Differentially Encoded Search Trees[†]

Francisco Claude[*], Patrick K. Nicholson[*], and Diego Seco[+]

[*]*Cheriton School of Computer Science*
University of Waterloo, Canada
{*fclaude,p3nichol*}*@cs.uwaterloo.ca*

[+]*Database Laboratory*
University of A Coruña, Spain
dseco@udc.es

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 encodes the differences between consecutive elements in the sequence, and includes additional information (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 comparison to show that it is also competitive with the existing techniques.

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 a challenge.

In this paper we propose a compressed representation for non-decreasing sets of integers, which efficiently supports random access and searches. Let $X = x_1, x_2, \ldots, x_n$ be a sequence of non-decreasing integer values, in Section 2 we present 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 left-most position where t would be inserted in X while preserving the order in the sequence[1].

This structure has applications, for example, in the representation of posting lists; one of the two main components of the ubiquitous inverted index [3, 14]. In addition, the partial sums problem can be thought of as a particular instance of this

[†]This work was supported in part by the Google U.S./Canada PhD Fellowship Program and the David R. Cheriton Scholarships Program (first author); an NSERC of Canada PGS-D Scholarship (second author); and Ministerio de Ciencia e Innovación [TIN2009-14560-C03-02, TIN2010-21246-C02-01, and CDTI CEN-20091048] and Xunta de Galicia [2010/17] (third author).

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

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: sum(Y, i) (retrieve the sum of all the values up to position i) and search(Y, t) (retrieve the smallest i such that 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, sum(Y, i) reduces to access(X, i) and search(Y, t) reduces to search(X, t). Partial sums can be applied, for example, to represent *Rank/Select* dictionaries. In Section 3, we analyze these applications and show the practical performance of our solution.

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 (γ-codes, δ-codes, Rice codes, etc.) [14]. In order to efficiently support random access and searches, these schemata are forced to 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 [9, 1, 15] and also on proposing storage schemata for those samplings [5]. However, in this paper we explore a completely different schema that avoids the use of sampling while maintaining good compression rate and efficiency solving queries.

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 is known as Directly Addressable Codes (DAC) [4]. It relies on the Vbyte coding [13], and performs a reorganization of these variable-length codes in order to allow efficient random access. This technique solves the random access, but not the search problem. In order to support searches, the same sampling schema mentioned above can be used. The main advantage of the use of DACs in these schemata 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.

Recently, Teuhola [12] presented a schema based on the encoding of a binary search tree of the sequence using a variation of the Interpolative coding [8, 9]. 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 the encoding of differences in 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 adding new interesting operations, for example, the batched searching 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 are suitable for secondary memory. In a future version of this paper, we will explore the advantages of this structure in comparison with other structures for secondary memory.

2 Differentially Encoded Search Tree (DEST)

The main idea behind our structure is to differentially encode the values inside a search tree. Every node stores the difference with its parent and, 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 limitations that would defy the purpose of our structure, which is representing data in little space. In order to cope with this restriction we define a *tree representation*, which defines how the tree is stored without pointers in a contiguous region of memory and some basic operations to navigate through it, and an *encoding*, which defines how to compress and decompress the values (differences) stored in this contiguous region of memory.

In this section, we explore two tree representations (binary and multiary) based on the folklore heap embedding in an array. The multiary variant can be extended for general trees using succinct representations for trees (we omit the details of this variant due to lack of space). We also explore different encodings; fixed length at each level in the tree, directly addressable codes, and combinations of both. We note that the selection of a tree representation and encoding is entirely application dependent.

2.1 Operations

In order to present a representation-independent solution, we define an abstract data type (ADT) tree T, and enumerate the basic operations it supports. We assume T has $\lceil n/k \rceil$ nodes, each of arity $k+1$. The operations 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.
- $\text{parent}_T(v)$: find the parent of node v in T.
- $\text{subtree\_size}_T(v)$: count the number of nodes in the sub-tree rooted at node v.
- $\text{node\_rank}_T(v)$: obtain the rank of node v in T.
- $\text{child\_rank}_T(v)$: compute the rank of node v among its siblings.

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 pointing at the current node, which stores the real value of the node (not its difference with the parent). We start at the root by storing $v = \text{root}_T$ and a vector $\mathbf{w} = (w_0, \ldots, w_{k-1})$ with the k values stored in the root. To move from v to its r-th child, we set $v = \text{child}_T(v, r)$, and update $\mathbf{w}$ setting $w_j = w_r - \text{fetch}_T(v, j)$ if $r < k$ or $w_j = w_r + \text{fetch}_T(v, j)$ otherwise. We can move to the parent of node v in a similar way. Note that we only have to

compute 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 element in the set (i.e., access(X, i)) in logarithmic time. This can be done by traversing the tree using subtree_size$_T(v)$ at each node to determine the direction to move. Assuming k constant, the whole traversal takes $O(\log n)$ time[2].

The last operation we consider is searching among the values of the original set X that is encoded in the tree T (i.e., search(X, t)). Note that this operation is essentially searching for a point where to insert the key t in the tree. 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 $O(\log n)$. We can also iterate over the values starting from a given finger, either forwards or backwards. This allows access to a range of contiguous elements in $O(\log n + \ell)$ time, where ℓ is the length of the range.

2.2 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 elements.

It is well know that such tree can be embedded in an array $A[1 \dots n]$, where position 1 represents the root. The left and right children of a node in position v are in positions $2v$ and $2v + 1$, respectively, and its parent is in position $\lfloor \frac{v}{2} \rfloor$. The operation child_rank$_T(v)$ is computed as $v \mod 2$. These operations are the building blocks that allow us to support navigation in the tree. Three issues that we need to address in order to give a complete description of the structure are how to compute subtree_size$_T(v)$ and node_rank$_T(v)$, 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 $2^{h-h(v)} - 1$ elements, 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 the subtree at any node in constant time.

Supporting node_rank$_T(v)$. Instead of supporting the node rank operation for a random 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 subtree_size$_T(v)$ operation.

[2]We only consider balanced trees. Otherwise, the complexities depend on the height of the tree.

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 $2^{h-1} - 1 < n < 2^h$. There are two cases:[3]

- Case $n < 3 \times 2^{h-2}$: 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 $n \geq 3 \times 2^{h-2}$: The root is in position 2^{h-1}.

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 space [10]. 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 implied by these cases [6, Theorem 4].

Lemma 2.1 *Given a sorted set of n elements, we can embed its elements into a heap-like binary search tree or binary DEST in linear time.*

2.3 Multiary Heap-Like Embedding

The case of a multiary heap-shaped embedding is very 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 elements. The root is at position 1 in the array.

To move from node v to its r-th child, we compute $v(k + 1) + rk$. 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 — 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 $\min(n, p_r) - p_\ell + 1$.

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 elements that form the last level, so that we end up with $j - 1$ complete trees of height $h - 1$ and at most one possibly incomplete tree of height $h - 1$ (see Figure 1).

The number of elements in a complete tree of height $h - 2$ is $(k + 1)^{h-2} - 1$. Let β be the number of elements that fill the last level of the trees with height

[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.

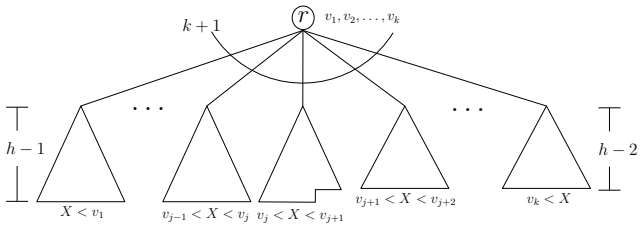


Figure 1: Illustration of the $(k+1)$-ary heap-like embedding.

$h-1$, 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 $\gamma = \beta \mod \left(k(k+1)^{h-2}\right)$ elements 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 elements every $(k+1)^{h-1}-1$ elements apart, then we jump $(k+1)^{h-2}-1+\gamma$ to retrieve the next element, and finally, for the remaining elements, we jump $(k+1)^{h-2}-1$ between each. This construction allows us to state the following lemma.

Lemma 2.2 *Given a sorted set of n elements, we can embed its elements into a heap-like $(k+1)$-ary search tree or 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 2.3 *Given n elements stored in a disk whose block size is B, we can build a DEST that supports access and search operations in $O(\log_B n)$ I/Os.*

2.4 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 embeddings presented is a pre-order traversal of the tree. Note that different encodings can be used to encode different chunks (e.g., different encoding for each level).

Our first proposal, DEST-DAC, uses Directly Addressable Codes (DACs) [4]. Although these codes do not guarantee constant time random access, they are very efficient and random access can be considered constant for practical purposes. They 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 approach achieves constant time access in the word-RAM model. The third alternative, 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, DEST-OPT, which, at each level, decides between either fixed length or DACs, and selects the one that uses less space.

3 Applications and Experiments

In this section we present some applications of our structure and show its practical performance. This section is not meant to be exhaustive but rather a proof of concept. All the experiments presented here were performed in an Intel Xeon E5520@2.27GHz, 74GB RAM, running Ubuntu server (kernel 2.6.31-19). We compiled with `gnu/g++` version 4.4.1 using `-O3` directive.

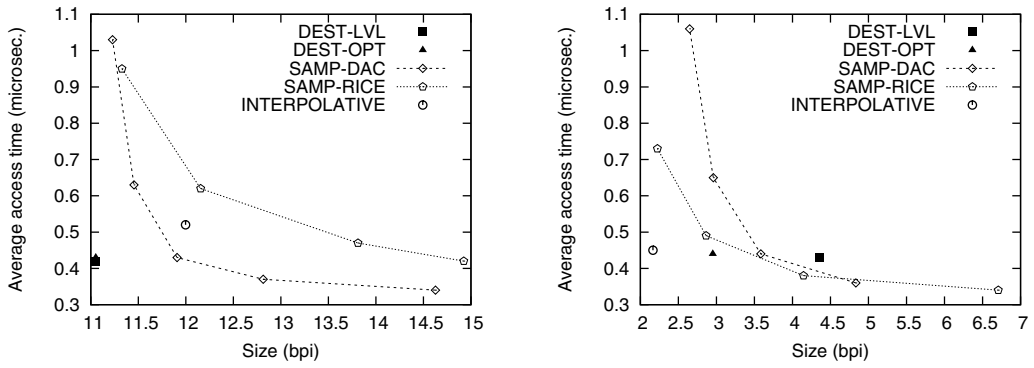
The implementations used in these experiments correspond with our own source code for all the variants of our structure based on the binary heap-like embedding (prefix DEST in the figures). These variants were described in Section 2.4. We also implemented two solutions based on variable length encoding and sampling. 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 [12] and we adapted the source code provided by the author. Finally, SA is the *sarray* by Okanohara and Sadakane [11], which is available in LIBCDS[4].

We first repeat the experiments presented by Teuhola in [12] including our structure. Figure 2 shows an example of these experiments. The main conclusion is that both non-sampling-based structures (ours and Teuhola's) provide interesting space-time trade-offs for both access and search operations comparing with the sampling-based structures. It is also interesting to notice that our structure stands out for uniform differences, but it is slightly worse for exponentially distributed data.

3.1 Posting Lists

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 [14, 5]. Many algorithms for list intersection [7, 2] 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 [5]. 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

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



(a) Uniform (b) Exponential

Figure 2: Space-time trade-off for the search(X, t) operation in 1,000,000 integers with differences distributed (a) uniformly in [0, 1023] and (b) exponentially with $\lambda = 1$.

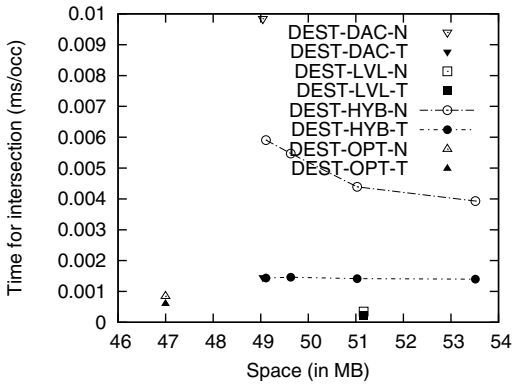
in $\lceil n/s \rceil \lceil \log_2 N \rceil$ bits (N is the length in bits of the encoded sequence). As we noted before, the use of DACs [4] does not require the offsets to be stored.

We implemented two variants of the Set-vs-Set (SVS) algorithm for intersection of posting lists. The naïve variant (*-N in the graphs) is a direct implementation of SVS that can be easily described with the operations presented in Section 2.1. This algorithm iterates over the m elements of the shortest list (using amortized constant time per element) and searches the longest list for each of that elements 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 elements are searched for in increasing order. We store the *trace* of all the nodes and their corresponding values accessed during the previous search (at most $\lceil \log_2(n+1) \rceil$). Subsequent searches start by finding in the trace the root of the subtree where the target element is stored (in $O(\log \log n)$ time) and then proceed in the same way. The complexity of this algorithm is $O\big(m(1 + \log \frac{n}{m})\big)$. This is an interesting trade-off offered by our structure, since it shows how to support *batch searching*.

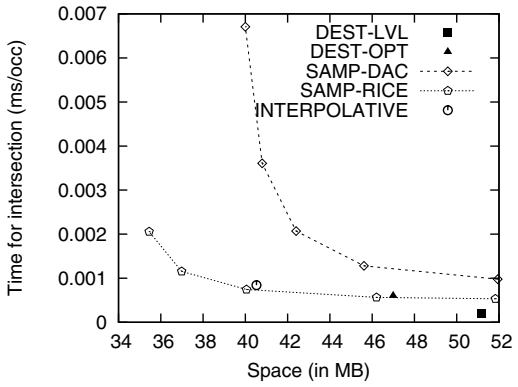
In Figure 3(a), we show an empirical comparison of all the variants of our structure. The set of posting lists come from a parsing of the collections FT91 to FT94 from TREC-4[5]. This parsing generates 502,259 posting lists but we only consider those lists with a hundred elements or more. As expected, 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 an element in the tree is higher.

In Figure 3(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.

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



(a) DEST implementations.



(b) General comparison.

Figure 3: Posting lists. Space-time trade-off for pairwise intersection.

3.2 Sparse Bitmaps

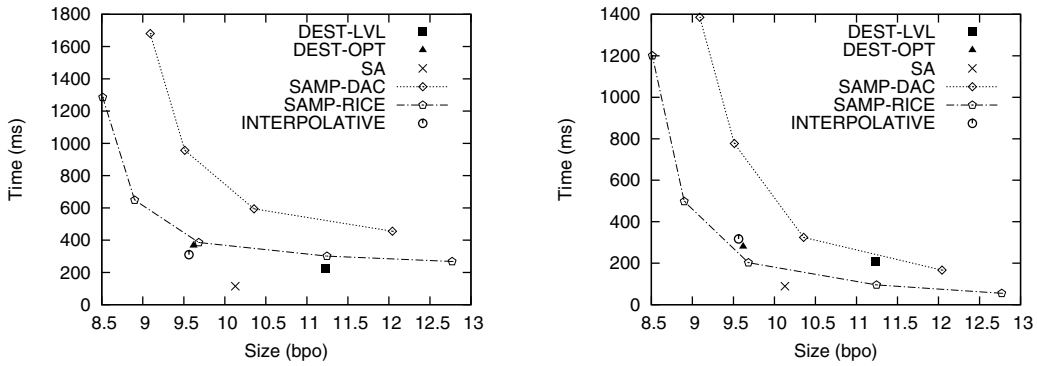
The well-known problem of *Rank/Select* dictionaries involves the representation of an ordered set $X \subset \{1, 2, \ldots, n\}$ supporting $\text{rank}(X, v)$ (the number of elements in X no greater than v) and $\text{select}(X, i)$ (the position of the i-th smallest element in X) operations. When 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 and Sadakane [11].

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: $\text{rank}(B, v) = \text{search}(Y, v) - 1$, and $\text{select}(B, i) = \text{sum}(Y, i)$.

In Figure 4 we compare the space-time trade-off offered by several structures. We note that the *sarray* 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 very similar, provide an interesting time performance when compared to SAMP-RICE, which is the only structure competing around that part of the trade-off.

4 Concluding Remarks

We have introduced a representation for non-decreasing sets of integers that provides space-efficient storage, by exploiting the monotonicity of the sequence, while keeping a good time performance in a broad range of operations; for example, searching. Throughout the paper, we have presented different variants of this structure that offer benefits in some specific scenarios. For example, the *multiary embedding* is suitable for secondary memory. Finally, we have noted that this structure has many applications, and is both practical and competitive.



(a) Rank (b) Select

Figure 4: Time for 1,000,000 queries on a bitmap of length 10^8 with 1% of 1 bits.

References

[1] Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Information Retrieval 8(1), 151–166 (2005)

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

[3] Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. Addison Wesley (1999)

[4] Brisaboa, N.R., Ladra, S., Navarro, G.: Directly Addressable Variable-Length Codes. In: Proc. 16th SPIRE. pp. 122–130 (2009)

[5] Culpepper, J.S., Moffat, A.: Efficient set intersection for inverted indexing. ACM Transactions on Information Systems 29(1), 1 (2010)

[6] Fich, F., Munro, J.I., Poblete, P.: Permuting in place. SIAM J. C. 24, 266 (1995)

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

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

[9] Moffat, A., Stuiver, L.: Binary interpolative coding for effective index compression. Information Retrieval 3(1), 25–47 (2000)

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

[11] Okanohara, D., Sadakane, K.: Practical Entropy-Compressed Rank/Select Dictionary. In: Proc. 9th ALENEX (2007)

[12] Teuhola, J.: Interpolative coding of integer sequences supporting log-time random access. Information Processing & Management 47(5), 742–761 (2011)

[13] Williams, H.E., Zobel, J.: Compressing Integers for Fast File Access. The Computer Journal 42(3), 193–201 (1999)

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

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