Searchable Compressed Representations of Very Sparse Bitmaps
(extended abstract)

Steven Pigeon [1]
pigeon@iro.umontreal.ca
McMaster University
Hamilton, Ontario

Xiaolin Wu [2]
xwu@poly.edu
Polytechnic University
Brooklyn, New York

Abstract

Very sparse bitmaps are used in a wide variety of applications, ranging from adjacency matrices in representation of large sparse graphs, representation of sparse space occupancy to book-keeping in databases. In this paper, we propose a method based on pruning of binary space partition (BSP) tree in minimal description length (MDL) principle for coding very sparse bitmaps. This new method for coding of sparse bitmaps meets seemingly competing objectives of good compression, the ability of conducting queries directly in compression domain, and simple and fast decoding.

1 Introduction

Bitmap is a convenient representation for many large data objects. The biggest advantage of bitmaps is that they supports $O(1)$ query operations. The examples are adjacency matrices of a large graph, the records of set membership, and space occupancy register. In these applications the bitmaps are typically very sparse (with the number of 0's much larger than the number of 1's) and extremely large in size. The challenge is how to compress a large sparse bitmap while still maintaining the advantage of rapid random access to individual bits of the compressed bitmap.

It is helpful to relate sparse bitmap to the problem of coding sparse occupancy of a large universe. Specifically, let $S = \{s_0, s_1, \cdots, s_{k-1}\}$ be a set of k numbers drawn from an interval I of natural numbers, which labels occupied locations within I. The interval I is very large, for example $I = [0, 2^n - 1]$ for some large integer $n \in \mathbb{N}$, whereas the cardinality of the set $|S| = k$ is very small compared to the size of interval. While a list of the k numbers may well be a more compact representation of S than a bitmap of 2^n entries, the former does not support $O(1)$ query on the set membership in S of any number in I as the latter does. The problem is to find a representation of S that is as compact as possible while still supporting fast and random query operations.

The traditional approaches to the problem is to use Golomb or arithmetic coding of run lengths between the members of the set (which is in fact differential coding) [8], or to use predictive coding or context-based coding [2, 3]. More recently, other approaches have been proposed for simple and

[1]Supported by Natural Sciences and Engineering Research Council of Canada.
[2]Supported by Natural Science Foundation Grant CCR-0208678.

fast decoding [7]. The problem of providing direct access in variable length record files also have been considered [9].

In this paper, we present a minimum description length (MDL) approach to devising the optimal representation of a very sparse occupancy set or equivalently very sparse bitmap, while allowing fast query on set membership.

2 Encoding the Set

We choose to code the set $S = \{s_0, s_1, \ldots, s_{k-1}\}$, $s_i \in I = [0, 2^n - 1]$, $n \in \mathbb{N}$, with a binary space partition tree (BSP) or regular binary tree. A BSP recursively subdivides I in equal parts until we reach intervals of size 1. This corresponds to a search tree over the entire range of I. The BSP prunes branches that do not have any element in S. An empty subtree is replaced by a "black" leaf that indicates that the corresponding subinterval is empty. Likewise, a complete subtree is replaced by a "white" leaf node whose corresponding interval is fully occupied by the set.

The set S is represented by the BSP whose leaves contain subsets of S. The tree itself is a full binary tree (a binary tree is full if each node is either a leaf or has exactly two children), without explicit interval specification. Knowing the path from the root to a node is sufficient to completely specify the interval location. In other words, at depth d (the root being at $d = 0$), one needs only the d bits from the path to decode the interval. Furthermore, $d-1$ of these d bits are shared among neighboring subtrees.

The compression methods for the sets contained in the leaves should be simple in order to achieve maximum decoding speed and support set operations in the compression domain. We chose only three different methods for the coding of the sets in the leaves. The first deals with leaves that represent intervals entirely empty of full. This is the pure node mode. The second method, bitmap, will allocate a bit for every number in the interval, without any compression. For every member of the set, the corresponding bit is set to 1 and all other bits are set to 0. The third method, compressed set, encodes sequentially the numbers in the set using a simple algorithm that is detailed later in Section 2.2. In the compressed set mode, sequential decompression is required to test for membership.

2.1 Encoding of the Tree Structure

The BSP tree is a full tree, and this allows us to encode it using only one bit per node. Let the reader consider fig. 1. Visiting the tree using preorder traversal, the encoder emits a 0 bits for an internal node and a 1 bit for a leaf, followed by this leaf's contents. Since the tree is full, it suffices to know if a node is an internal node or a leaf to recover the tree structure.

After the bit for a leaf is emitted, a code follows to determine what compression method is used to encode the contents. This code favors the compressed set mode. A 0 bit indicates that the set

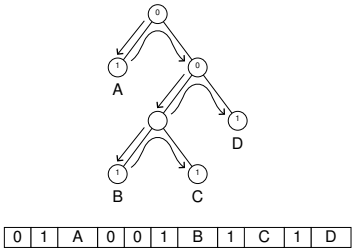

Figure 1: A preorder walk allows us to unambiguously encode the tree structure and the contents of its leaves. An internal node is represented by a 0 bit, a leaf is represented by a 1 bit. The actual contents of the leaves, here depicted as A, B, C and D, which is of variable size, is encoded using a self-delimited code.

is encoded using the compressed set method. The code 10 indicates a bitmap and 11 a pure node. A code for a pure node is followed by one more bit that tells if the interval is full or empty. The code for a bitmap is followed immediately by the raw bitmap data. The flag for a compressed set is followed immediately by the compressed set bit stream.

2.2 Encoding of Leaf Sets

The code for pure nodes hardly requires more explanations. The bitmap and the compressed set methods do. In this section, we describe both in details.

Let us begin with the bitmap. The size of the bitmap is implicitely specified by the leaf to which it is attached. Let this leaf i contain the set $S_i = \{s_0, s_1, \ldots, s_{l-1}\}$ and correspond to the interval $I_i = [a, b]$. The size of the bitmap, $(b - a + 1)$ bits, is a power of 2. The elements in S_i are coded relative to the beginning of the interval. The bits at offsets $s_j - a$, for $j = 0, 1, \ldots, l - 1$, are set to 1. All the other bits are set to zero. The membership information of a number $a \leqslant s_t \leqslant b$ is directly encoded by the bit at offset $n_t - a$.

In the compressed set method, the cardinality of the set is encoded first, followed by the enumeration of the members. The cardinality of the set is encoded using an Elias γ code [4, 1]. The members themselves are encoded using a progressively narrowing interval method. The Elias γ code for an interger l is bipartite. The first part is the unary code for $\lceil \lg l \rceil$, the length of the integer l. The unary code for an integer j is j ones followed by a zero. The second part of the code is the natural binary representation of l without its most significant bit. The resulting code length is $O(2 \lg l)$.

For the set $S_i = \{s_0, s_1, \ldots, s_{l-1}\}$ itself, we use the following method. The set is entirely contained within the interval $I_i = [a, b]$. The first member, s_0 can be any of $[a, b]$, so it is coded using $\lceil \lg(b - a + 1) \rceil$ bits. Once s_0 is encoded, we know that $s_1 \in [s_0 + 1, b]$ since $s_1 > s_0$. For s_1, $\lceil \lg(b - s0) \rceil$ bits are used. For s_2, the interval becomes $[s_1 + 1, b]$ so that $\lceil \lg(b - s_1) \rceil$ bits are needed. For s_j, $j > 1$, the interval is $[s_{j-1} + 1, b]$ and $\lceil \lg(b - s_{j-1}) \rceil$ bits are used.

The expected length in bits of a compressed set of cardinality l and with interval size of 2^m is approximatively $2\lceil \lg l \rceil + m(l - 2)$. If the members are uniformely distributed in the interval, then

roughly half of them are in the first half of the interval. These are coded using m bits. Roughly a quarter of them fall in the 3rd quarter of the interval. These are coded using $m-1$ bits. Then an eighth is found in the seventh eighth of the interval, requiring $m-2$ bits, and so on until all the interval is covered. This gives us an expected length of

$$l\sum_{i=1}^{m}\frac{1}{2^i}(m-i+1) = l(m-2+2^{-m}) \approx l(m-2)$$

bits for a set of cardinality l and interval size 2^m. To this, $2\lceil \lg l \rceil$ bits are added for the γ code of the cardinality.

Decoding using this method is very fast because, except for the γ code, the length of the each code is know before it is read. As there is only one γ code per leaf set, the cost of its decoding is negligible. On the encoding side, it is hardly a problem since computing $\lceil \lg i \rceil$ is an inexpensive operation for which some interesting methods have been developped [6]. Extractions and insertions of short bit strings of known length can be performed quite efficiently using machine-size integer operations rather than manipulations on individual bits.

3 Pruning Algorithms

Having decided on the methods for the encoding of the tree structure and the leaf subsets, we now investigate how to construct the optimal BSP tree that minimizes the description length of the set S. In this section, we present two algorithms to build minimal description length BSP trees.

3.1 The Forward Split Method

The forward split algorithm starts with a single node, the root, containing S, the set of k numbers to code with corresponding interval $I = [0, 2^n - 1]$. The algorithm will recursively split nodes if it is beneficial to do so in terms of compression. A parent node will spawn two children if the cost of coding the set at the parent, given the current interval size, is higher than the cost of coding the two sets corresponding to the children. If children are spawned, the algorithm is reapplied recursively on each child.

Let $L_I(S)$ be the set of values of S that are in the left portion of I, and $R_I(S)$, the set of values of S that are in the right portion. Let $I = [a, b]$. The left subinterval is given by $I_l = [a, a + \lfloor (b-a)/2 \rfloor]$, while the right is given by $I_r = [a + 1 + \lfloor (b-a)/2 \rfloor, b]$. Consequently, $L_I(S) = S \cap I_l$ and $R_I(S) = S \cap I_r$. Let $C(L)$ be the shortest coding length for the set L given the compression methods. That is,

$$C(L) = \min(C_p(L), C_b(L), C_c(L))$$

where $C_p(L)$ is 1 if L can be encoded as a pure node and ∞ otherwise, $C_b(L)$ is the cost of a bitmap, which is always equal to the current interval size, and where $C_c(L)$ is the compressed set

representation of L. The different code length includes all the information needed for unambiguous decoding.

A leaf i with list S_i and interval I_i, will spawn two children iff

$$C(S_i) > 2 + C\left(L_{I_i}(S_i)\right) + C\left(R_{I_i}(S_i)\right) \tag{1}$$

that is, the cost of coding the lists separately, plus the 2 bits needed to encode these new nodes as leaves, must be less that the code length of the parent node. The left child of node i receives $L_{I_i}(S_i)$ as set (with corresponding interval $(I_i)_l$, while the right child receives $R_{I_i}(S_i)$ and $(I_i)_r$. If children are created, the algorithm is reapplied recursively on each.

This algorithm has expected complexity $O(k \lg k)$ and worst case $O(nk)$. If early splits create roughly equal sized sublists, then at most at depth $(\lg k)$ all sets are reduced to one element. If, on the contrary, no splits occur in the early stages, then after depth $O(n - \lg k)$ splits must occur or the algorithm terminates. If the algorithm produces splits at this depth, then there are at most $O(\lg k)$ of them, until total depth of n is reached. At each level, at most $O(k)$ are examined in order to test for eq. (1), giving a total complexity of $O(2k \lg k) = O(k \lg k)$ in the average case and $O(2nk) = O(nk)$ in the worst case.

3.2 The Prune Back Method

The BSP tree of the members of S is first built. Again, S has k elements and the initial interval is $I = [0, 2^n - 1]$. Using postorder traversal, we test each non-leaf node for merging. If eq. (1) is not satisfied, the children nodes are merged into the parent node. The parent's set becomes the union of the children's sets and the children deleted. The shortest representation of the parent's set is chosen. The parent's interval remain unchanged. The merging can only occur between sibling leaves. If one child is not a leaf, merging is impossible.

Building the initial BSP requires $O(nk)$ operations. Set merging is computed in $O(1)$ using linked lists as internal representation. Testing eq. (1) is linear in the cardinality of the sets. Pruning starts from the leaves at $O(n)$ depth but normally stops at depth $O(\lg k)$. Neglecting the cost of building the BSP tree, the algorithm is $O((n - \lg k)k)$, since at any given depth, no more than k elements are examined. The worst case consist in the case were all nodes are merged to the root, for $O(nk)$.

4 Query in the Compressed Domain

Set membership queries should be conducted in compressed domain whenever possible. We have to be able to search the compressed representation of the data. This can be done in several ways, with different time and memory trade-offs.

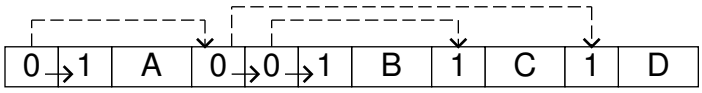

Figure 2: The tree in fig. 1 produces the above compressed data. The dashed arrows in the figure represent the right-child pointers. The short solid arrows represent the left-child pointers. The left child is always found immediately after its parent, whereas the right child is found at a location that depends on the size of the left subtree rooted at its sibling.

A first strategy is to scan the compressed file when it is opened to recreate in memory only the BSP. In this BSP, leaves contain not the actual data, but only its location within the compressed file. The preorder walk in the tree described in Section 2.1 let us retrieve the tree structure. When a leaf is reached, its contents may have to be decoded. In the case of a pure node or a bitmap node, the size is known and the bits can be safely skipped over. In the case of a compressed set node, one has to decode the complete leaf data in order to "skip" the leaf data.

The preorder scan of the tree and its encoding (see Section 2.1) ensures us that the bits for the left child are to be found immediately after the bit of the parent node. However, the bits of the right child are to be found after all the bits of the left subtree. Consider fig. 2. This location can not be foretold; one has to decode the left subtree first. One can scan the entire file at open time and cache the right-child pointers in a space-efficient yet rapidly searchable structure. A possible structure would be a simple table of tuples $\langle\, i, R(i) \,\rangle$, sorted on i, that give, for each node i the position of its right child, $R(i)$. The information of the left children can be read sequentially from the file while the position of the right children is obtained from the table. This allow direct access to leaves without keeping the entire BSP tree in memory.

Once the position of the leaf data in the compressed file is obtained, one still has to decompress the leaf data. If the leaf i is a pure node (a node corresponding to an interval of all zeros or all ones), the test of membership is $O(1)$. If the leaf i data is an uncompressed bitmap, then access to the information is direct. It suffice to know s_t, the element for which we are testing membership, and a_i, the lower bound of the interval corresponding to this leaf i. We test the bit at offset $s_t - a_i$ relative to the beginning of the bitmap. This is $O(1)$ as it does not depend on the interval size nor the rank of the number to test.

If the data is a compressed set, the search is sequential. First, the cardinality must be decoded, then each set member. The cardinality m is encoded using a γ code, and can be read in $O(\lg m)$ steps. The members are coded using a code of known length, so decoding can be $O(1)$. However, one has to potentially decode the entire set to test for membership. This procedure is expected $O(m)$. To bound decoding time, to at most m' steps, eq. (1) can be modified to

$$\big(C(S_i) + \lambda(|S_i| > m')\big) > c_0 + C\left(L_{I_i}(S_i)\right) + C\left(R_{I_i}(S_i)\right)$$

where $(a > b)$ is an indicator function that takes value 1 if the condition is satisfied and 0 otherwise. The parameter λ is some large value that incites the algorithm to split whenever the current set cardinality is larger than m'. However, experimental results, discussed in Section 5, show that the lists created by the algorithms remain short in average; in fact their expected length is $O(\lg k)$, thus bounding search time by $O(\lg k)$.

5 Results

In this section we present our experimental results using the proposed prune algorithm. Table 1 details the compression results obtained by several compression algorithms. Fig. 3 presents the same results graphically.

Various algorithms are compared. The Raw "algorithm" is the simple binary representation of the set using machine-size integers (32 bits). The RLL (*run length limited*) algorithm codes runs limited in length. For this algorithm, the data is considered as a full length bitmap, with a bit set to one if its position corresponds to a member in the set. The run of zeros between 1's are constrained to be at most $2^{16} - 1$. A run length of $2^{16} - 1$ is automatically followed by another run, possibly of length zero. A run less than $2^{16} - 1$ is followed by a byte with at least one bit set. RLU (*run length universal*) is essentially the same method except that the codes for the run lengths are a parametric universal codes [10]. The parameter is optimized for each file.

The tree algorithm is simply the unpruned BSP tree. The tree is encoded as described in Section 2.1. In addition to the bits needed to encode the tree itself, one extra bit per leaf is needed to encode its color. A leaf can represent an interval that is completely full or completely empty; with degenerate case where a leaf represents an interval of length 1. The final candidate algorithm Gzip is applied to the raw binary membership representation of the set. For Gzip we used maximum compression setting (option "-9").

Only the tree and pruned tree algorithms offer direct access to the data without modification. All other algorithms must decompress the data up to the point of interest. To be fair, the RLL, RLU, and Gzip algorithms may also be modified to have certain degree of random access to the compressed data. For instance, data can be partitioned in pages of a given size, pages that can be indexed. This index only needs to retain the lowest value contained in a page as well as the location of the page in the file. Small pages, say 4K bytes uncompressed, can be compressed separately and indexed. To access a particular item, the index is searched, then the corresponding page is decompressed and searched. But an algorithm like Gzip will have inferior compression performance on smaller pages since it exploits redudancy on a large horizon to compress. Table 1 shows that Gzip must have a rather large file before compressing at all.

MDL pruning produces files that are about 60% the size of those produced by Gzip, and 10% smaller than the next best algorithm, RLU. The files produced by our algorithm, contrary to Gzip, remain searchable in their compressed form. RLL eventually produce files that are comparatively

k	Raw	RLL	RLU	Tree	Pruned	`Gzip -9`
10	40	14943.5	48.5	75.1	37.3	68.6
100	400	16285.8	399.5	660.9	362.9	429.0
1000	4000	18394.0	3500.5	5790.0	3218.9	4029.0
10000	40000	38381.6	30277.0	49549.8	28039.7	39939.6
100000	400000	2999972.0	256161.0	412519.0	238910.0	386135.0

Table 1: Results for the various algorithms with an interval size of 2^{32}. The results are average over 100 trials for each k. The k column indicates the number of elements in the set. Raw is the size of k integers saved as 32 bits integers. RLL, for *run length limited* encodes runs of at most $2^{16} - 1$. RLU, *run length universal coding*, uses a parametric universal code for the run length, with a parameter computed from the data. Tree is the simple unpruned BSP, with leaf type information. Pruned presents our results with the MDL pruning algorithm. Lastly, `Gzip -9` is Gzip 1.2.4 (current version) with -9 option for "maximal compression".

small; in fact, it will perform best when the expected gap between two elements is about 2^{15} to $2^{16} - 1$. The unpruned tree algorithm is hopeless as it always produces a file that is larger than even the raw representation of the set.

The other important feature of our algorithms is the expected shallow depths of the search trees and relatively small sizes of the leaf subsets created. Experimental data show that if the k elements in S are drawn uniformely over the interval $I = [0, 2^n - 1]$, the average size of the sets is approximately $O(\lg k)$, and that the tree has therefore $O(k/\lg k)$ leaves. The expected length of the path from the root to the leaves is $O(\lg k - \lg\lg k) = O(\lg k)$. Searching the leaf subsets is either $O(1)$ if they are represented by a pure node or a bitmap or $O(\lg k)$ if they are encoded as compressed sets. The total expected search time is therefore $O(\lg k)$.

6 Conclusion

In this paper we present an MDL-based pruned tree algorithms for compression of very sparse bitmaps. We show that, not only it produces a highly compressed representation of sparse bitmaps, it also supports random access to individual bits directly in the compressed domain, with very little auxiliary memory. We have also shown that the expected access time with the compressed representation is $O(\lg k)$, where k is the number of 1's in the bitmap. The two proposed algorithms have a worst case time complexity of $O(klogN)$, where N is the size of the bitmap.

References

[1] T. C. Bell, J. G. Cleary, I. H. Witten, *Text compression*, Prentice-Hall 1990

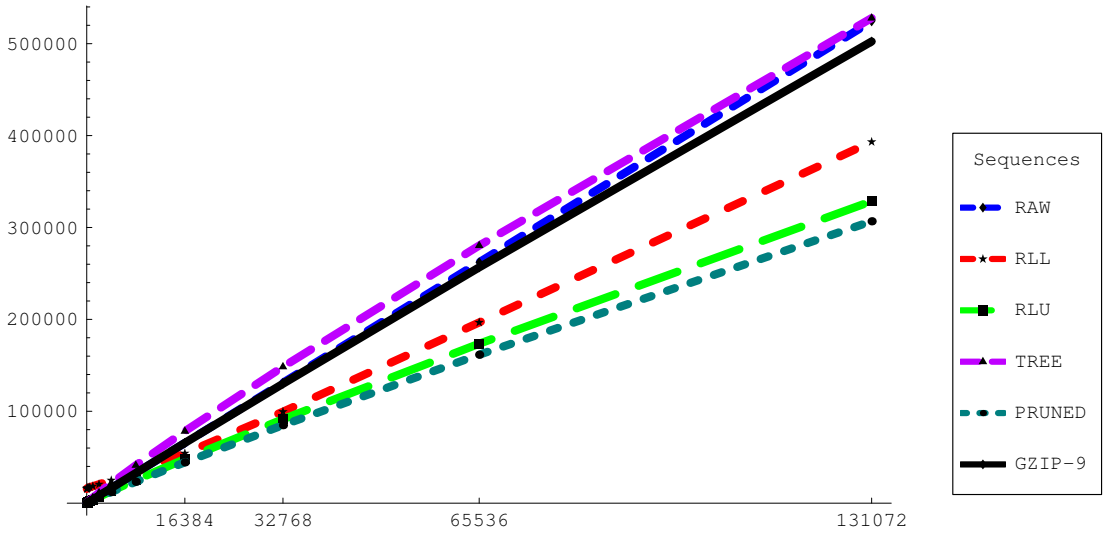

Figure 3: The results compared graphically. The y axis is the size of the file in bytes, while the x axis is the number of items in the set. (For more on the compression algorithms, see caption of Table 1.) We see that all the algorithm have nice predictable behavior.

[2] A. Bookstein, S. T. Klein, "Models for Bitmap Generation: A systematic Approach to bitmap Compression", Inf. Proc. & Management, v28, p. 735–748, 1992

[3] A. Bookstein, S. T. Klein, T. Raita, "Simple Bayesian Models for Bitmap Compression", J of Inf. Retrieval, v4#1, p. 315–328, 2000

[4] P. Elias, "Universal codeword sets and representations of the integers", *IEEE Trans. Inf. Theory*, IT-21(2), p. 193-203, 1975

[5] J. Goldstein, R. Ramakrishnan, U. Shaft, "Compressing Relations and Indexes", *Proc. IEEE Conf. on Data Engineering*, p. 370–379, 1998.

[6] S. W. Warren, jr, *Hacker's Delight*, Addison-Wesley, 2002

[7] Alistair Moffat, Lang Stuiver, "Binary Interpolative Coding for Effective Index Compression ", Information Retrieval, v3#1 p. 25–47, July 2000

[8] Alistair Moffat, Justin Zobel, "Parameterised Compression for Sparce Bitmaps", Tech Rep 92/1, Department of Computer Science, University of Melbourne

[9] Alistair Moffat, Justin Zobel, "Supporting Random Access in files of Variable Length Records", Inf. Process. Letters, v46#2, p 71–77, 1993

[10] Steven Pigeon, *Contributions à la compression de données*, Ph. D. Thesis, Dept of Computer Science, University of Montreal, 2002