Practical Dynamic Entropy-Compressed Bitvectors with Applications [⋆]

Joshimar Cordova and Gonzalo Navarro

CeBiB — Center of Biotechnology and Bioengineering,
Department of Computer Science, University of Chile,
{jcordova,gnavarro}@dcc.uchile.cl

Abstract. Compressed data structures provide the same functionality as their classical counterparts while using entropy-bounded space. While they have succeeded in a wide range of static structures, which do not undergo updates, they are less mature in the dynamic case, where the theory-versus-practice gap is wider. We implement compressed dynamic bitvectors B using $|B|H_0(B)+o(|B|)$ or $|B|H_0(B)(1+o(1))$ bits of space, where H_0 is the zero-order empirical entropy, and supporting queries and updates in $\mathcal{O}(w)$ time on a w-bit word machine. This is the first implementation that provably achieves compressed space and is also practical, operating within microseconds. Bitvectors are the basis of most compressed data structures; we explore applications to sequences and graphs.

1 Introduction

Compact data structures have emerged as an attractive solution to reduce the significant memory footprint of classical data structures, which becomes a more relevant problem as the amount of available data grows. Such structures aim at representing the data within almost its entropy space while supporting a rich set of operations on it. Since their beginnings [12], several compact structures have been proposed to address a wide spectrum of applications, with important success stories like ordinal trees with full navigation in less than 2.5 bits [1], range minimum queries in 2.1 bits per element [7], and full-text indexes using almost the space of the compressed text [15], among others. Most of the major practical solutions are implemented in the *Succinct Data Structures Library* [10], which offers solid C++ implementations and extensive test datasets.

Most of these implemented structures, however, are static, that is, they do not support updates to the data once they are built. While dynamic variants exist for many compact data structures, they are mostly theoretical and their practicality is yet to be established.

At the core of many compact structures lay simple bitvectors supporting two important queries: counting the number of bits b up to a given position (*rank*) and finding the position of the i-th occurrence of bit b (*select*). Such bitvectors enable well-known compact structures like sequences, two-dimensional

[⋆] Funded by Basal Funds FB0001 and with Fondecyt Grant 1-140796, Conicyt, Chile.

grids, graphs, trees, etc. Supporting insertion and deletion of bits in the bitvectors translates into supporting insertion and deletions of symbols, points, edges, and nodes, in those structures. Very recent work [16] shows that dynamic bitvectors are practical and that compression can be achieved for skewed frequencies of 0s and 1s, provided that the underlying dynamic memory allocation is handled carefully. Furthermore, the authors implement the *compressed RAM* [13] and show that it is practical by storing in it a directed graph.

In this paper we build on a theoretical proposal [17] to present the first practical dynamic bitvector representations whose size is provably entropy-bounded. A first variant represents $B[1,n]$ in $nH_0(B)+o(n)$ bits, where H_0 denotes the zero-order empirical entropy. For bitvectors with few 1s, a second variant that uses $nH_0(B)(1+o(1))$ bits is preferable. Both representations carry out updates and *rank/select* queries in time $\mathcal{O}(w)$ on a w-bit machine. In practice, the times are just a few microseconds and the compression obtained is considerable. Instead of using our structure to implement a compressed RAM, we use our bitvectors to implement *a*) a practical dynamic wavelet matrix [5] to handle sequences of symbols and two-dimensional grids, and *b*) a compact dynamic graph that achieves considerable space savings with competitive edge-insertion times.

Along the way we also describe how we handle the dynamic memory allocation with the aim of reducing RAM fragmentation, and unveil a few related practical results that had not been mentioned in the literature.

2 Basic Concepts

Given a sequence $S[1,n]$ over the alphabet $[1,\sigma]$, $access(S,i)$ returns the character $S[i]$, $rank_c(S,i)$ returns the number of occurrences of character c in $S[1,i]$ and $select_c(S,j)$ returns the position of the j-th occurrence of c. The (empirical) zero-order entropy of S is defined as $H_0(S)=\sum_{1\leq c\leq\sigma}\frac{n_c}{n}\lg\frac{n}{n_c}$, where c occurs n_c times in S, and is a lower bound on the average code length for any compressor that assigns fixed (variable-length) codes to symbols. When $\sigma=2$ we refer to the sequence as a bitvector $B[1,n]$ and the entropy becomes $H_0(B)=\frac{m}{n}\lg\frac{n}{m}+\frac{n-m}{n}\lg\frac{n}{n-m}$, where $m=n_1$. The entropy decreases when m is closer to 0 or n. In the first case, another useful formula is $H_0(B)=\frac{m}{n}(\lg\frac{n}{m}+\mathcal{O}(1))$.

Dynamism is supported by the operations $insert(S,i,c)$, which inserts the character c before position i in S and moves characters $S[i,n]$ one position to the right; $delete(S,i)$, which removes character $S[i]$ and moves the characters $S[i+1,n]$ one position to the left; and $modify(S,i,c)$, which sets $S[i]=c$.

Uncompressed (or plain) bitvector representations use $n+o(n)$ bits, and can answer queries in $\mathcal{O}(1)$ time [3]. Compressed representations reduce the space to $nH_0(B)+o(n)$ bits while retaining the constant query times [24]. Dynamic bitvectors cannot be that fast, however: queries require $\Omega(\lg n/\lg\lg n)$ time if the updates are to be handled in $\mathcal{O}(\text{polylog}\, n)$ time [8]. Dynamic plain bitvectors with optimal times $\mathcal{O}(\lg n/\lg\lg n)$ for all the operations exist [23]. Mäkinen and Navarro [17] presented the first dynamic bitvectors using compressed space, $nH_0(B)+o(n)$ bits, and $\mathcal{O}(\lg n)$ times. It is possible to improve the times to the optimal $\mathcal{O}(\lg n/\lg\lg n)$ within compressed space [21], but the solutions are complicated and unlikely to be practical.

A crucial aspect of the dynamic bitvectors is memory management. When insertions/deletions occur in the bit sequence, the underlying memory area needs to grow/shrink appropriately. The classical solution, used in most of the theoretical results, is the allocator presented by Munro [18]. Extensive experiments [16] showed that this allocator can have a drastic impact on the actual memory footprint of the structure: the standard allocator provided by the operating system may waste up to 25% of the memory due to fragmentation.

The first implementation of compact dynamic structures we know of is that of Gerlang [9]. He presents dynamic bitvectors and wavelet trees [11], and uses them to build a compact dynamic full-text index. However, memory management is not considered and bitvectors $B[1,n]$ use $\mathcal{O}(n)$ bits of space, $3.5n$–$14n$ in practice. A more recent implementation [25] has the same problems and thus is equally unattractive. Brisaboa et al. [2] also explore plain dynamic bitvectors; they use a B-tree-like structure where leaves store blocks of bits. While their query/update times are competitive, the space reported should be read carefully as they do not consider memory fragmentation. In the context of compact dynamic ordinal trees, Joannou and Raman [14] present a practical study of dynamic Range Min-Max trees [21]. Although the space usage is not clear, the times are competitive and almost as fast as the static implementations [1].

There also exist open-source libraries providing compact dynamic structures. The *ds-vector* library [22] provides dynamic bitvectors and wavelet trees, but their space overhead is large and their wavelet tree is tailored to byte sequences; memory fragmentation is again disregarded. The compressed data structures framework *Memoria* [26] offers dynamic compact bitvectors and ordinal trees, among other structures. A custom memory allocator is provided to reduce fragmentation, but unfortunately the library is not in a stable state yet (as confirmed by the author of the library).

Klitzke and Nicholson [16] revisit dynamic bitvectors. They present the first practical implementation of the memory allocation strategy of Munro [18] tailored to using compact data structures, and show that it considerably reduces memory fragmentation without incurring in performance penalties. They present plain dynamic bitvectors $B[1,n]$ using only $1.03n$ bits. For bitvectors with $m \ll n$ 1s, they build on general-purpose compressors `lz4` and `lz4hc` to reduce the space up to $0.06n$. However, they lack theoretical guarantees on the compression achieved. While their work is the best practical result in the literature, the code and further technical details are unfortunately unavailable due to legal issues (as confirmed by the first author).

3 Dynamic Entropy-Compressed Bitvectors

In this section we present engineered dynamic bitvectors that achieve zero-order entropy compression. These are based on the ideas of Mäkinen and Navarro [17], but are modified to be more practical. The following general scheme underlies almost all practical results to date and is used in this work as well. The bitvector $B[1, n]$ is partitioned into *chunks* of contiguous bits and a balanced search tree (we use AVLs) is built where the leaves store these chunks. The actual partition strategy and storage used in the leaves vary depending on the desired compression. Each internal node `v` of the balanced tree stores two fields: `v.ones` (`v.length`) is the number of 1s (total number of bits) present in the left subtree of `v`. The field `v.length` is used to find a target position i in B: if $i \leq$ `v.length` we descend to the left child, otherwise we descend to the right child and i becomes $i -$ `v.length`. This is used to answer *access*/*rank* queries and also to find the target leaf where an update will take place (for *rank* we add up the `v.ones` field whenever we go right). The field `v.ones` is used to answer $select_1(B, j)$ queries: if $j \leq$ `v.ones` the answer is in the left subtree; otherwise we move to the right child, add `v.length` to the answer, and j becomes $j -$ `v.ones`. For $select_0(B, j)$ we proceed analogously, replacing `v.ones` by `v.length` $-$ `v.ones`. The leaves are sequentially scanned, taking advantage of locality. Section 3.2 assumes the tree is traversed according to these rules.

3.1 Memory management

Although Klitzke and Nicholson [16] present and study a practical implementation of Munro's allocator [18], the technical details are briefly mentioned and the implementation is not available. We then provide an alternative implementation with its details. In Section 5, both implementations are shown to be comparable.

Munro's allocator is tailored to handle small *blocks* of bits, in particular blocks whose size lies in the range $[L, 2L]$ for some $L = \text{polylog}\, n$. It keeps $L+1$ linked lists, one for each possible size, with $L+1$ pointers to the heads of the lists. Each list l_i consists of fixed-length *cells* of $2L$ bits where the blocks of i bits are stored contiguously. In order to allocate a block of i bits we check if there is enough space in the head cell of l_i, otherwise a new cell of $2L$ bits is allocated and becomes the head cell. To deallocate a block we fill its space with the last block stored in the head cell of list l_i; if the head cell no longer stores any block it is deallocated and returned to the OS. Finally, given that we move blocks to fill the gaps left by deallocation, *back pointers* need to be stored from each block to the external structure that *points* to the block, to update the pointers appropriately. Note that in the original proposal a block may span up to two cells and a cell may contain pieces of up to three different blocks.

Implementation. Blocks are fully stored in a single cell to improve locality. As in the previous work [16], we only allocate blocks of *bytes*: L is chosen as a multiple of 8 and we only handle blocks of size $L, L+8, L+16, \ldots, 2L$, rounding the requested sizes to the next multiple of 8. The cells occupy $T = 2L/8$ bytes and are allocated using the default allocator provided by the system. Doing increments of 8 bits has two benefits: the total number of allocations is reduced and the memory pointers returned by our allocator are byte-aligned. The head pointers and lists l_i are implemented verbatim. The *back pointers* are implemented using a folklore idea: when allocating a block of l bytes we instead allocate $l + w/8$

bytes and store in the first w bits the address of the *pointer* to the block, so that when moving blocks to fill gaps the pointer can be modified. This creates a strong binding between the external structure and the block, which can be pointed only from one place. This restriction can be alleviated by storing the pointer in our structure, in an immutable memory area, and let the external structures point to the pointer. This requires that the external structures know that the handle they have for the block is not a pointer to the data but a pointer to the pointer. In this sense, the memory allocator is not completely transparent.

As a further optimization, given that our dynamic bitvectors are based on search trees, we will be constantly (de)allocating very small structures representing the nodes of the trees (eg. 4 words for a AVL node). We use another folklore strategy for these structures: given that modern operating systems usually provide 8MB of *stack* memory for a running process, we implement an allocator on top of that memory, avoiding the use of the heap area for these tiny structures; (de)allocation simply moves the end of the stack.

3.2 Entropy-based compression

Our first variant builds on the compression format of Raman et al. [24, 17], modified to be practical. We partition the bitvector B into chunks of $\Theta(w^2)$ bits and these become the leaves of an AVL tree. We store the chunks using the *(class, offset)* encoding (c, o) [24]: a chunk is further partitioned into blocks of $b = w/2$ bits; the class of a block is the number of 1s it contains and its offset is the index of the block among all possible blocks of the same class when sorted lexicographically. A class component requires $\lg w$ bits, while the offset of a block of class k requires $\lg \binom{b}{k}$ bits. All class/offset components are concatenated in arrays C/O, which are stored using our custom memory allocator. The overall space of this encoding is $nH_0(B) + o(n)$ bits [24]. The space overhead of the AVL tree is $\mathcal{O}(n/w)$ bits, since there are $\mathcal{O}(n/w^2)$ nodes, each requiring $\Theta(w)$ bits. Since $w = \Omega(\lg n)$, this overhead is $o(n)$. It is important to notice that while leaves represent $\Theta(w^2)$ *logical* bits, the *actual* space used by the (c, o) encoding may be considerably smaller. In practice we choose a parameter L', and all leaves will store a number of physical bytes in the range $[L', 2L']$.

To answer $access(B, i)/select(B, j)$ queries we navigate, using the AVL tree, to the leaf containing the target position and then decode the blocks sequentially until the desired position is found. A block is decoded in constant time using a lookup table that, given a pair (c, o), returns the original b bits of the block. This table has $2^{w/2}$ entries, which is small and can be regarded as program size, since it does not depend on the data. Note that we only need to decode the last block; for the previous ones the class component is sufficient to determine how much to advance in array O. For $rank_1(B, i)$ we also need to add up the class components (i.e., number of 1s) up to the desired block. Again, this only requires accessing array C, while O is only read to decode the last block. We spend $\mathcal{O}(\lg n)$ time to navigate the tree, $\mathcal{O}(w)$ time to traverse the blocks in the target leaf, and $\mathcal{O}(w)$ time to process the last block bitwise. Thus queries take $\mathcal{O}(w)$ time. In practice we set $b = 15$, hence the class components require 4 bits (and can be read by pairs from each single byte of C), the (uncompressed) blocks are 16-bit integers, and the decoding table overhead (which is shared by all the bitvectors) is only 64KB.

To handle updates we navigate towards the target leaf and proceed to decompress, update, and recompress all the blocks to the right of the update position. If the number of physical bytes stored in a leaf grows beyond $2L$ we split it in two leaves and add a new internal node to be tree; if it shrinks beyond L we move a single bit from the left or right sibling leaf to the current leaf. If this is not possible (because both siblings store L physical bytes) we merge the current leaf with one of its siblings; in either case we perform rotations on the internal nodes of the tree appropriately to restore the AVL invariant.

Recompressing a block is done using an encoding lookup table that, given a block of b bits, returns the associated (c, o) encoding. This adds other 64KB of memory. To avoid overwriting memory when the physical leaf size grows, recompression is done by reading the leaf data and writing the updated version in a separate memory area, which is later copied back to the leaf.

3.3 Compression of very sparse bitvectors

When the number m of 1s in B is very low, the $o(n)$ term may be significative compared to $nH_0(B)$. In this case we seek a structure whose space depends mainly on m. We present our second variant (also based on Mäkinen and Navarro [17]) that requires only $m \lg \frac{n}{m} + \mathcal{O}(m \lg\lg \frac{n}{m})$ bits, while maintaining the $\mathcal{O}(w)$-time complexities. This space is $nH_0(B)(1 + o(1))$ bits if $m = o(n)$.

The main building blocks is Elias δ-codes [6]. Given a positive integer x, let $|x|$ denote the length of its binary representation (eg. $|7| = 3$). The δ-code for x is obtained by writing $||x|| - 1$ zeros followed by the binary representation of $|x|$ and followed by the binary representation of x without the leading 1 bit. For example $\delta(7) = 01111$ and $\delta(14) = 00100110$. It follows easily that the length of the code $\delta(x)$ is $|\delta(x)| = \lg x + 2 \lg\lg x + \mathcal{O}(1)$ bits.

We partition B into chunks containing $\Theta(w)$ *1s*. We build an AVL tree where leaves store the chunks. A chunk is stored using δ-codes for the distance between pairs of consecutive 1s. This time the overhead of the AVL tree is $\mathcal{O}(m)$ bits. By using the Jensen inequality on the lengths of the δ-codes it can be shown [17] that the overall space of the leaves is $m \lg \frac{n}{m} + \mathcal{O}(m \lg\lg \frac{n}{m})$ bits and the redundancy of the AVL tree is absorbed in the second term. In practice we choose a constant M and leaves store a number of 1s in the range $[M, 2M]$. Within this space we now show how to answer queries and handle updates in $\mathcal{O}(w)$ time.

To answer $access(i)$ we descend to the target leaf and start decoding the δ-codes sequentially until the desired position is found. Note that each δ-code represents a run of 0s terminated with a 1, so as soon as the current run contains the position i we return the answer. To answer $rank(i)$ we increase the answer by 1 per δ-code we traverse. Finally, to answer $select_1(j)$, when we reach the target leaf looking for the j-th local 1-bit we decode the first j codes and add their sum (since they represent the lengths of the runs). Instead, $select_0(j)$ is very similar to the $access$ query.

To handle the insertion of a 0 at position i in a leaf we sequentially search for the δ-code that contains position i. Let this code be $\delta(x)$; we then replace it by $\delta(x+1)$. To insert a 1, let $i' \leq x+1$ be the local offset inside the run $0^{x-1}1$ (represented by the code $\delta(x)$) where the insertion will take place. We then replace $\delta(x)$ by $\delta(i')\delta(x-i'+1)$ if $i' \leq x$ and by $\delta(x)\delta(1)$ otherwise. In either case (inserting a 1 or a 0) we copy the remaining δ-codes to the right of the insertion point. Deletions are handled analogously; we omit the description. If, after an update, the number of 1s of a leaf lies outside the interval $[M, 2M]$ we move a run from a neighbor leaf or perform a split/merge just as in the previous solution and then perform tree rotations to restore the AVL invariant.

The times for the queries and updates are $\mathcal{O}(w)$ provided that δ-codes are encoded/decoded in constant time. To decode a δ-code we need to find the highest 1 in a word (as this will give us the necessary information to decode the rest). Encoding a number x requires efficiently computing $|x|$ (the length of its binary representation), which is also the same problem. Modern CPUs provide special support for this operation; otherwise we can use small precomputed tables. The rest of the encoding/decoding process is done with appropriate bitwise operations. Furthermore, the local encoding/decoding is done on sequential memory areas, which is cache-friendly.

4 Applications

4.1 Dynamic sequences

The wavelet matrix [5] is a compact structure for sequences $S[1, n]$ over a fixed alphabet $[1, \sigma]$, providing support for $access(i), rank_c(i)$ and $select_c(i)$ queries. The main idea is to store $\lg \sigma$ bitvectors B_i defined as follows: let $S_1 = S$ and $B_1[j] = 1$ iff the most significant bit of $S_1[j]$ is set. Then S_2 is obtained by moving to the front all characters $S_1[j]$ with $B_1[j] = 0$ and moving the rest to the back (the internal order of front and back symbols is retained). Then $B_2[j] = 1$ iff the second most significant bit of $S_2[j]$ is set, we create S_3 by shuffling S_2 according to B_2, and so on. This process is repeated $\lg \sigma$ times. We also store $\lg \sigma$ numbers $z_j = rank_0(B_j, n)$. The $access/rank/select$ queries on this structure reduce to $\mathcal{O}(\lg \sigma)$ analogous queries on the bitvectors B_j, thus the times are $\mathcal{O}(\lg \sigma)$ and the final space is $n \lg \sigma + o(n \lg \sigma)$ (see the article [5] for more details).

Our results in Section 3 enable a dynamic implementation of wavelet matrices with little effort. The insertion/deletion of a character at position i is implemented by the insertion/deletion of a single bit in each of the bitvectors B_j. For insertion of c, we insert the highest bit of c in $B_1[i]$. If the bit is a 0, we increase z_1 by one and change i to $rank_0(B_1, i)$; otherwise we change i to $z_1 + rank_1(B_1, i)$. Then we continue with B_2, and so on. Deletion is analogous. Hence all query and update operations require $\lg \sigma$ $\mathcal{O}(w)$-time operations on our dynamic bitvectors. By using our uncompressed dynamic bitvectors, we maintain a dynamic string $S[1, n]$ over a (fixed) alphabet $[1, \sigma]$ in $n \lg \sigma + o(n \lg \sigma)$ bits, handling queries and updates in $\mathcal{O}(w \lg \sigma)$ time. An important result [11] states that if the bitvectors B_j are compressed to their zero-order entropy $nH_0(B_j)$, then the overall space is $nH_0(S)$. Hence, by switching to our compressed dynamic bitvectors (in particular, our first variant) we immediately achieve $nH_0(S)+o(n\lg\sigma)$ bits and the query/update times remain $\mathcal{O}(w\lg\sigma)$.

4.2 Dynamic graphs and grids

The wavelet matrix has a wide range of applications [19]. One is directed graphs. Let us provide dynamism to the compact structure of Claude and Navarro [4]. Given a directed graph $G(V,E)$ with $n=|V|$ vertices and $e=|E|$ edges, consider the adjacency list $G[v]$ of each node v. We concatenate all the adjacency lists in a single sequence $S[1,e]$ over the alphabet $[1,n]$ and build the dynamic wavelet matrix on S. Each outdegree d_v of vertex v is written as 10^{d_v} and appended to a bitvector $B[1,n+e]$. The final space is $e\lg n(1+o(1))+\mathcal{O}(n)$ bits.

This representation allows navigating the graph. The outdegree of vertex v is computed as $\mathit{select}_1(B,v+1)-\mathit{select}_1(B,v)-1$. The j-th neighbor of vertex v is $\mathit{access}(S,\mathit{select}_1(B,v)-v+j)$. The edge (v,u) exists iff $\mathit{rank}_u(S,\mathit{select}_1(B,v+1)-v-1)-\mathit{rank}_u(S,\mathit{select}_1(B,v)-v)=1$. The main advantage of this representation is that it also enables *backwards* navigation of the graph without doubling the space: the indegree of vertex v is $\mathit{rank}_v(S,e)$ and the j-th *reverse* neighbor of v is $\mathit{select}_0(B,\mathit{select}_v(S,j))-\mathit{select}_v(S,j)$.

To insert an edge (u,v) we insert a 0 at position $\mathit{select}_1(B,u)+1$ to increment the indegree of u, and then insert in S the character v at position $\mathit{select}_1(B,u)-u+1$. Edge deletion is handled in a similar way. We thus obtain $\mathcal{O}(w\lg n)$ time to update the edges. Unfortunately, the wavelet matrix does not allow changing the alphabet size . Despite this, providing edge dynamism is sufficient in several applications where an upper bound on the number of vertices is known.

This same structure is useful to represent two-dimensional $n\times n$ grids with e points, where we can insert and delete points. It is actually easy to generalize the grid size to any $c\times r$. Then the space is $n\lg r(1+o(1))+\mathcal{O}(n+c)$ bits. The static wavelet matrix [5] can count the number of points in a rectangular area in time $\mathcal{O}(\lg r)$, and report each such point in time $\mathcal{O}(\lg r)$ as well. On our dynamic variant, times become $\mathcal{O}(w\lg r)$, just like the time to insert/delete points.

5 Experimental Results and Discussion

The experiments were run on a server with 4 Intel Xeon cores (each at 2.4GHz) and 96GB RAM running Linux version 3.2.0-97. All implementations are in `C++`.

We first reproduce the memory fragmentation *stress* test [16] using our allocator of Section 3.1. The experiment initially creates n chunks holding C bytes. Then it performs C steps. In the i-th step n/i chunks are randomly chosen and their memory area is expanded to $C+i$ bytes. We set $C=2^{11}$ and use the same settings [16] for our custom allocator: the cell size T is set to 2^{16} and L is set to 2^{11}. Table 1 shows the results. The memory consumption is measured as the *Resident Set Size (RSS)*,[1] which is the actual amount of physical RAM retained

[1] Measured with `https://github.com/mpetri/mem_monitor`

$\lg n$	`malloc` RSS	`custom` RSS	`malloc` time	`custom` time
18	0.889	0.768	0.668	0.665
19	1.777	1.478	1.360	1.325
20	3.552	2.893	2.719	2.635
21	7.103	5.727	5.409	5.213
22	14.204	11.392	10.811	10.446
23	28.407	22.725	21.870	21.163
24	56.813	45.388	45.115	43.081

Table 1. Memory consumption measured as RSS in GBs and CPU time (seconds) for the RAM fragmentation test.

by a running process. `Malloc` represents the default allocator provided by the operating system and `custom` is our implementation. Note that for all the tested values of n our allocator holds less RAM memory, and in particular for $n = 2^{24}$ (i.e., $nC = 32$GB) it saves up to 12GB. In all cases the CPU times of our allocator are faster than the default `malloc`. This shows that our implementation is competitive with the previous one [16], which reports similar space and time.

Having established that our allocator enables considerable reductions in RAM fragmentation, we study our practical compressed bitvectors. We generate random bitvectors of size $n = 50 \cdot 2^{23}$ (i.e., 50MB) and set individual bits to 1 with probability p. We consider skewed frequencies $p = 0.1, 0.01$, and 0.001. Preliminary testing showed that, for our variant of Section 3.2, setting the range of physical leaf sizes to $[2^{11}, 2^{12}]$ bytes provided the best results. Table 2 gives our results for the compression achieved and the time for queries and updates (averaging insertions and deletions). We achieve 0.3–0.4 bits of redundancy over the entropy, which is largely explained by the component c of the pairs (c, o): these add $\lg(b + 1)/b = 4/15 = 0.27$ bits of redundancy, whereas the O array adds up to $nH_0(B)$. The rest of the redundancy is due to the AVL tree nodes and the space wasted by our memory allocator. For the very sparse bitvectors ($p = 0.001$), the impact of this fixed redundancy is very high.

Operation times are measured by timing 10^5 operations on random positions of the bitvectors. The queries on our first variant take around $1\mu s$ (and even less for *access*), whereas the update operations take 8–$15\mu s$. The operations become faster as more compression is achieved.

For the very sparse bitvectors ($p = 0.001$) we also test our variant of Section 3.3. Preliminary testing showed that enforcing leaves to handle a number of 1s in the range $[128, 256]$ provided the best results. The last row of Table 2 shows the compression and timing results for this structure. As promised in theory, the compression achieved by this representation is remarkable, achieving 0.02 bits of redundancy (its space is much worse on the higher values of p, however). The query times become slightly over $1\mu s$ and the update times are around $5\mu s$.

Finally, we present a single application for the dynamic wavelet matrix and graphs. We find the weakly connected components of a sample of the *DBLP* social graph stored using the dynamic representation of Section 4.2 with plain

p	MB	Bits/n	$-H_0(B)$	Updates	Access	Rank	Select
0.1	38.57	0.77	0.30	15.08	0.80	1.10	1.20
0.01	21.27	0.43	0.35	10.77	0.60	0.90	1.10
0.001	19.38	0.39	0.38	8.50	0.70	0.90	1.00
$*$0.001	1.50	0.03	0.02	5.26	1.38	1.47	1.35

Table 2. Memory used (measured as RSS, in MB, in bits per bit, and in redundancy over $H_0(B)$) and timing results (in microseconds) for our compressed dynamic bitvectors. The first three rows refer to the variant of Section 3.2, and the last to Section 3.3.

Structure	RSS	ratio	Build time	ratio	BFS time	ratio
`std::vector`	22.20	1.00	7.40	1.00	0.06	1.00
wavelet matrix	4.70	0.21	12.42	1.68	9.34	155.67
previous [16]		0.30		9.00		30.00

Table 3. Memory usage (MBs) and times (in seconds) for the online construction and a breadth-first traversal of the *DBLP* graph to find its weakly connected components. The data for previous work [16] is a rough approximation.

dynamic bitvectors, that is, they are stored verbatim. We use range $[2^{10}, 2^{11}]$ bytes for the leaf sizes.

The sample dataset consists of 317,080 vertices and 1,049,866 edges taken from `https://snap.stanford.edu/data/com-DBLP.html`, with edge directions assigned at random. We build the graph by successive insertions of the edges. Table 3 shows the memory consumption, the construction time (i.e., inserting all the edges), and the time to perform a breadth-first search of the graph. Our baseline is a representation of graphs based on adjacency lists implemented using the `std::vector` class from the `STL` library in `C++`, where each directed edge (u, v) is also stored as (v, u) to enable backwards navigation. Considerable space savings are achieved using the dynamic wavelet matrix, 5-fold with respect to the baseline. The edge insertion times are very competitive, only 70% slower than the baseline. The time to perform a full traversal of the graph, however, is two orders of magnitude slower.

We now briefly make an informal comparison between our results and the best previous work [16] by extrapolating some numbers.[2] For bitvectors with density $p = 0.1$ our first variant achieves 77% compression compared to their 85%. For $p = 0.01$ ours achieves 43% compared to their 35%, and for $p = 0.001$ our second variant achieves 3% compared to their 6%. In terms of running times our results handle queries in about $1\mu s$ and updates in 8–$15\mu s$, while their most practical variant, based on `lz4`, handles queries and updates in around 10–$25\mu s$. These results are expected since the encodings we used ((c, o) pairs and δ-codes) are tailored to answer *rank*/*select* queries without the need of full decompression.

[2] A precise comparison is not possible since their results are not available. We use their plots as a reference.

Finally, they also implement a compressed dynamic graph (based on compressed RAM and not on compact structures). The rough results (extrapolated from their own comparison against `std::vector`) are shown in the last line of Table 3: they use more 50% more space and 5 times more construction time than our implementation, but their BFS time is 5 times faster.

6 Conclusions

We have presented the first practical entropy-compressed dynamic bitvectors with good space/time theoretical guarantees. The structures solve queries in around a microsecond and handle updates in 5–15 microseconds. An important advantage compared with previous work [16] is that we do not need to fully decompress the bit chunks to carry out queries, which makes us an order of magnitude faster. Another advantage over previous work is the guaranteed zero-order entropy space, which allows us using bitvectors for representing sequences in zero-order entropy, and full-text indexes in high-order entropy space [15].

Several improvements are possible. For example, we reported times for querying random positions, but many times we access long contiguous areas of a sequence. Those can be handled much faster by remembering the last accessed AVL tree node and block. In the (c, o) encoding, we would access a new byte of C every 30 operations, and decode a new block of O every 15, which would amount to at least an order-of-magnitude improvement in query times. For δ-encoded bitvectors, we would decode a new entry every n/m operations on average.

Another improvement is to allow for faster queries when updates are less frequent, tending to the fast static query times in the limit. We are studying policies to turn an AVL subtree into static when it receives no updates for some time. This would reduce, for example, the performance gap for the BFS traversal in our graph application once it is built, if further updates are infrequent.

Finally, there exist theoretical proposals [20] to represent dynamic sequences that obtain the optimal time $\mathcal{O}(\lg n/\lg\lg n)$ for all the operations. This is much better than the $\mathcal{O}(w \lg \sigma)$ time we obtain with dynamic wavelet matrices. An interesting future work path is to try to turn that solution into a practical implementation. It has the added benefit of allowing us to update the alphabet, unlike wavelet matrices.

Our implementation of dynamic bitvectors and the memory allocator are available at `https://github.com/jhcmonroy/dynamic-bitvectors`.

References

1. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. 12th ALENEX. pp. 84–97 (2010)
2. Brisaboa, N., de Bernardo, G., Navarro, G.: Compressed dynamic binary relations. In: Proc. 22nd DCC. pp. 52–61 (2012)
3. Clark, D.: Compact PAT Trees. Ph.D. thesis, Univ. Waterloo, Canada (1996)

4. Claude, F., Navarro, G.: Extended compact web graph representations. In: Algorithms and Applications (Ukkonen Festschrift). pp. 77–91. LNCS 6060 (2010)
5. Claude, F., Navarro, G., Ordóñez, A.: The wavelet matrix: An efficient wavelet tree for large alphabets. Inf. Sys. 47, 15–32 (2015)
6. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. Inf. Theory 21(2), 194–203 (1975)
7. Ferrada, H., Navarro, G.: Improved range minimum queries. In: Proc. 26th DCC (2016), to appear
8. Fredman, M., Saks, M.: The cell probe complexity of dynamic data structures. In: Proc. 21st STOC. pp. 345–354 (1989)
9. Gerlang, W.: Dynamic FM-Index for a Collection of Texts with Application to Space-efficient Construction of the Compressed Suffix Array. Master's thesis, Univ. Bielefeld, Germany (2007)
10. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Proc. 13th SEA. pp. 326–337 (2014)
11. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th SODA. pp. 841–850 (2003)
12. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS. pp. 549–554 (1989)
13. Jansson, J., Sadakane, K., Sung, W.K.: Cram: Compressed random access memory. In: Proc. 39th ICALP. pp. 510–521 (2012)
14. Joannou, S., Raman, R.: Dynamizing succinct tree representations. In: Proc. 11th SEA. pp. 224–235 (2012)
15. Kärkkäinen, J., Puglisi, S.: Fixed block compression boosting in fm-indexes. In: Proc. 18th SPIRE. pp. 174–184 (2011)
16. Klitzke, P., Nicholson, P.K.: A general framework for dynamic succinct and compressed data structures. In: Proc. 18th ALENEX. pp. 160–173 (2016)
17. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Trans. Alg. 4(3), article 32 (2008), 38 pages
18. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in o(log2 n) time. J. Comp. Sys. Sci. 33(1), 66–74 (1986)
19. Navarro, G.: Wavelet trees for all. J. Discr. Alg. 25, 2–20 (2014)
20. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. SIAM J. Comp. 43(5), 1781–1806 (2014)
21. Navarro, G., Sadakane, K.: Fully-functional static and dynamic succinct trees. ACM Trans. Alg. 10(3), article 16 (2014)
22. Okanohara, D.: Dynamic succinct vector library. https://code.google.com/archive/p/ds-vector/, accessed: 2016-01-30
23. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Proc. 7th WADS (2001)
24. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Alg. 3(4) (2007)
25. Salson, M.: Dynamic fm-index library. http://dfmi.sourceforge.net/, accessed: 2016-01-30
26. Smirnov, V.: Memoria library. https://bitbucket.org/vsmirnov/memoria/, accessed: 2016-01-30