFERENCES

Anurag Acharya, Huican Zhu, and Kai Shen. 1999. Adaptive algorithms for cache-efficient trie search. In *Algorithm Engineering and Experimentation*. Lecture Notes in Computer Science, Vol. 1619. Springer, 296–311.

AOL. 2006. AOL Search Data. Retrieved September 9, 2014, from http://www.gregsadetsky.com/aol-data/.

Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. 2010. Succinct trees in practice. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 84–97.

Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2009. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*. 785–794.

Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2011. Theory and practice of monotone minimal perfect hashing. *ACM Journal on Experimental Algorithmics* 16, Article No. 3.2. DOI:http://dx.doi.org/10.1145/1963190.2025378

Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2006. Cache-oblivious string B-trees. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, New York, NY, 233–242.

David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing trees of higher degree. *Algorithmica* 43, 4, 275–292.

Daniel K. Blandford and Guy E. Blelloch. 2008. Compact dictionaries for variable-length keys and data with applications. *ACM Transactions on Algorithms* 4, 2, Article No. 17.

Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A scalable fully distributed Web crawler. *Software: Practice and Experience* 34, 8, 711–726.

Nieves R. Brisaboa, Rodrigo Cánovas, Francisco Claude, Miguel A. Martínez-Prieto, and Gonzalo Navarro. 2011. Compressed string dictionaries. In *Proceedings of the 10th International Conference on Experimental Algorithms (SEA'11)*. 136–147.

Gerth Stølting Brodal and Rolf Fagerberg. 2006. Cache-oblivious string dictionaries. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06)*. ACM, New York, NY, 581–590.

Sheng-Yuan Chiu, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. 2010. I/O-efficient compressed text indexes: From theory to practice. In *Proceedings of the 2010 Data Compression Conference (DCC'10)*. IEEE, Los Alamitos, CA, 426–434.

David Richard Clark. 1998. *Compact Pat Trees*. Ph.D. Dissertation. University of Waterloo, Waterloo, Ontario, Canada.

Francisco Claude and Gonzalo Navarro. 2010. Fast and compact Web graph representations. *ACM Transactions on the Web* 4, 4, Article No. 16.

Pooya Davoodi, Rajeev Raman, and S. Srinivasa Rao. 2012. Succinct representations of binary trees for range minimum queries. In *Proceedings of the 18th Annual International Conference on Computing and Combinatories (COCOON)*. 396–407.

Peter Elias. 1974. Efficient storage and retrieval by content and address of static files. *Journal of the ACM* 21, 2, 246–260.

Peter Elias. 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory* 21, 2, 194–203.

Robert M. Fano. 1971. On the number of bits required to implement an associative memory. Memorandum 61. Computer Structures Group, Project MAC. MIT, Cambridge, MA.

Paolo Ferragina and Roberto Grossi. 1999. 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.

Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. 2008. On searching compressed string collections cache-obliviously. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'08)*. 181–190.

Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. 2009. Compressing and indexing labeled trees, with applications. *Journal of the ACM* 57, 1, Article No. 4.

Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing* 35, 2, 378–407.

Bo-June (Paul) Hsu and Giuseppe Ottaviano. 2013. Space-efficient data structures for top-k completion. In *Proceedings of the 22th International Conference on World Wide Web (WWW'13)*. 583–594.

Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS'89)*. 549–554.

Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. 2012. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences* 78, 2, 619–631.

Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley, Reading, MA.

Donald E. Knuth. 2009. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks and Techniques; Binary Decision Diagrams* (12th ed.). Addison-Wesley Professional.

N. Jesper Larsson and Alistair Moffat. 1999. Offline dictionary-based compression. In *Proceedings of the Data Compression Conference*. 296–305.

LAW. 2011. Laboratory for Web Algorithmics—Datasets. Retrieved September 9, 2014, from law.dsi.unimi.it/datasets.php.

LIBCDS. 2011. LIBCDS—Compact Data Structures Library. Retrieved September 9, 2014, from https://github.com/fclaude/libcds.

J. Ian Munro and Venkatesh Raman. 2001. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing* 31, 3, 762–776. DOI: http://dx.doi.org/10.1137/S0097539799364092

Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*.

Kunihiko Sadakane and Gonzalo Navarro. 2010. Fully-functional succinct trees. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*. 134–149.

SDSL. 2010. SDSL 0.9—Succinct Data Structure Library. Retrieved September 9, 2014, from http://www.uni-ulm.de/in/theo/research/sdsl.html.

Daniel Dominic Sleator and Robert Endre Tarjan. 1981. A data structure for dynamic trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC'81)*. 114–122.

SUCCINCT. 2012. Succinct Library. Retrieved September 9, 2014, from http://github.com/ot/succinct.

SUX4J. 2011. Sux 0.7 and Sux4J 2.0.1—Implementing Succinct Data Structures. Retrieved September 9, 2014, from http://sux.di.unimi.it/.

TX. 2010. Tx 0.18—Succinct Trie Data Structure. Retrieved September 9, 2014, from http://code.google.com/p/tx-trie/.

Sebastiano Vigna. 2008. Broadword implementation of rank/select queries. In *Proceedings of the 7th International Conference on Experimental Algorithms (WEA'08)*. 154–168.

Hugh E. Williams and Justin Zobel. 1999. Compressing integers for fast file access. *Computer Journal* 42, 3, 193–201.

Received April 2012; revised July 2012; accepted July 2014