Fast, Small, Simple Rank/Select on Bitmaps*

Gonzalo Navarro[1] and Eliana Providel[1,2]

[1] Department of Computer Science, University of Chile, gnavarro@dcc.uchile.cl
[2] Escuela de Ingeniería Civil Informática, Facultad de Ingeniería,
Universidad de Valparaíso, Valparaíso, Chile, eliana.providel@uv.cl

Abstract. *Rank* and *select* queries on bitmaps are fundamental for the construction of a variety of compact data structures. Both can, in theory, be answered in constant time by spending $o(n)$ extra bits on top of the original bitmap, of length n, or of a compressed version of it. However, while the solution for *rank* is indeed simple and practical, a similar result for *select* has been elusive, and practical compact data structure implementations avoid its use whenever possible. In addition, the overhead of the $o(n)$ extra bits is in many cases very significant. In this paper we bridge the gap between theory and practice by presenting two structures, one using the bitmap in plain form and another using a compressed form, that are simple to implement and combine much lower space overheads than previous work with excellent time performance for *rank* and *select* queries. In particular, our structure for plain bitmaps is far smaller and faster for *select* than any previous structure, while competitive for *rank* with the best previous structures of similar size.

1 Introduction

Compact data structures represent data within little space and can efficiently operate on it, in contrast to plain structures that do not compress, and with pure compression that needs full decompression in order to operate the data. They have become particularly interesting due to the increasing performance gap between successive levels in the memory hierarchy, in particular between main memory and disk. Since Jacobson's work [9], compact data structures for trees [9, 11], graphs [9, 11], strings [8, 6], and texts [8, 4], etc. have been proposed.

Jacobson [9] noticed that bit vectors were fundamental to support various compact data structures. In particular, he used the following two operations to simulate general trees, and since then these two operations have proved fundamental to implement many other compact structures:

$$rank_b(B, i) = \text{number of occurrences of bit } b \text{ in } B[0, i];$$
$$select_b(B, i) = \text{position of the } i\text{-th occurrence of bit } b \text{ in } B.$$

Much effort has been spent on implementing *rank* and *select* efficiently. In theory, they have long been solved in $O(1)$ time using just $o(n)$ bits on top of B

* Partially funded by Fondecyt Grant 1-110066 and Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

[2, 10], an even on top of a compressed representation of B using $nH_0(B) + o(n)$ bits [15, 14]. $H_0(B) = n_0 \lg(n/n_0) + n_1 \lg(n/n_1)$, where B has n_0 0s and n_1 1s, is the called the zero-order entropy of B, and is smaller when B has a few 0s or 1s. By default we assume $b = 1$ (many applications need just $rank_1$ and $select_1$).

The practical solutions for *rank* over plain bitmaps are indeed simple, and there exist implementations solving it within a few tenths of microsecond (μsec) using 5% of extra space on top of B [7]. For *select*, instead, no practical solution with guaranteed constant time exists. The original solution [2] leads to more than 60% extra space in practice [7]. Only very recently there have been better proposals running in a few tenths of μsec and requiring around 25% [12] and even 7% [16] of extra space, on a bitmap with half 0s and 1s. Even worse, those structures only solve *select*, and thus need to be coupled with another solving *rank*, and we need two copies of them to solve both $select_0$ and $select_1$ (this does not happen with *rank* because $rank_0(i) = i - rank_1(i)$). On the other hand, implementations of compressed solutions [15, 3] pose in practice an overhead over 27% (of the original bitmap size) on top of the entropy of the bitmap.

This situation has noticeably slowed down the growth of the area of compact data structures in practice: While many solutions building on *select*, or relying on bitmap compression, are very attractive in theory, in practice one must try to avoid the use of *select*, and have in mind the significant space overhead associated to the $o(n)$ terms. We largely remedy both problems in this paper.

First, we introduce a new implementation of compressed bitmaps [15] that retains the time performance of the current implementation [3] while drastically reducing its space overhead by 50%–60%. The resulting structure has a space overhead of around 10% (of the original bitmap size) on top of the entropy (compare to the 27% of the current implementation [3]) and solves *rank* queries within about 0.4 μsec and *select* within 1 μsec. This is achieved by replacing the use of universal tables by on-the-fly generation of their cells. These universal tables are of size exponential on a block size t, and the space overhead of the data structure is $O(n \lg t/t)$. Removing the tables let us use t values that are 4 times larger, thereby reducing the space by a factor of $\approx \lg 4/4 = 50\%$.

Second, we present a new combined data structure that solves *rank* and *select* on plain bitmaps, instead of solving each operation separately. It integrates two samplings, one for *rank* that is regular on the bitmap positions, and one for *select* that is regular on the positions of the 1s. Each operation uses its own sampling but takes advantage of the other if possible. We show that this structure is able to solve both *rank* and *select* queries within around 0.2 μsec, using just 3% of extra space on top of the plain bitmap. This is an unprecedented result, very far from what current representations achieve, that finally puts *rank* and *select* on the map of the operations that can be used in practice without reservations.

2 Related Work

Jacobson [9] showed that attaching a dictionary of size $o(n)$ to a bit vector $B[0, n-1]$ is sufficient to support *rank* operation in constant time on the RAM

model. Later, Munro [10] and Clark [2] obtained constant-time complexity for *select* on the RAM model, using also $o(n)$ extra space. Golynski [5] showed how to reduce the $o(n)$ term to $O(n \lg\lg n/\lg n)$, and that this space is optimal if B is stored explicitly. Others have studied representations that store a compressed form of B, useful when it has few or many 1s [13, 15, 12, 14]. Some [15, 14] use $nH_0(B) + o(n)$ bits and solve both operations in constant time, where $o(n)$ can be as small as $O(n/\lg^c n)$ for any constant c [14]. Others [12] solve *select* in constant time and *rank* in time $O(\lg(n/n_1))$, using $nH_0(B) + O(n_1)$ bits.

The original solution for *rank* was simple, easy to program, and required three table accesses. It has followed a simple path through practice. For example, a competitive implementation [7] solves *rank* in a few tenths of microseconds using 5% of extra space on top of B, and in a few hundredths using 37.5% of extra space. The first figure is sufficiently fast in most practical scenarios.

The solution for *select*, instead, has followed a more complicated path. The original constant-time solution [2, 10] is much more complicated than that of *rank*, and in practice it is much slower and requires more space. An implementation [7] showed that it requires more than 60% of extra space. Indeed, a simple solution that works better [7] for all reasonable bitmap sizes is to solve *select* via binary searches on the *rank* directories. This, however, makes *select* significantly slower than *rank*, and less attractive in practice.

Okanohara and Sadakane [12] proposed a simplification of the original solution that, although does not guarantee constant time, it works very well in practice (a few tenths of microseconds). It requires, however, around 25% of extra space and does not solve *rank*, so their complete *rank*/*select* solution for their structure called *dense array* requires more than 50% extra space. Later on, Vigna [16] achieved similar times within much less space overhead (7% to 12% in our experiments), with a structure called *simple-select*. Again, this structure does not solve *rank*. He proposed other two structures, *rank9sel* and *rank9b*, that solve both operations within the same time and a space overhead of around 24%.

As for compressed bitmaps, Claude and Navarro [3] implemented the solution by Raman et al. [15], achieving around 27% space overhead (this percentage refers to the original bitmap size) on top of the entropy of the bitmap, and solved *rank* within tenths of microseconds and *select* within a microsecond. This space overhead is very significant in practice. The structure is useful for densities (n_1/n) of up to 20% (at which point the entropy is more than 80% anyway, so not much can be done). Okanohara and Sadakane [12] presented a structure called *sparse array*, that uses $n_1 \lg(n/n_1) + O(n_1)$ bits, and is very fast to solve *rank* and *select* (as fast as *rank* on plain bitmaps). However, its space overhead is very large for all but very sparse bitmaps (densities below 5%).

3 A Structure for Compressed Bitmaps

We describe our *rank*/*select* data structure for compressed bitmaps. It is based on the proposal by Raman *et. al.* [15] and its practical implementation [3]. We first describe this implementation and then our improvement.

The structure partitions the input bitmap B into blocks of length $t = \lceil \frac{\lg n}{2} \rceil$. These are assigned to *classes*: a block with k 1s belongs to class k. Class k contains $\binom{t}{k}$ elements, so $\lceil \lg \binom{t}{k} \rceil$ bits are used to refer to an element of it. A block is identified with a pair (k, r), where k is its class ($0 \leq k \leq t$, using $\lceil \lg(t+1) \rceil$ bits), and r is the index of the block within the class (using $\lceil \lg \binom{t}{k} \rceil$ bits). A global universal table for each class k translates in $O(1)$ time any index r into the corresponding t bits. The sizes of all those tables add up to $2^t\, t$ bits.

The sequence of $\lceil n/t \rceil$ class identifiers k is stored in one array, K, and that of the indexes r is stored in another, R. Note the indexes are of different width, so it is not immediate how to locate the r value of the j-th block. The blocks are grouped in superblocks of length $s = \lfloor t \lg n \rfloor$. Each superblock stores the *rank* up to its beginning, and a pointer to R where its indexes start.

To solve $rank(i)$, we first compute the superblock i belongs to, which stores the *rank* value up to its beginning. Then we scan the classes from the start of the superblock, accumulating the k values into our *rank* answer. At the same time we maintain the pointer to array R: the superblock knows the pointer value corresponding to its beginning, and then we must add $\lceil \lg \binom{t}{k} \rceil$ for each class k we process. These values are obviously preprocessed, and note we do not access R. This scanning continues up to the block i belongs to, whose index is extracted from R and its bits are recovered from the universal table. We then scan the remaining bits of the block and finish. Solving $select(j)$ is analogous, except that we first binary search for the proper superblock and then scan the blocks.

The size of R is upper bounded by $nH_0(B)$, and the main space overhead comes from K, which uses $n\lceil \lg(t+1)/t \rceil$ bits (the superblocks add $O(n/t)$ bits, which is negligible). So we wish to have a larger t to reduce the main space overhead. The size $2^t\, t$ of the universal table, plus its low access locality, however, prevents using a large t. In the practical implementation [3] they used $t = 15$, so K is read by nibbles and the universal table requires only 64 KB of memory. The price is that the space overhead is $\lceil \lg(t+1)/t \rceil = 4/15 \approx 27\%$.

Our proposal. We propose to remove the universal table and compute its entry on the fly. Although we require $O(t)$ time to rebuild a block from its index r, we note this is done only once per query, so the impact should not be much. As a reward, we will be able to use larger block sizes, such as $t = 31$ (with a space overhead of $5/31 \approx 16\%$) and $t = 63$ (with a space overhead of $6/63 \approx 9.5\%$).[3]

The computation of *r*. At compression time, the index r of a block of length t and k 1s is assigned as follows. We start with $r = 0$. We consider the first bit. We number first the blocks with this first bit in zero (there are $\binom{t-1}{k}$ of them) and then those with the first bit in one (there are $\binom{t-1}{k-1}$). Therefore, if the first bit is zero we just continue with the next bit (now the block is of length $t-1$ and still has k 1s). Otherwise, we increase r by $\binom{t-1}{k}$ and continue with the next bit (now the block is of length $t-1$ and has $k-1$ 1s).

[3] We do not use blocks larger than 63 because we would need special multiword arithmetics in our computer, and all of our multiword approaches have been orders of magnitude slower than the native arithmetic.

Reconstructing a block on the fly. We must reverse the encoding process. By checking the range of values r belongs to, we extract the bits consecutively. If $r \geq \binom{t-1}{k}$, then the first bit of the block was a 1. In this case we decrement t and k, and decrease r by $\binom{t-1}{k}$. Otherwise, the first bit of the block was a 0. In this case we only decrement t. Now we continue extracting the next bit from a block of length $t-1$.

Note we can stop if r becomes zero, as this means that all the remaining bits are zero. Also, to solve *rank* or *select* queries, it may also be possible to stop this process before obtaining all the bits of the block. Finally, we (obviously) have all the binomial coefficients precomputed.

4 A Structure for Plain Bitmaps: Combined Sampling

We introduce a structure that answers *rank*/*select* queries on top of plain bitmaps. This structure combines two sampling schemes: the first one is a regular sampling on positions, that is, we divide the input bitmap B into blocks of fixed size S_r. Then, we store a sampling of *rank* answers from the beginning of B up to the end of each block, $\texttt{sampleRank}[j] = rank_1(B, j \cdot S_r)$. In the second sampling scheme, we create blocks containing exactly S_s 1s. Note these blocks are of variable length. We ensure that each block starts with a 1, to minimize unnecessary scanning. Thus $\texttt{sampleSelect}[j] = select_1(B, j \cdot S_s + 1)$. Both samplings are similar to the top-level samplings proposed in classical solutions [2, 10] to *rank* and *select* queries. The novelty is in how we combine them to solve each query.

Answering *rank* queries. To solve $rank_1(i)$, we start from block number $j = \lfloor S_r/i \rfloor$, up to which the number of 1s in B is $\texttt{sampleRank}[j]$, and scan the bitmap from position $i' = j \cdot S_r + 1$ to position i. This scanning is done bytewise, using a universal preocomputed table that counts the number of 1s in all the 256 bytes[4]. The sampling step S_r is always a multiple of 8, so that the scanning starts at a byte boundary. On the other extreme, we may need to individually scan the bits of the last byte.

Up to this point the technique is standard, and in fact corresponds to the "small space" variant of González et al. [7]. The novelty is that we will also make use of table `sampleSelect` to speed up the scanning. Imagine that $r = \texttt{sampleRank}[j]$, and we have to complete the scanning from bit positions i' to i. It is possible that some `sampleSelect` values point between i' and i, and we can easily find them: compute $r' = \lfloor r/S_s \rfloor$. Then $\texttt{sampleSelect}[r']$ points somewhere before i'. We scan $\texttt{sampleSelect}[r'+k]$ for $k = 1, 2 \ldots$, until $\texttt{sampleSelect}[r'+k+1] > i$. At this point we call $i'' = \texttt{sampleSelect}[r'+k] \leq i$, and know that $rank_1(i'') = (r'+k) \cdot S_s + 1$. Thus we only need to scan from $i''+1$ to i. This can give a speedup unless $k = 0$.

A final tweak is that starting from positions $i''+1$ that are not byte-aligned is cumbersome and affects the performance. For this reason we slightly change

[4] In some architectures, a native *popcount* operation may be faster. We have not exploited this feature as it is not standard.

the meaning of table `sampleSelect`$[j]$. Instead of pointing to the precise *bit* position of the $(j \cdot S_s + 1)$-th 1, it points to its *byte* position, so the scanning is always resumed from a byte-aligned position. A problem is that now do not know which is the *rank* value up to the beginning of the byte (previously we knew that there were exactly $j \cdot S_s + 1$ bits set up to position `sampleSelect`$[j]$). Fortunately, pointing to bytes instead of bits frees three bits from the integers where the positions are stored. The three least significant bits are therefore used to store *correction* information, namely the number of 1s from the beginning of the byte pointed by `sampleSelect`$[j]$ to just before the exact position of the $(j \cdot S_s + 1)$-th 1. Therefore, the number of 1s up to the beginning of the byte pointed by `sampleSelect`$[j]$ is $j \cdot S_s$ minus the correction information.

Answering *select* queries. To solve $select_1(B, i)$, we first compute $j = \lfloor S_s/i \rfloor$, and use `sampleSelect`$[j]$ to find a byte-aligned position p up to where we know the value $r = rank_1(B, p-1)$. Then we scan byte-wise, incrementing r using the universal *popcount* table, until we find a byte where we exceed the rank value we are looking for, $r > i$. Then we rescan the last byte bit-wise until finding the exact position of the i-th 1 in B.

Once again, the byte-wise scanning can be sped up because we may go through several sampled values in `sampleRank`. We compute $q = \lfloor p/S_r \rfloor$, and scan the successive values `sampleRank`$[q+k]$, $k = 1, 2 \ldots$ until `sampleRank`$[q+k+1] > i$. Then we may start the byte-wise scanning from position $p' = (q+k) \cdot S_r$, with rank value $r =$ `sampleRank`$[q+k]$, thus speeding up the process.

5 Experimental Results

We compare our implementations with various existing solutions. All the experiments were carried out on a machine with two IntelCore2 Duo processors, with 3 Ghz and 6 MB cache each, and 8 GB of RAM. The operating system is Ubuntu 8.04.4. We used `gcc` compiler with full optimization. We experiment on bitmaps of length $n = 2^{28}$ and average all our values over one million repetitions.

5.1 Compressed representations

To benchmark this structure we compared the performance of *rank* and *select* operations with the original Raman *et. al.* (`RRR` [15]) implementation [3], and with the sparse array of Okanohara and Sadakane (`Sada sparse` [12]). For our structure we considered block sizes of $b = 15$, 31, and 63, and for each block size we considered superblocks of 32, 64, and 128 blocks. (Recall that RRR can use only blocks of size 15.) We generated random bitmaps with densities 5%, 10%, and 20% (their zero-order entropies are 0.286, 0.469, and 0.772, respectively).

Fig. 1 gives the results. Increasing the block size from 15 to 63 greatly reduces the space overhead of the representation, to around 40%–50% of its value.

Our implementation is 30-50% faster than RRR for *select* queries, even with the same block size $b = 15$. Our times for *select* are almost insensitive to the

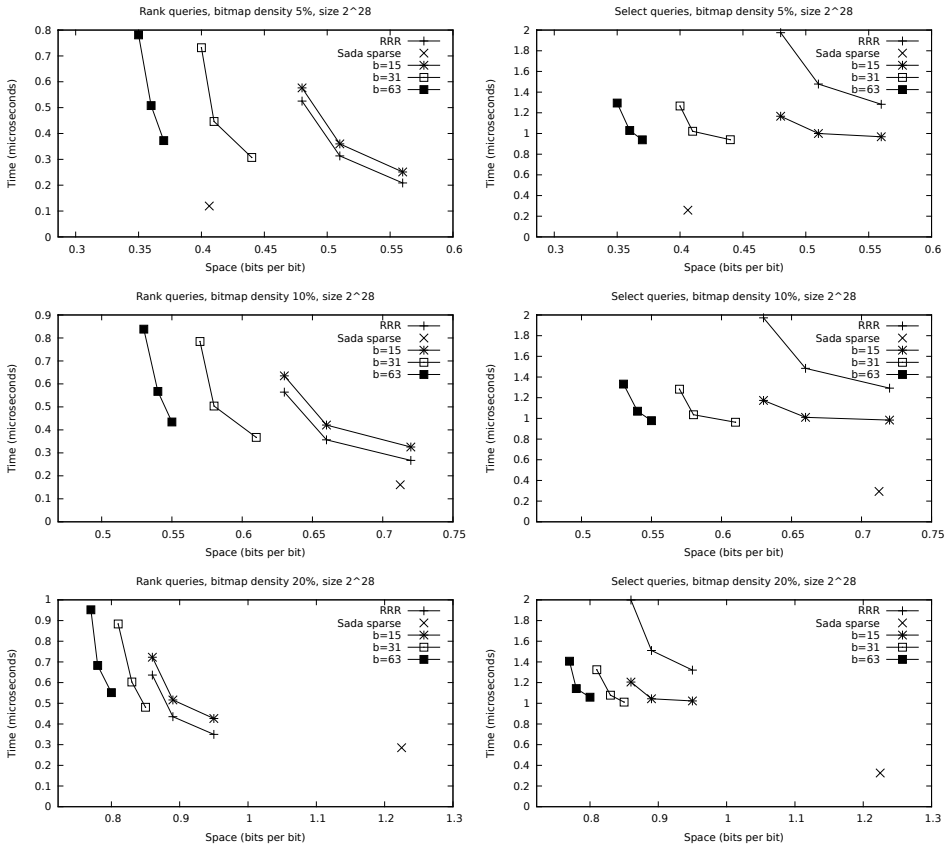

Fig. 1. Results for compressed bitmaps. Time is measured in μsec per query and space in total bits per bit of the bitmap. The x coordinates start at the zero-order entropy of the bitmap, so that the redundancy can be appreciated.

block size. This shows that the time to decode a block is negligible compared to that of the binary search. On the other hand, smaller superblocks make the operation faster, and a superblock of 32 or 64 blocks is recommended.

RRR is slightly faster than our implementation for *rank*, and also using longer blocks has a more noticeable (but still slight) effect on the performance. Overall, the price is very low compared to the large space gain.

Finally, Sada sparse is significantly faster than all other implementations, especially for *select*. However, its space is competitive only for very low densities.

Overall, using block size $b = 63$ and superblocks of 32 or 64 blocks, our data structure computes *rank* in about half a microsecond, *select* in about one microsecond, and requires a space overhead of less than 10% on top of the entropy. Overall this is a very convenient alternative for sparse bitmaps (of density up to 20%, as thereafter the entropy becomes too close to 1.0). For very sparse bitmaps

(density well below 5%), Sada sparse technique becomes the best choice. Note that our result would immediately improve on 128-bit processors.

5.2 Combined Sampling

We compare our combined sampling structure with several others. We consider, from González et al. [7], the variant that use blocks and superblocks and pose 37.5% extra space (`RG 37%`) and the one that uses one level of blocks (`RG 1Level`). From Vigna [16] we consider variants `rank9sel`, `rank9b`, and `simple-select`. From Clark [2] we consider its implementation for *select* [7] that uses 60% of extra space (`Clark`). Finally, from Okanohara and Sadakane [12] we use their dense bitmaps (`Sada dense`).

For our structure, we try different S_r and S_s combinations in the range $[2^8, 2^{13}]$. We show a curve for each S_r, with varying values of S_s. In addition, to test whether our combined approach is better than each structure separately, we add a modified Combined Sampling structure in which each kind of query only uses its respective supporting structure. In that case we use only the S_r (for *rank*) or S_s (for *select*) sampling, without using the other sampling.

We made two experiments: in the first, we evaluate *rank* and *select* queries separately, to study their performance. In the second, we emulate a real application that requires both kinds of queries, and test combined sequences with different proportions of *rank*/*select* queries.

Independent queries We execute either *rank* or *select* queries over bitmaps of densities 10%, 50%, and 90%. The results are shown in Fig. 2.[5]

Note that our techniques provide a continuum of space/time tradeoffs by varying S_r and S_s. The most interesting zone is the one where the extra space is below 5% and the times are below 0.2 μsec (for *rank*) and 0.3 μsec (for *select*). In particular, using $S_r = 1024$ and $S_s = 8192$ achieves extra space around 3% and times in the range 0.1–0.2 μsec for *rank* and 0.2–0.25 for *select*. This space/time tradeoff had never been achieved before.

By regarding the performance without using S_s, it can be seen that the use of the sampling of *select* does not make a big difference for *rank* (so we could use the basic technique that does not use `sampleSelect`). However, the performance without using S_r shows that *select* is sped up considerably by using the sampling of *rank*. As a result, our combined structure is more than two separate structures put together, as most of the structures in the literature.

For *rank*, structures like rank9b, rank9sel, and Sada dense, occupy significantly more space and are not noticeably faster than our 3%-space combination. Only structure RG 37% is considerably faster, yet it requires 10 times more

[5] We are aware that most of our curves are visually indistinguishable. However, our aim is to show that, using *some* parameterization (for which we will give explicit recommendations), our curves improve upon previous work, which is clearly distinguishable. Thus our curves can be regarded as a single, thick curve. We have not replaced them by a single one formed by the dominant points because we want to emphasize that the results are not much sensitive to a careful parameterization.

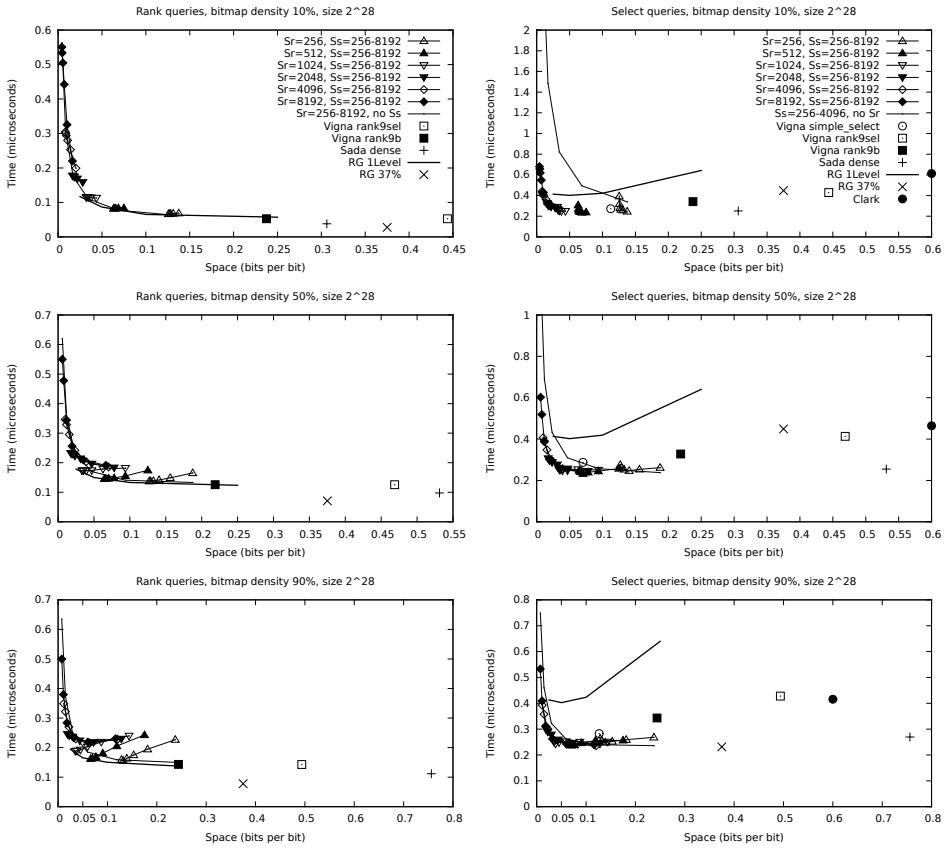

Fig. 2. Rank/Select queries for plain bitmaps. Time is measured in μsec per query and space in extra bits per bit of the bitmap.

space. Structure RG 1Level achieves a performance very similar to that of our new structure, as expected.

For *select*, however, RG 1Level performs poorly. Other structures that do not achieve better time than our 3%-space combination, while using much more space, are rank9b, rank9sel, RG37%, Sada dense, and Clark. The best performance, still not competitive with ours, is that of simple-select. This one reaches similar time, but 2–3 times more space than our 3%-space combination.

Mixed queries In this experiment we use bitmap density 50% and execute mixed queries, which combine *rank* and *select*, to mimic applications that need both of them. We try *rank*/*select* proportions of 90%/10%, 70%/30%, 50%/50%, and 30%/70%. We include a variant of our structure with the same value for both samplings, $S_r = S_s$. The results are shown in Fig. 3 (see also Footnote 3).

It can be seen that, as soon as we have more than 10% of *select* queries, our new structures become unparalleled. This is because operation *select* is, in

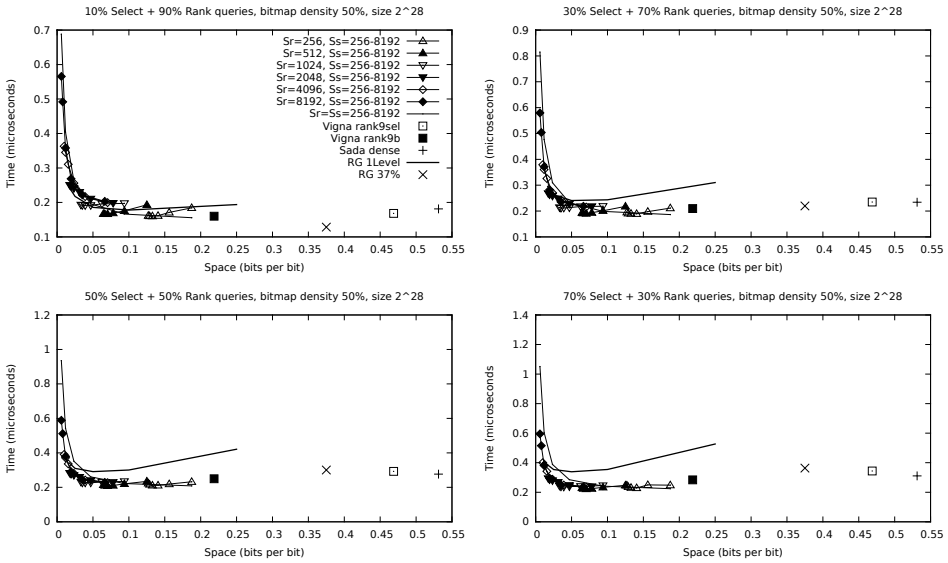

Fig. 3. Performance of mixed *rank*/*select* queries. Time is measured in μsec per query and space in extra bits per bit of the bitmap.

other structures, much slower than *rank*, and therefore a small fraction of those queries affects the overall time. Our structures, on the other hand, retain a performance close to 0.2 μsec within 3% of extra space, achieved with $S_r = 1024$ and $S_s = 8192$. The variant using the same value for both samples is usually suboptimal: Spending more space on the *rank* sampling pays off.

5.3 Applications

We illustrate the many applications of *rank*/*select* data structures with two examples. In the first one, we use our compressed representation to reimplement an FM-index [4]. This is a data structure that represents a text collection in compressed form and offers search capabilities on it. Its most practical and compact implementation [3] uses a Huffman-shaped tree storing at most $n(H_0(T)+1)$ bits, and those bits are represented in compressed form. The FM-index can count the number of occurrences of a pattern P in T via a number of *rank* operations over the bitmaps. We took this FM-index implementation from `http://libcds.recoded.cl`, and compared this implementation with one where we changed the compressed bitmaps to our new implementation. We indexed the 50 MB English text from `http://pizzachili.dcc.uchile.cl` and searched for 50,000 patterns of length 20 extracted at random from the text.

As Fig. 4 (left) shows, our index with $b = 63$ is slightly slower than the original, but still it counts in 90 μsec using 0.28 bits per bit, whereas the original one needs 0.335 bits per bit to achieve the same time. That is, we reduce the

space by around 16%. Indeed, we achieve the least space ever reported for an FM-index for this text (the results on other texts were similar).

Our second experiment uses a recent benchmark on compact representations of general trees of n nodes, using $2n+o(n)$ bits [1]. They showed that LOUDS [9] was one of the best performers, if its limited functionality was sufficient. LOUDS simulates various tree operations (like going to the parent, to the i-th child, etc.) via *rank*/*select* queries on the $2n$ bits that describe the tree topology.

We consider their 72-million node "suffix tree" [1] and, following their experimental setup, choose 33,000 random nodes with 5 children or more, and carry out the operation that finds the 5th child. Fig. 4 (right) shows the effect of replacing the *rank*/*select* structure used in LOUDS with our combined sampled structure (using various samplings in the range $S_r = [512, 2048]$ and $S_s = [1024, 32768]$). As it can be seen, LOUDS is much better than the alternatives, and we improve RG-based LOUDS as soon as we use 3% of extra space or more, reaching an improvement of up to 20% when using the same space.

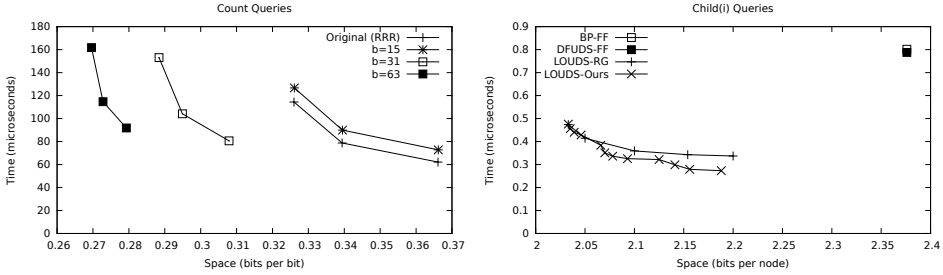

Fig. 4. On the left, counting on the FM-index. On the right, operation *child* on suffix trees. Time is measure in μsec per query, and space in overall number of bits per bit of the plain encoding of the sequence.

6 Conclusions

We have introduced two new techniques that aim at solving two recurrent problems in *rank*/*select* data structures: (1) too much space overhead, (2) bad performance for *select*. We have achieved (i) a structure for compressed bitmaps that retains the performance of existing ones, but reduces their space overhead by 50%–60%, reaching just 0.1 bit of space overhead over the entropy and solving *rank* in about 0.4 μsec and *select* in about 1 μsec; and (ii) a structure for plain bitmaps that combines *rank*/*select* data in a way that achieves just 3% of extra space and solves both operations in about 0.2 μsec, very far from the current space/time tradeoffs. In additions, our structures are rather simple.

These are two very relevant improvements that will find immediate applications. We have already illustrated one application to compressed text indexes and another to compact representations of trees.

Our data structures could perform slower for *select* if the bitmap had very long areas without 1s, even if we would traverse then fast using the S_r sampling. For such long areas we could apply the basic idea of Clark [2] and store the precomputed *select* answers inside. This requires insignificant extra space if applied on very long blocks. Also, in order to support $select_0$ we could add another sparse sampling using S_s (recall that in the recommended setup this stores just one integer every 8192 positions, so doubling this sampling impacts very little). Finally, it is natural to combine our two ideas into a single index for compressed bitmaps: right now our compressed representation carries out *select* by binary searching the superblocks. Our combined structure could speed this up to the times for *rank*, which are around 0.4 μsec on the compressed structure. This is our future work plan.

References

1. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th ALENEX*, pages 84–97, 2010.
2. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
3. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, LNCS 5280, pages 176–187, 2008.
4. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.
5. A. Golynski. Optimal lower bounds for rank and select indexes. In *Proc. 33th ICALP*, LNCS 4051, pages 370–381, 2006.
6. A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.
7. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. 4th WEA*, pages 27–38, 2005. Posters.
8. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
9. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554, 1989.
10. I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.
11. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th FOCS*, pages 118–126, 1997.
12. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th ALENEX*, 2007.
13. R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th ICALP*, pages 595–604, 1999.
14. M. Patrascu. Succincter. In *Proc. 49th FOCS*, pages 305–313, 2008.
15. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
16. S. Vigna. Broadword implementation of rank/select queries. In *Proc. 7th WEA*, LNCS 5038, pages 154–168, 2008.