On the Compression of Search Trees[☆,☆☆]

Francisco Claude[a], Patrick K. Nicholson[b], Diego Seco[c,*]

[a]*Escuela de Informática y Telecomunicaciones, Universidad Diego Portales, Chile*
[b]*Cheriton School of Computer Science, University of Waterloo, Canada*
[c]*Departamento de Ingeniería Informática y Ciencias de la Computación, Universidad de Concepción, Chile*

Abstract

Let $X = x_1, x_2, \ldots, x_n$ be a sequence of non-decreasing integer values. Storing a compressed representation of X that supports *access* and *search* is a problem that occurs in many domains. The most common solution to this problem uses a linear list and encodes the differences between consecutive values with encodings that favor small numbers. This solution includes additional information (i.e. samples) to support efficient searching on the encoded values. We introduce a completely different alternative that achieves compression by encoding the differences in a search tree. Our proposal has many applications, such as the representation of posting lists, geographic data, sparse bitmaps, and compressed suffix arrays, to name just a few. The structure is practical and we provide an experimental evaluation to show that it is competitive with the existing techniques.

Keywords: Data Structure, Compression, Integers, Random access, Search

[☆]A preliminary partial version on this work appeared in Proc. DCC'12 pp. 357–366.

[☆☆]This work was supported in part by the Google US/Canada PhD Fellowship Program, the David R. Cheriton Scholarships Program, and CONICYT Fondecyt Iniciación 11130104 (first author); an NSERC of Canada PGS-D Scholarship (second author); and CONICYT Fondecyt Iniciación 11130377 and Ministerio de Ciencia e Innovación (PGE/FEDER) grant TIN2009-14560-C03-02 (third author). Work partially done while the authors were at the David R. Cheriton School of Computer Science, University of Waterloo, Canada.

[*]Corresponding author. Tel.: +56 (41) 220 4692; fax: +56 (41) 221 770

Email addresses: `fclaude@recoded.cl` (Francisco Claude), `p3nichol@cs.uwaterloo.ca` (Patrick K. Nicholson), `dseco@udec.cl` (Diego Seco)

1. Introduction

The storage of ordered sets of integers is a fundamental problem in computer science that has applications in many domains. When space is not an issue these sets can be stored in arrays, which support random access and efficient searches. However, space is a constraining factor in most domains, and the compression of these sets can save a considerable amount of space. As compression techniques are usually based on variable length encoding, random access and efficient searches become challenging.

Common solutions to this problem rely on the fact that differences between consecutive values in a sequence are often small numbers, and encode these differences with encodings that favor small numbers. In order to efficiently support random access and searches, these schemes are forced to store sampled absolute values. These samples can be thought as an index built on top of the data that requires more space the higher the sampling rate. As the sampling rate is a parameter that provides a space-time trade-off, these solutions are called *parametric*.

In this paper we propose a (*non-parametric*) compressed representation for non-decreasing sets of integers that does not require us to build an index on top of the data, and still offers efficient support for random access and searches. In this sense our method is similar to a recent proposal by Teuhola (2011), but the techniques are different and of independent interest. Let $X = x_1, x_2, \ldots, x_n$ be a sequence of non-decreasing integer values, we propose a compressed representation of X that, in logarithmic time, supports:

- access(X, i): retrieve the value at position i in X.
- search(X, t): retrieve the position of the left-most stored value greater or equal than t[1].

This structure has applications in the representation of posting lists, which are one of the two main components of the ubiquitous inverted index (Baeza-Yates & Ribeiro-Neto, 1999; Witten et al., 1999). A basic primitive operation for these indexes is to find the intersection of posting lists, which provides a solution to handle multi-word queries. Although this primitive operation can be supported through sequential merging of the lists, some

[1]Note that we can also return the value stored in that position, thus solving the membership variant of the problem.

of the most efficient approaches to solve intersection queries (Hwang & Lin, 1972; Demaine et al., 2000; Barbay & Kenyon, 2002; Baeza-Yates, 2004) take advantage of the fact that some lists are much shorter than the others, and make use of random access and search capabilities.

As another application, the partial sums problem can be thought of as a particular instance of this problem. In the partial sums problem we are given a sequence of n non-negative values, $Y = y_1, y_2, \ldots, y_n$, and we have to support the following operations:

- $\mathrm{sum}(Y, i)$: retrieve the sum of all the values up to position i.
- $\mathrm{search}(Y, t)$: retrieve the smallest i such that $\mathrm{sum}(Y, i) \geq t$.

Note that $X = x_1, x_2, \ldots, x_n$ can be defined as a sequence of partial sums of values in the sequence $Y = y_1, y_2, \ldots, y_n$, so that $x_i = \sum_{j=1}^{i} y_j$. In this way, $\mathrm{sum}(Y, i)$ reduces to $\mathrm{access}(X, i)$ and $\mathrm{search}(Y, t)$ reduces to $\mathrm{search}(X, t)$. Partial sums can be applied, for example, to represent *Rank*/*Select* dictionaries (Okanohara & Sadakane, 2007).

The article is organized as follows. In Section 2 we survey the main solutions to this problem and describe their basic properties. In Section 3 we describe our new solution to this problem. Section 4 discusses some applications, and presents an experimental evaluation of our solution both in general and for each particular application. Finally, Section 5 concludes the paper with some brief remarks, and avenues for future research.

2. Related Work

As we mentioned above, common solutions to this problem encode the differences between consecutive values with encodings that favor small numbers, such as γ-codes, δ-codes, or Rice codes (see Witten et al. (1999) for a comprehensive survey). In order to efficiently support random access and searches, these schemes store sampled absolute values, and the sampling rate provides a space-time trade-off. A great deal of research has been carried out on the development of new algorithms to encode small numbers (Moffat & Stuiver, 2000; Anh & Moffat, 2005; Yan et al., 2009) and also on proposing storage schemes for those samplings (Culpepper & Moffat, 2010). All these techniques can be classified as *parametric* solutions, whereas our method is *non-parametric*.

Recently, Teuhola (2011) presented a *non-parametric* scheme based on the encoding of a binary search tree of the sequence using a variation of the interpolative coding (Moffat & Stuiver, 1996, 2000). The author turns interpolative coding into an index by an address calculation that guarantees enough space in the worst case (experiments show that the redundancy added by this address calculation is only about 1 bit per symbol). Although our proposal is also based on encoding the differences between values using a search tree, there are some important differences. First, our work is not dependent on a particular encoding (any encoding supporting efficient random access may be plugged into our structure). Second, we extend the representation, and add new interesting operations; for example, a batched searching operation used for intersecting inverted lists. We also provide a more elegant solution for representing the tree when n is not a power of 2. Furthermore, we extend this generalized result to trees of arbitrary constant degree, which is suitable for secondary memory. Finally, we describe an approach for handling general trees, with the potential of becoming dynamic.

Among all the encoding techniques for integers there is one of special interest for us because we use it as part of our solution. This technique was introduced by Brisaboa et al. (2013) and is known as Directly Addressable Codes (DACs). It performs a reorganization of variable-length codes in order to allow efficient random access. This technique solves the problem of supporting random access, but not the search problem. In order to support searches, the same sampling scheme mentioned above can be used. The main advantage of the use of DACs in these schemes is that, as they allow direct access to any position, no pointers from each sample to the subsequent code are necessary. Although this might mean a significant reduction of space, in many applications the lower compression rate achieved by these codes counteracts this improvement.

3. Differentially Encoded Search Tree (DEST)

The main idea behind our structure is to differentially encode the values inside a search tree built over the sequence of non-decreasing values, X. Every node stores the difference between its value and the one represented by its parent node. By knowing whether the node is a left or right child (this is for the binary case, we explore the general case further in this section), one can determine whether the difference is positive or negative.

The standard pointer-based tree representation has space requirements that would defy the purpose of our structure, due to the space required by the pointers, $O(\log n)$ bits per node. For this reason we use alternative representations throughout this work. In order to better cope with this restriction our structure separates the representation into two parts: *tree representation* and *encoded values*. The tree representation deals with navigating the tree and queries the encoded values to retrieve the information stored in the nodes. This allows us to plug different space-efficient tree representations with suitable integer encoding techniques.

In order to present a representation-independent solution, we define an abstract data type (ADT) tree T and enumerate the basic operations it supports (Section 3.1). In this section, we also explore two tree representations, binary (Section 3.2) and multiary (Section 3.3), both based on the folklore heap embedding in an array. In addition, we show how the multiary variant can be extended for general trees using succinct representations for trees (Section 3.4). Finally, in Section 3.5 we explore different encodings: fixed length at each level in the tree, Directly Addressable Codes (DACs), and combinations of both. We note that the selection of a tree representation and encoding is entirely application dependent.

3.1. Operations

Let $X = x_1, x_2, \ldots, x_n$ be a sequence of non-decreasing integer values and T be a differentially encoded search tree representing X. We assume T has $\lceil n/k \rceil$ nodes, each of arity $k+1$. The operations T supports are:

- root_T: obtain the root of the tree.
- $\text{fetch}_T(v, r)$: retrieve the value stored in node v at position $r = 0 \ldots k-1$. Recall that this is the difference between the real value, and that of the parent node.
- $\text{child}_T(v, r)$: find the r-th child of node v in T. Unlike the previous case, $r = 0 \ldots k$.
- $\text{parent}_T(v)$: find the parent of node v in T.
- $\text{subtree\_size}_T(v)$: count the number of values in the sub-tree rooted at node v.
- $\text{child\_rank}_T(v)$: compute the rank of node v among its siblings.

- $\text{map}_T(v, r)$: map the r-th value of node v to its position in the original sequence X.

In the subsequent sections we present several tree representations and encodings that support these basic operations in constant time. Here, we use that assumption and analyze the two operations of interest (i.e. $\text{access}(X, i)$ and $\text{search}(X, t)$).

We also assume that at all times we have a finger v pointing at the current node, which stores the real value $\mathbf{w}$ of the node (not its difference with the parent). Therefore, any tree traversal will start by accessing the root ($v = \text{root}_T$) and storing its value in $\mathbf{w}$. In the following discussion we assume k is a constant, and thus, this takes constant time. Note that in a multiary embedding $\mathbf{w} = (w_0, \ldots, w_{k-1})$ is a vector (the binary embedding is a particular case $\mathbf{w} = (w_0)$). We can move from v to its r-th child also in constant time. To do this, we set $v = \text{child}_T(v, r)$ and update $\mathbf{w}$ setting $w_j = w_r - \text{fetch}_T(v, j)$ if $r < k$ and $w_j = w_r + \text{fetch}_T(v, j)$ otherwise. We can move to the parent of node v in a similar way. Note that we only have to compute on the fly the values in $\mathbf{w}$ that we need, thus we are not forced to pay $O(k)$ every time we move.

Using the aforementioned operations and assumptions, we support the access to the i-th value in the set (i.e. $\text{access}(X, i)$) in logarithmic time. We only consider balanced trees. Otherwise, the complexities depend on the height of the tree. This can be done by traversing the tree using binary search at each node to determine the direction to move. These binary searches use the $\text{map}_T(v, r)$ operation to compare the target position i with the position in X that corresponds with the r-th value of v. This takes $O(\log k)$ time at each node, and thus the whole traversal takes $\lceil \log_{k+1} n \rceil \cdot O(\log k) = O(\log n)$ time. Algorithm 1 shows the pseudocode for this operation.

The last operation we consider is searching among the values of the original set X that is encoded in the tree T (i.e. $\text{search}(X, t)$). Note that this operation is essentially searching for a position where the key t can be inserted in the tree while preserving the order of the sequence. We traverse the tree in a similar way, but in this case we decide the direction to move by binary searching the real values of the node. This takes $O(\log k)$ time at each node for a total search time of $O(\log n)$. Algorithm 2 describes this traversal. We can also iterate over the values starting from a given finger, either forwards or backwards. This allows access to a range of contiguous values in $O(\log n + \ell)$ time, where ℓ is the length of the range.

Algorithm 1 access(X, i), T is the DEST representation of X

```
$v \leftarrow \mathrm{root}_T$ {$v$ is the current node}
$w \leftarrow 0$ {$w$ is the current value}
$pos \leftarrow -1$ {$pos$ is the position in $X$ of the current value}
$rightMostChild \leftarrow true$
while $pos <> i$ do
  $c \leftarrow$ successor_search$(v, i)$ {The successor search uses $\mathrm{map}_T(v, r)$ to map each position $r$ in node $v$ to its corresponding position in $X$ and compare it with $i$}
  $pos \leftarrow \mathrm{map}_T(v, c)$
  if $rightMostChild$ then
    $w' \leftarrow w + \mathrm{fetch}_T(v, c)$ {The root is a special case of $rightMostChild$}
  else
    $w' \leftarrow w - \mathrm{fetch}_T(v, c)$
  if $pos = i$ then
    return $w'$
  if $pos < i$ then
    $v \leftarrow \mathrm{child}_T(v, c + 1)$
    $rightMostChild \leftarrow true$
  else
    $v \leftarrow \mathrm{child}_T(v, c)$
    $rightMostChild \leftarrow false$
  $w \leftarrow w'$
```

3.2. Binary Heap-Like Embedding

Our first representation uses a perfectly balanced tree where all levels of the tree are full except the last one, in which only a contiguous prefix is filled with values.

It is well known that such a tree can be embedded in an array $A[1 \ldots n]$[2], where position 1 represents the root. The left and right children of a node at position v are at positions $2v$ and $2v + 1$, respectively, and its parent is at position $\left\lfloor \frac{v}{2} \right\rfloor$. The operation child_rank$_T(v)$ is computed as $v \mod 2$. These operations are the building blocks that allow us to navigate the tree. Some issues that we need to address in order to give a complete description of the

[2]Note that indexes start from 1 here, as this simplifies the formulas for navigating A.

Algorithm 2 search(X, t), T is the DEST representation of X

```
$v \leftarrow \text{root}_T$ {$v$ is the current node}
$w \leftarrow 0$ {$w$ is the current value}
$rightMostChild \leftarrow true$
**while** $v$ is a node **do**
    $c \leftarrow$ successor_search$(v, t)$ {The successor search uses $w$ and $\text{fetch}_T(v, r)$
    to decompress each value at position $r$ in node $v$}
    **if** $rightMostChild$ **then**
        $w' \leftarrow w + \text{fetch}_T(v, c)$ {The root is a special case of $rightMostChild$}
    **else**
        $w' \leftarrow w - \text{fetch}_T(v, c)$
    **if** $w' = t$ **then**
        **return** $\text{map}_T(v, c)$
    **if** $w' < t$ **then**
        $v \leftarrow \text{child}_T(v, c + 1)$
        $rightMostChild \leftarrow true$
    **else**
        $v \leftarrow \text{child}_T(v, c)$
        $rightMostChild \leftarrow false$
    $w \leftarrow w'$
**return** $n$ {The length of $X$, which indicates that $t$ is greater than all the
values of $X$}
```

structure are how to compute $\text{subtree\_size}_T(v)$ and $\text{map}_T(v, r)$, and how to build this structure efficiently.

Supporting subtree_size$_T(v)$. The tree has depth $h = \lceil \log_2(n + 1) \rceil$ and we can determine in constant time the height of a given node v by computing $h(v) = \lceil \log_2(v + 1) \rceil$. For each internal node v, the height of the subtree rooted at v is either $h - h(v) + 1$ or $h - h(v)$. We know that the subtree contains at least $2^{h-h(v)} - 1$ values, so we only need to count the number of nodes in the last level. In order to do that, we compute the positions p_ℓ and p_r corresponding with the leftmost and rightmost descendants of node v, respectively. Since the subtree rooted at v has height $h - h(v) + 1$, $p_\ell = v2^{h-h(v)}$ and $p_r = (v+1)2^{h-h(v)}-1$. By comparing p_ℓ with n, we can determine whether there are any nodes in level $h - h(v) + 1$. If there are, we can count them by computing $\min(n, p_r) - p_\ell + 1$. Thus, we can compute the size of

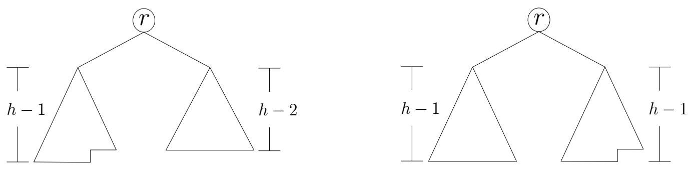

Figure 1: Illustration of possible cases in the binary heap-like embedding.

the subtree at any node in constant time.

Supporting $map_T(v, r)$. In the binary case, we can picture the $\mathrm{map}_T(v, 0)$ operation as computing the rank of the node (value) in the set, since every node stores a single value. Instead of supporting the node rank operation for an arbitrary node, the finger pointing to the current node stores its rank, in addition to its real value. Thus, we only need to update the rank when moving to another node and this can be done in constant time using the $\mathrm{subtree\_size}_T(v)$ operation.

Building the structure. Given a sorted array $B[1 \ldots n]$, we want to re-arrange B to obtain A, which represents the binary search tree where B is embedded. The main issue here is to determine the position in B where the root lies. Determining the root is simple if $n = 2^h - 1$, since it is exactly in the middle of the array at position 2^{h-1}. However, to give a practical construction we need to handle the general case, when n is not a power of 2. Assume $2^{h-1} - 1 < n < 2^h$. Figure 1 shows the possible scenarios that could arise. There are two cases:[3]

- Case $n < 3 \times 2^{h-2}$: This corresponds to the case on the left in Figure 1 and, in this case, the root corresponds to position $n - 2^{h-2} + 1$. Basically, we have two full binary trees of height $h - 2$, plus the root, plus the last level of the left subtree.
- Case $n \geq 3 \times 2^{h-2}$: This case is represented on the right side in Figure 1, and the root is at position 2^{h-1}.

[3]Note that when $n = 3 \times 2^{h-2} - 1$ the subtrees rooted at the children of the root are complete trees of height $h - 1$, the left subtree, and $h - 2$, the right one.

The value of h can be computed from n using the most significant bit of n, which can be done in constant time with $o(n)$ extra bits of space (Munro, 1996). Modern architectures also support this operation at hardware level. One interesting observation is that the root is at position $\max(2^\ell, n-2^\ell+1)$, where $\ell = \lfloor \log_2(n/3) \rfloor + 1$. This is equivalent to the two cases considered above. We note that we can even construct the embedding in place by following cycles in the permutation defined by these cases (Fich et al., 1995, Theorem 4).

Lemma 3.1. *Given a sorted array of n values, we can embed its values into a heap-like binary search tree or binary DEST in linear time.*

3.3. Multiary Heap-Like Embedding

The case of a multiary heap-shaped embedding is similar to that of the binary case. We consider a fixed constant fan-out of $k+1$ for each node, thus every node stores k values. The root is at position 1 in the array.

To move from node v to its r-th child, we compute $v(k+1)+rk$. Note that nodes are identified by the position of the first value, for example, for $k=3$ the 0-th child of the root is the node $v=4$. We can also compute the parent of node v as $k\left\lfloor \frac{k^2+v-1}{k(k+1)} \right\rfloor - (k-1)$. The computation of $\text{subtree\_size}_T(v)$ is also similar to that of binary trees. We move to the leftmost and rightmost descendants of position v, which are placed in the last level of a subtree of height $h-h(v)+1$, where $h(v) = \lceil \log_{k+1}(v+1) \rceil$. If the leftmost child is at a position $p_\ell = v(k+1)^{h-h(v)}$, greater than n, we know that the size of the subtree is that of a complete tree of height $h-h(v)$. Otherwise, given the position of the rightmost node, $p_r = p_\ell + k(k+1)^{h-h(v)} - 1$, the subtree contains $\min(n, p_r) - p_\ell + 1$ values.

We can compute $\text{map}_T(v, r)$ in a similar way, but in this case we move to the leftmost descendant of node v and to the rightmost descendant of the r-th child of v to compute the number of values in the last level. We add this number to the r values stored in node v up to position r plus the r complete subtrees of height $h-h(v)$ rooted at them.

The construction is slightly more complicated than in the binary case. We can compute the height $h = \lceil \log_{k+1}(n+1) \rceil$, and each of the $k+1$ subtrees of the root has height either $h-1$ or $h-2$. The ones that have height $h-2$ are complete trees. Thus, there are the equivalent of $k+1$ complete trees of height $h-2$ and some extra values that form the last level, so that we end

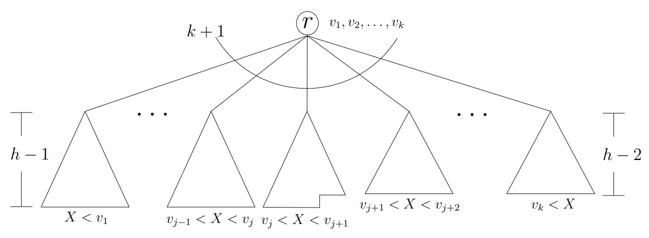

Figure 2: Illustration of the $(k+1)$-ary heap-like embedding.

up with $j-1$ complete trees of height $h-1$ (for some $j \in [2..k+2]$) and at most one possibly incomplete tree of height $h-1$ (see Figure 2).

The number of values in a complete tree of height $h-2$ is $(k+1)^{h-2}-1$. Let β be the number of values that fill the last level of the subtrees with height $h-1$, we can write n as $n = (k+1)^{h-1} - 1 + \beta$. The number of nodes in the last level of a tree of height $h-1$ is $k(k+1)^{h-2}$. From the previous paragraph, we get that $\beta = n + 1 - (k+1)^{h-1}$, and we can determine the value of j by computing $j = \left\lfloor \frac{\beta}{k(k+1)^{h-2}} \right\rfloor$. Subtree $j+1$ has $\gamma = \beta \mod \left(k(k+1)^{h-2}\right)$ values in level $h-1$. The special case of $\gamma = 0$ means that the j-th subtree has height $h-2$, and all the subtrees to the left are complete trees of height $h-1$.

Finally, to construct the $(k+1)$-ary tree from a sorted array B, we need to locate the k nodes that go into the root and then recurse into the $k+1$ chunks that represent the subtrees. To achieve this, we compute h, j, and γ, then we select the first j values every $(k+1)^{h-1}-1$ values apart, then we jump $(k+1)^{h-2} - 1 + \gamma$ to retrieve the next value, and finally, for the remaining values, we jump $(k+1)^{h-2}-1$ between each. This construction allows us to state the following lemma.

Lemma 3.2. *Given a sorted array of n values, we can embed its values into a heap-like $(k+1)$-ary search tree or multiary DEST in linear time.*

An observation about this particular tree shape is that it is suitable for secondary memory. If we store the data structure on a disk that has block size B, then we just need to set $k = B-1$ and the search I/O complexity becomes $O(\log_B n)$.

Theorem 3.3. *Given n values stored in a disk whose block size is B, we can build a multiary DEST that supports access and search operations in $O(\log_B n)$ I/Os.*

Theorem 3.3 is particularly interesting in the context of space-efficient data structures. Our proposal behaves well when used on disk (i.e. it matches the lower bound in the I/O-model) without any special consideration. This is not the case most of the time, as it has been discussed in the past (Ferragina, 2010).

3.4. General Trees

Note that we have not mentioned much about the operation $\text{fetch}_T(v, r)$. This is because in the two previous representations it is quite straightforward[4], since we can map a node to an array storing the encoded values. Furthermore, the heap-embedding representation allows us to divide the array into levels and still be able to determine which level and position to access. This is not the case for general trees.

Most succinct tree representations support $\text{subtree\_size}_T(v)$ and can map nodes to an identifier that would allow to embed the values in a single array (Farzan, 2009; Sadakane & Navarro, 2010). The only issue is that the mapping to positions in the array of encoded values is not arbitrary but rather constrained by the tree representation. For example, using the Fully-Functional representation (Sadakane & Navarro, 2010), we can obtain the pre- or post- order of a node, and this requires $2n + o(n)$ bits beyond the space required to represent the encoded array of integers. The operations in the tree take constant time.

If we want to represent each level in a separate array the situation is more complicated. This is because we can no longer support $\text{subtree\_size}_T(v)$, or map a node to its rank in its level with a single tree representation (with the representations existing at this time). A very simple approach to fix this is to use two representations simultaneously: LOUDS (Jacobson, 1989), and Fully-Functional (Sadakane & Navarro, 2010). Fully-Functional would be used to answer, in constant time, all but computing the rank inside the level, which can be done in constant time using LOUDS. Both provide constant time navigation, so we can afford to navigate through both trees in a synchronized

[4]We comment more on this in the next section.

fashion. The disadvantage of this solution is that it requires $4n+o(n)$ bits plus the space required by the encoded values. Therefore, if encoding the arrays separately by level saves more than $2n$ bits, then this approach outperforms the representation of the previous paragraph.

3.5. Encoding DEST

Throughout this section we assume that all the values in the tree are stored in a contiguous region of memory using an encoding that supports random access in constant time. Recall that the embedding defines a mapping from the nodes of the tree to their position in this region of memory. For example, the mapping for the two heap-based embeddings presented is a level-wise traversal of the tree (considering the nodes at each level in order). Note that different encodings can be used to encode different chunks (e.g. different encodings for each level).

Our first proposal, named DEST-DAC, uses the Directly Addressable Codes (DACs) proposed by Brisaboa et al. (2013). Although these codes do not guarantee random access in constant time, they are very efficient and the worst case for random access is $\lceil \frac{\log_2 M}{b+1} \rceil$, where M is the largest value of the sequence and b is a parameter that defines the block size. Note that this can be considered constant time for practical purposes. DACs are variable length codes, thus compression is achieved provided small values are more frequent. When the differences are larger, a reasonable alternative is to encode the values using fixed length codes. In our second variant, named DEST- LVL, we encode the values at each level of the tree using fixed length codes. This encoding achieves constant time access in the word-RAM model.

Differences stored in lower levels of a DEST are expected to be smaller than those stored in higher levels. This is because nodes in lower levels map to positions in the original sequence close to the mapping of their parents. For example, in a binary DEST the mapping of a node in the last level is contiguous to the mapping of its parent. Therefore, a sensitive approach is to combine the two previous approaches by taking this into consideration. Our third alternative, named DEST-HYB, encodes the first levels of the tree using a fixed length encoding and the rest using DACs. The number of levels encoded with each variant provides a space-time trade-off. Finally, we propose an optimal space variant, named DEST-OPT, which, at each level, decides between either fixed length or DACs, and selects the one that uses less space. Note that this is optimal for the combination of both encodings, there may exist other encodings that achieve better space offering the same time complexities.

4. Applications and Experiments

In this section we present some applications of our structure and show its practical performance. All the experiments presented here were performed in an Intel Xeon E5520@2.27GHz, 72GB RAM, running Ubuntu server (kernel 2.6.31-19). We compiled with `gnu/g++` version 4.4.1 using `-O3` directive.

The implementations used in these experiments correspond with our own source code for all the variants of our structure based on the binary heap-like embedding (prefix DEST in the figures). These variants were described in Section 3.5. We also implemented two solutions based on differentially encoding the values stored in a linear list (plus sampling to support efficient access and search operations). We use Rice codes as an example of a commonly used approach that stores the offsets of the blocks (SAMP-RICE), and DACs as a variant that does not need to store them (SAMP-DAC). INTERPOLATIVE corresponds with the structure presented in Teuhola (2011), and we used the source code provided by the author. Finally, SA is the *sarray* by Okanohara & Sadakane (2007), which is available in LIBCDS[5].

4.1. Performance Overview

For our structure, we tested the four encodings proposed in Section 3.5, named DEST-DAC, DEST-LVL, DEST-HYB, and DEST-OPT. Our empirical evaluation shows that two of them clearly outperform the others: DEST-OPT is the most space-efficient variant and DEST-LVL is the fastest implementation. We only consider these two best performing implementations in all the experiments, except in Section 4.2 (Figure 6), where we include them all to give the reader an intuition regarding the difference between them.

As a first experiment, we repeat the measurements presented in Teuhola (2011) including these two implementations of our structure. In Figure 3 we show the compression efficiency of the structures in bits per source integer. We use two synthetic datasets with 1,000,000 integers each. In the first one, named Uniform, differences between consecutive values were uniformly generated in $[0, 2^q - 1]$, with $q = 1, \ldots, 10$. For the Exponential dataset, differences

[5]`http://libcds.recoded.cl`

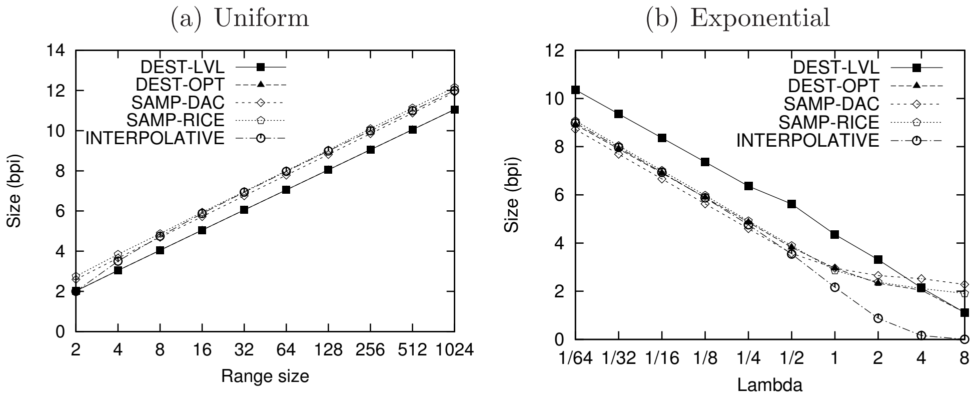

Figure 3: Compression efficiency in bits per integer. Datasets contain 1,000,000 integers with differences distributed (a) uniformly and (b) exponentially.

were generated according to the inverse of the cumulative distribution function $F(x, \lambda) = 1 - e^{-\lambda x}$ and λ was assigned values $1/64, 1/32, \ldots, 8$. Note that larger values of λ skew the distribution, thus increasing the probability of zeros.

For uniformly distributed integers both DEST-LVL and DEST-OPT, which perform similar, require less space than any of the other data structures. However, the compression efficiency of these two variants is different for exponentially distributed values. In this case, DEST-OPT is comparable with the state-of-the-art, but DEST-LVL requires about 1 bit more per integer. This is because DEST-LVL encodes all the values in the same level using the same number of bits and, for exponential distributions, it is likely to have at least one large value at each level and therefore the encoding can not take advantage of small values. As noted in Teuhola (2011), for $\lambda > 1$ the INTERPOLATIVE method requires less space than any other method due to its ability to represent long runs of zeros very compactly. Recall that in this paper we call INTERPOLATIVE to the data structure presented in Teuhola (2011), which is some times referred in that paper as *address calculation code* to differentiate it from the interpolative encoding (which does not support random access and efficient searches).

Next we focus on the space-time trade-off offered by the different data structures for both access and search operations. Figure 4 shows a summary of these experiments. In these experiments we also used datasets with 1,000,000 integers. For the Uniform dataset, differences were generated in the range $[0, 1023]$ and for the Exponential dataset, we used $\lambda = 1$. Similar

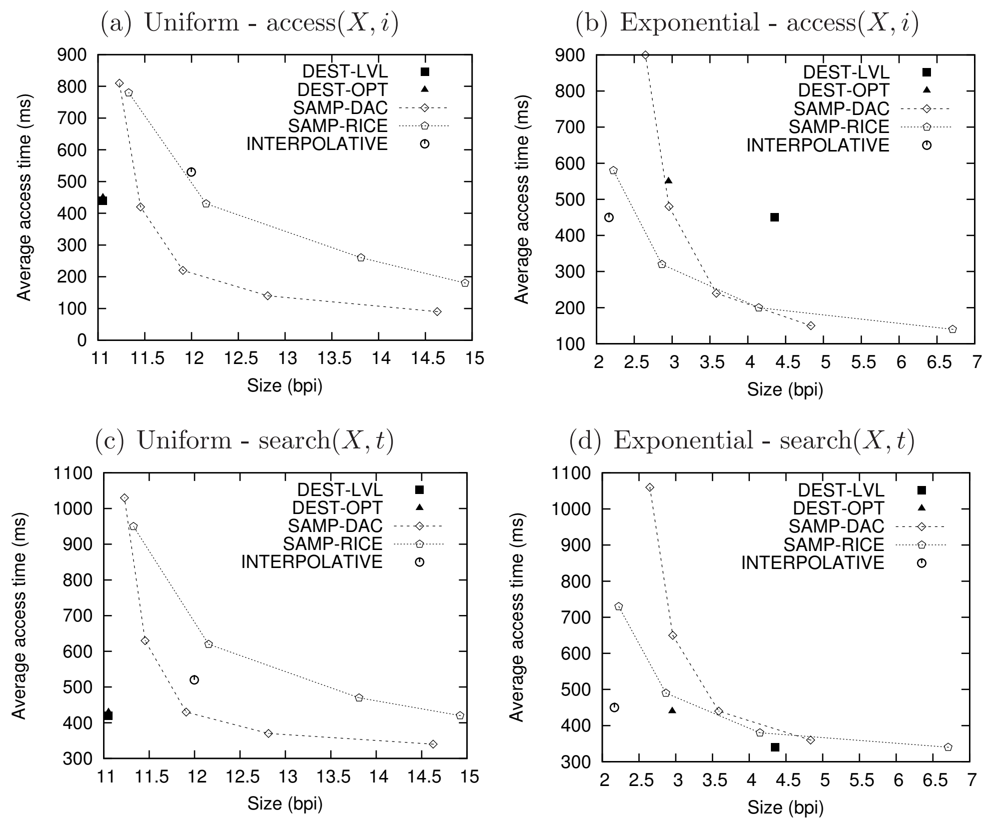

Figure 4: Space-time trade-off for the access(X, i) and search(X, t) operations in 1,000,000 integers with differences distributed uniformly in [0, 1023] and exponentially with $\lambda = 1$.

results were obtained with different configurations, but in the case of the Exponential dataset, recall from Figure 3 that this experiment corresponds to a configuration favorable to the INTERPOLATIVE structure. The main conclusion is that both non-parametric structures (ours and Teuhola's) provide interesting space-time trade-offs for both operations comparing with the sampling-based structures. We also notice that our structure stands out for uniform differences, but this flips around for exponentially distributed data.

Since we mainly focus on the search operation—we are compressing search trees—we emphasize that, for search, our structure clearly outperforms the state-of-the-art in the Uniform dataset and is competitive in the Exponential dataset (it requires less than 1 bit more than the INTERPOLATIVE structure and is slightly faster). In Figure 5 we show timing results for this operation for increasing n. We use the same configuration as in the pre-

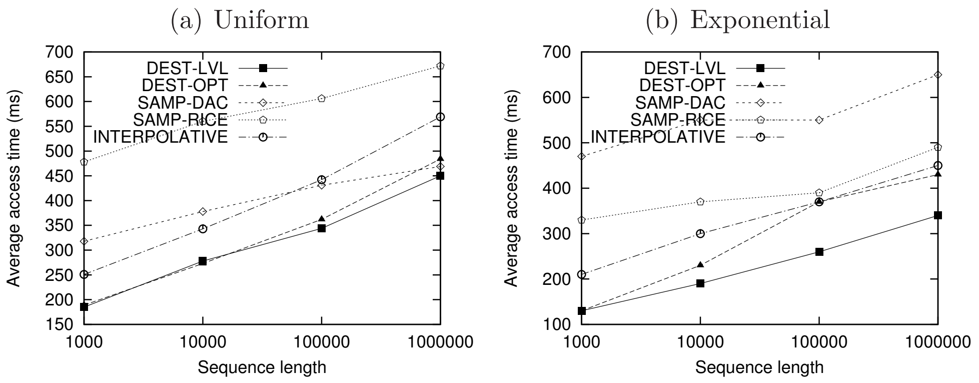

Figure 5: Scalability of the data structures for the search(X, t) operation. Differences were distributed uniformly in [0, 1023] and exponentially with $\lambda = 1$.

vious experiment. For the parametric structures (SAMP-DAC and SAMP-RICE) we chose sampling rates that provide a space requirement comparable with those of the non-parametric structures (around 12 bpi for the Uniform dataset, and 3 bpi for the Exponential dataset). This experiment shows the good scalability of our proposal.

4.2. Posting Lists

Posting lists are a main component of the ubiquitous inverted index (Baeza-Yates & Ribeiro-Neto, 1999; Witten et al., 1999). They store the occurrences (which can be documents, blocks, or offsets) of each *word* in the vocabulary. Therefore, finding the intersection of posting lists is a basic primitive operation as it can be used to perform multi-word queries. Although there exist many algorithms to perform this primitive operation, they can be broadly classified into those based on sequential merge, which perform better when the lists are about the same size, and those that search the longest list for each value of the shortest one (Hwang & Lin, 1972; Demaine et al., 2000; Barbay & Kenyon, 2002; Baeza-Yates, 2004), which perform better when the lengths of the lists are unbalanced. We call this variant Set-vs-Set (SVS).

Apart from the merge-wise alternative, the other algorithms are supported by any structure being *navigable* (i.e. support access to the *successor* and *predecessor* of a certain value) and *searchable* (i.e. support membership queries). These requirements are fulfilled when the values are stored in raw encoding, but these posting lists should be compressed, since space is a constraining factor for inverted indexes.

Let $P = p_1, p_2, \ldots, p_n$ be a posting list. A typical compressed representation of P constructs a sequence $P' = p_1, p_2 - p_1, \ldots, p_n - p_{n-1}$ of *d-gaps* and stores each value in P' using a variable-length encoding (Witten et al., 1999; Culpepper & Moffat, 2010). As we mention above, many algorithms for list intersection require a navigable and searchable data structure. These algorithms are not efficiently supported because the d-gaps have to be sequentially decompressed. This problem is usually overcome by sampling the original posting list (Culpepper & Moffat, 2010). In the rest of the paper, we refer to this scheme as a differential encoding of a linear list. Let the sampling rate s be a parameter of the data structure. The array $samples[1, \lceil n/s \rceil]$ stores a sample every s values such that $samples[i] = p_{is}$. This adds up to $\lceil n/s \rceil \lceil \log_2 p_n \rceil$ bits. In addition, the offsets corresponding with the beginning of each sampled block have to be stored in $\lceil n/s \rceil \lceil \log_2 N \rceil$ bits (N is the length in bits of the encoded sequence). As we noted before, the use of DACs (Brisaboa et al., 2013) does not require the offsets to be stored.

We implemented two variants of the SVS algorithm for intersection of posting lists. The naïve variant (*-N in the graphs) is a direct implementation of SVS that can be described with the operations presented in Section 3.1. This algorithm iterates over the m values of the shortest list (using amortized constant time per value) and searches the longest list for each of that values in $O(\log n)$ time. Thus, this algorithm performs list intersection in $O(m \log n)$ time. The second variant (*-T) exploits the fact that target values are searched for in increasing order, and the result is summarized in the following lemma.

Lemma 4.1. *Suppose we are given a sorted list L_1 of length m, and a second list L_2 of length $n > m$, both represented using DEST with the binary heap embedding. By using $O(\log n)$ extra words of space, we can compute the intersection between L_1 and L_2 in $O\big(m(1 + \log \frac{n}{m})\big)$. We call this procedure, supported by the DEST and $O(\log n)$ words of extra space,* batch searching.

Proof. Let T be the binary-heap-shaped tree representing L_2. We search for each value in L_1, in order, inside T. While performing the searches, we store the *trace* of the nodes, and their corresponding values, where we branched left during the previous search. There are at most $\lceil \log_2(n+1) \rceil$ such nodes and values. Subsequent searches start by finding in the trace the root of the subtree where the target value is stored. This is done by scanning the trace in the reverse order that nodes were visited.

For simplicity in our analysis, we will assume that L_2 is represented by a full heap. If it is not, then we can add dummy nodes to the heap, and make it full without affecting the asymptotic complexity. We also assume that all searches end at a leaf in the tree representing L_2. If this is not the case, and the value is found in an internal node u, then we can traverse to the left-most leaf of the subtree induced by u.

We use E to denote the set of all leaf nodes reached while searching for the values in L_1. The size of E is bounded by m. It is easy to see that the algorithm described gives a traversal of the minimal subtree of T that contains all nodes in E, and those are the only leaves accessed during the traversal.

We use $LCA(e_1, e_2)$ to denote the lowest common ancestor of nodes e_1 and e_2 in T. Define the set $S = \{v \in T \mid v = LCA(e_1, e_2), \{e_1, e_2\} \subseteq E\}$. We will split the cost of the traversal done by the searching algorithm into two parts, t_1 and t_2: t_1 corresponds to the number of distinct edges in paths between values in S and the root, and t_2 corresponds to the number of distinct edges in paths between values in S and values in E that do not contain any other values in S. The total cost of the traversal is at most $2(t_1 + t_2)$.

Consider a value $s \in S$ whose parent, $\bar{s}$, is not in S. Let s' be the sibling of s. Let e_1 and e_2 be the right- and left-most nodes whose LCA corresponds to s, respectively. Without loss of generality, assume that s is the left child of $\bar{s}$; the other case follows by symmetry. Suppose we take e_2 and we move it to be a leaf of the subtree induced by s', noting that no value in E belongs to that tree, since $\bar{s}$ is not in S. Then t_1 decreases by *at most* 1, since we remove the edge connecting s and $\bar{s}$. Moving e_2 increases t_2 by *at least* 1, since we add at least the edge between $\bar{s}$ and s'.

This shows that the worst case distribution of values in E is when the parent of every node in S is also in S. In other words, S corresponds to a heap-shaped tree with $m/2$ leaves covering the top of the tree representing L_2, therefore, $t_1 = O(m)$. The value t_2 can be computed as well, since it corresponds to m disjoint paths starting at the lowest values in S towards the leaves. Each path has length $\log_2 n - \lceil \log_2 m \rceil = O(1 + \log \frac{n}{m})$. So $t_2 = O(m + m \log \frac{n}{m})$, and this concludes the proof. $\square$

In Figure 6 (a), we show an empirical comparison of all the variants of our structure. The set of posting lists come from a parsing of the collections FT91

(a) DEST implementations. (b) General comparison.

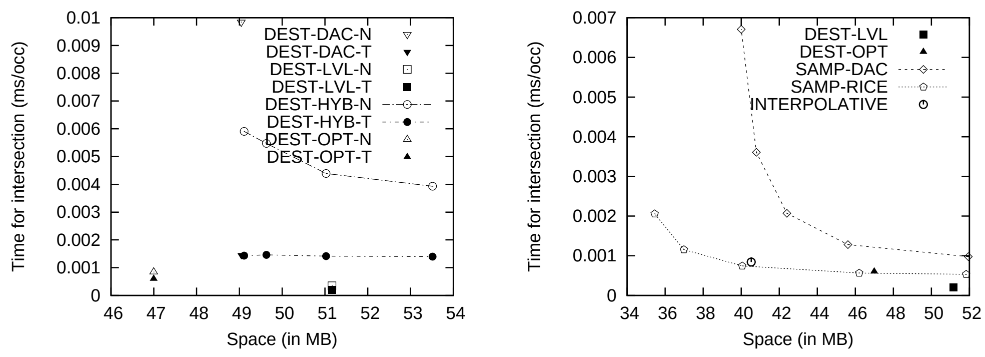

Figure 6: Posting lists. Space-time trade-off for pairwise intersection.

to FT94 from TREC-4[6]. This parsing generates 502,259 posting lists but we only consider those lists with a hundred values or more. As expected, two of our variants stand out from the others: DEST-OPT is the most space-efficient variant and DEST-LVL is the fastest implementation. The improvement achieved in the SVS algorithm by storing the trace is also remarkable. This is more evident in the DEST-DAC and DEST-HYB implementations where the cost of accessing a value in the tree is higher.

In Figure 6 (b), we include other structures existing in the literature. We corroborated the hypothesis of the lower compression achieved by the DACs. They require more space even though the offsets are not stored. Regarding our structure, it stands out in the time comparison in exchange for a higher space consumption.

4.3. Sparse Bitmaps

The well-known problem of *Rank*/*Select* dictionaries involves the representation of an ordered set $X \subset \{1, 2, \ldots, n\}$ supporting rank(X, v) (the number of values in X no greater than v) and select(X, i) (the position of the i-th smallest value in X) operations. When the values in the ordered set X come from the positions of the 1 bits in a sparse bitmap $B = [1, S]$, the most practical representation we are aware of is the *sarray* proposed by Okanohara & Sadakane (2007).

[6]http://trec.nist.gov

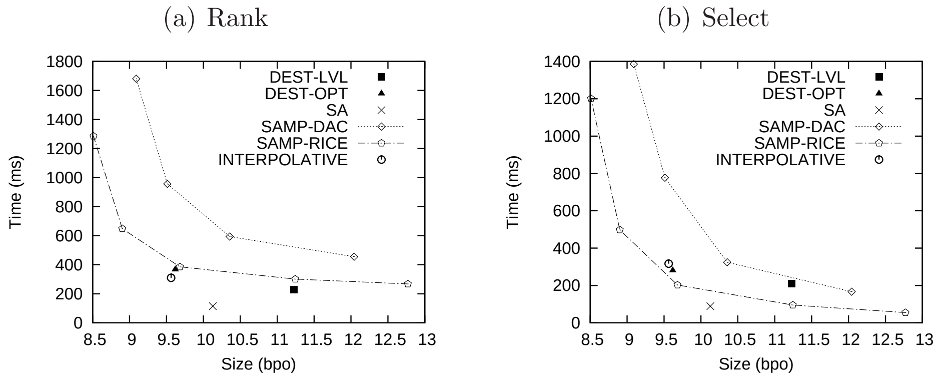

Figure 7: Time for 1,000,000 queries on a bitmap of length 10^8 with 1% of 1 bits.

We noticed that this problem can be reduced to the partial sums problem, so we formalize the reductions in the following lemma:

Lemma 4.2. *Let $Y = y_1, y_2, \ldots, y_n$ be a sequence representing the positions of the* 1 *bits in a sparse bitmap $B = [1, S]$. Any solution to the* searchable partial sums problem *over Y provides a solution to the* Rank/Select *problem on B using the following reductions: $rank(B, v) = search(Y, v) - 1$, and $select(B, i) = sum(Y, i)$.*

In Figure 7 we compare the space-time trade-off offered by several structures. We note that the *sarray* (SA) is the fastest structure both for *rank* and *select* operations. However, this experiment shows a new practical trade-off: the SAMP-RICE solution. In addition, both our structure and Teuhola's structure, which perform similarly, provide an interesting time performance when compared to SAMP-RICE, which is the only structure competing around that part of the trade-off.

4.4. Geographic Data

Due to the increasing demand of geographic services in the Web (e.g. geo-located business, geographic advertising, etc.), the indexing of geographic data has become very popular in the last decade. The most common (and simple) geographic data type is a two-dimensional point, which can be represented by a pair of coordinates. Recently, Arroyuelo et al. (2011) presented the first data structure to index points that supports a range query algorithm adaptive to a certain measure of difficulty of the instance.

In that paper, the authors present a practical data structure for orthogonal range queries over two-dimensional point datasets. Their structure is based on partitioning the input into a set of non-crossing monotonic chains. The structure is adaptive to the number of chains that are needed to represent the input, and their experimental evaluation shows that it is competitive with the state-of-the-art.

It is obvious that when this kind of structure is used for the Web, space is a constraining factor. Although the original proposal requires linear space, constants are not negligible for practical purposes. We show how to modify their approach by representing each chain using an instance of our structure, yielding a more space-efficient data structure.

It is interesting to notice that for this range searching data structure, the DEST for the x coordinates has the same shape as the one for the y coordinates. Furthermore, corresponding pairs of points are mapped to the same node in the tree. This allows us to embed both in the same tree/array. The navigation is quite similar to the one already presented.

We implemented this structure and compared it with a practical variant of the chain structure presented in Claude et al. (2010). Our implementation is based on the DEST-DAC described in Section 3.5. Table 1 summarizes the obtained results. We included two of the datasets used in the original paper: Italy and China, which are provided by the Georgia Tech TSP Web page[7]. Time results in Table 1 were obtained with the *tiny* query set (see Fig. 1 in Claude et al. (2010)) and are showed in seconds to perform 1,000,000 queries. Note that in their paper the real coordinates of the points are stored in the chains. We use a well-known technique by Gabow et al. (1984) to work with rank of the coordinates and not with the coordinates themselves. These real coordinates are stored in sorted arrays in order to translate the queries to the rank space. Results reported in Table 1 do not consider this space because is the same for both variants.

These results are promising because our structure considerably reduces the space while keeping competitive query times. This is especially interesting when we factor in that the original structure is already quite efficient in its space consumption when compared to the state-of-the-art (Claude et al., 2010). In addition, we observe two beneficial properties of our variant. First, the usage of an instance of our structure to represent each chain entails a

[7]http://www.tsp.gatech.edu

Table 1: Experimental comparison of the original chains data structure and our DEST-based variant.

	Italy		China	
	Space (KB)	Time (s.)	Space (KB)	Time (s.)
Claude et al. (2010)	135.30	12.12	562.45	13.78
DEST	110.08	70.06	353.17	185.17

space overhead. This overhead is negligible when the datasets are very large (the common scenario in real applications) but even in small datasets, such as *Italy*, the structure achieves good results. Second, a reduction in the number of chains also reduces the space of the structure (the aforementioned overhead is reduced and, in addition, the differences in the tree will be smaller). This is important because the query time of this structure is adaptive to the number of chains, and thus, an improvement in this number reduces both space and time.

5. Concluding Remarks

Space-efficient storage for non-decreasing sets of integers has previously been achieved by encoding differences between consecutive values in their linear list representation. This family of solutions requires sampling to provide efficient random access and searches. These solutions have been called parametric because of the sampling rate parameter, which provides a space-time trade-off. Non-parametric solutions have been recently introduced by Teuhola (2011).

We have presented an alternative non-parametric representation that provides space-efficient storage by exploiting the monotonicity of the sequence, while keeping a good time performance, especially for searching. Although similar in spirit, these two non-parametric solutions use different techniques that are of independent interest (see Konow et al. (2013) for a recent application of our techniques). We have shown some advantages of our proposal, such as independence on the encoding, a richer set of operations (e.g. batched queries), and a more elegant solution for representing the tree when the number of elements is not a power-of-2. In addition, our experimental evaluation has shown that our solution is competitive in a broad range of applications (e.g. representing posting lists and sparse bitmaps).

Throughout the paper, we have presented different variants of our structure that offer benefits in specific scenarios, both in theory and practice. For example, the multiary embedding is suitable for secondary memory and the variant based on general trees can adapt the shape of the tree to the distribution of the data, thus improving the compression ratio. This raises an interesting question that we leave as an open problem: given an encoding, what is the optimal shape for the tree in the static case? Another interesting line of future work is whether a dynamic version of our structure can be developed. This is especially important in some applications of our structure, for example, to integrate our solution for geographic data in spatial databases. Finally, our experimental evaluation has been presented as a proof of concept and more effort could be spent on engineering the implementation of differentially encoded search trees. For example, it would be interesting to develop a space-efficient spatial index and perform extensive experimentation to test its performance.

6. Acknowledgements

We would like to thank Jukka Teuhola for providing the source code for interpolative coding and help on how to use it. We would also like to thank J. Ian Munro and Gonzalo Navarro for useful comments regarding previous versions of this paper.

References

Anh, V. N., & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval*, *8*, 151–166.

Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Munro, J. I., Nicholson, P. K., Salinger, A., & Skala, M. (2011). Untangled monotonic chains and adaptive range search. *Theor. Comput. Sci.*, *412*, 4200–4211.

Baeza-Yates, R. A. (2004). A Fast Set Intersection Algorithm for Sorted Sequences. In *Proc. 15th CPM* (pp. 400–408).

Baeza-Yates, R. A., & Ribeiro-Neto, B. A. (1999). *Modern Information Retrieval*. Addison Wesley.

Barbay, J., & Kenyon, C. (2002). Adaptive intersection and t-threshold problems. In *Proc. 13th SODA* (pp. 390–399).

Brisaboa, N. R., Ladra, S., & Navarro, G. (2013). Dacs: Bringing direct access to variable-length codes. *Inf. Process. Manage.*, *49*, 392–404.

Claude, F., Munro, J. I., & Nicholson, P. K. (2010). Range queries over untangled chains. In *Proc. 17th SPIRE* (pp. 82–93).

Culpepper, J. S., & Moffat, A. (2010). Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, *29*, 1.

Demaine, E. D., López-Ortiz, A., & Munro, J. I. (2000). Adaptive set intersections, unions, and differences. In *Proc. 11th SODA* (pp. 743–752).

Farzan, A. (2009). *Succinct Representation of Trees and Graphs*. Ph.D. thesis University of Waterloo.

Ferragina, P. (2010). Data structures: Time, i/os, entropy, joules! In *Proc. 18th ESA* (pp. 1–16).

Fich, F., Munro, J. I., & Poblete, P. (1995). Permuting in place. *SIAM J. C.*, *24*, 266.

Gabow, H. N., Bentley, J. L., & Tarjan, R. E. (1984). Scaling and related techniques for geometry problems. In *Proc. 16th STOC* (pp. 135–143). ACM.

Hwang, F. K., & Lin, S. (1972). A Simple Algorithm for Merging Two Disjoint Linearly-Ordered Sets. *SIAM Journal on Computing*, *1*, 31–39.

Jacobson, G. (1989). Space-efficient static trees and graphs. In *Proc. 30th SFCS* (pp. 549–554).

Konow, R., Navarro, G., Clarke, C., & López-Ortíz, A. (2013). Faster and smaller inverted indices with treaps. In *Proc. 36th SIGIR* (pp. 193–202). ACM Press.

Moffat, A., & Stuiver, L. (1996). Exploiting clustering in inverted file compression. In *Proc. 6th DCC* (pp. 82–91).

Moffat, A., & Stuiver, L. (2000). Binary interpolative coding for effective index compression. *Information Retrieval*, *3*, 25–47.

Munro, J. I. (1996). Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science* (pp. 37–42). Springer.

Okanohara, D., & Sadakane, K. (2007). Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. 9th ALENEX*.

Sadakane, K., & Navarro, G. (2010). Fully-functional succinct trees. In *Proc. 21st SODA* (pp. 134–149).

Teuhola, J. (2011). Interpolative coding of integer sequences supporting log-time random access. *Information Processing & Management*, *47*, 742–761.

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing.

Yan, H., Ding, S., & Suel, T. (2009). Inverted index compression and query processing with optimized document ordering. In *Proc. 18th WWW* (pp. 401–410).