On Dynamic Bitvector Implementations

Saska Dönges* Simon J. Puglisi* Rajeev Raman†

*Univeristy of Helsinki
†University of Leicester

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

CONTENTS

1. Brief problem statement and description of where we started
2. Our contributions and the effects these have on performance
3. Future work

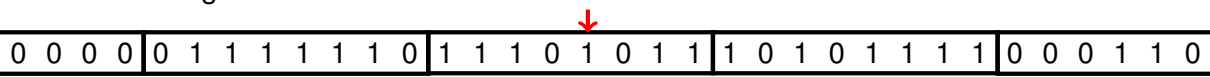
BACKGROUND

Supported operations for static bit vectors are

- Access ($\mathsf{at}(i) =$ value of i-th bit),
- Rank ($\mathsf{rank}(i) = \sum_{j=0}^{i} \mathsf{at}(i) =$ number of 1-bits in the first i bits), and
- Select ($\mathsf{select}(i) = \operatorname{argmin}_j \mathsf{rank}(j) \geq i =$ location of the i-th 1-bit)

.

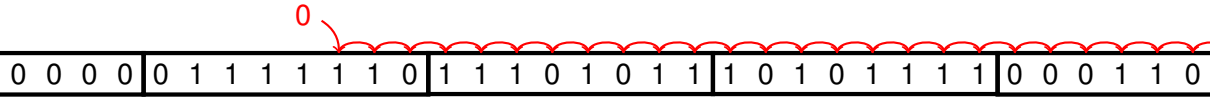
Bitvectors, even uncompressed, are useful building blocks for space efficient data structures, e.g. Compressed strings that support random access are commonly used in e.g. bioinformatics.



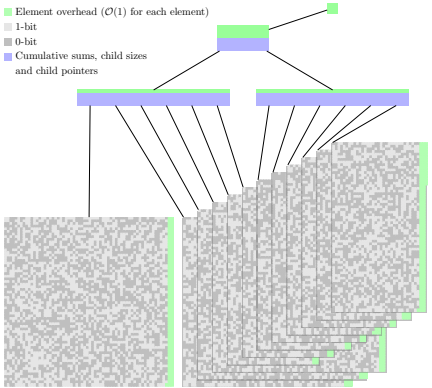
BACKGROUND

If the bitvector is made to support modifications (set, insert, remove), the derived data structures can also be made dynamic.

Tha same representation for the bit vector can't be used practically since any insertion or removal requires rewriting the bitvector (and any support structure) in the worst case.



BACKGROUND



State-of-the-art currently* is a B-tree-like data structure implemented by Nicola Prezza†.

The bitvector implementation (among other neat stuff) is contained in the `DYNAMIC` C++ template library: `https://github.com/xxsds/DYNAMIC`

*To the best of our knowledge

†Prezza, N. (2017). "A Framework of Dynamic Data Structures for String Processing". LIPIcs.

Our contributions

1. Full rewrite of the data structure from scratch.
2. Branch selection using branchless binary search.
3. Buffering insertions / removals at the leaf blocks.
4. Temporary static support structures.
5. Dynamic hybrid leaf compression using run-length-encoding.

Our contributions

1. Full rewrite of the data structure from scratch.
2. Branch selection using branchless binary search.
3. Buffering insertions / removals at the leaf blocks.
4. Temporary static support structures.
5. Dynamic hybrid leaf compression using run-length-encoding.

CONTRIBUTIONS – REWRITE

The implementation of the dynamic “Searchable partial sum” and leaf structures in `DYNAMIC` is not specifically engineered with just bitvectors in mind.

Rewriting the entire structure purely for bitvectors allowed for some optimizations that would be more difficult to implement as changes to `DYNAMIC`.

Most of the current code base is highly optimized for code path and cache efficiency.

CONTRIBUTIONS – REWRITE

In addition, changes to how the tree is balanced to eliminate insertion and removal oscillations present in `DYNAMIC`.

Before splitting or merging, we check if shuffling elements between nodes is sufficient. And if splitting or merging is required, the operations are done so that the same node will not need to be rebalanced again soon.

After any balancing operation, the involved nodes (internal or leaf) are guaranteed to support $\Theta(b)^\dagger$ operations before rebalaning.

---

†where b is the maximum node / leaf size

CONTRIBUTIONS – BRANCHING

To execute operations on the bitvector, the correct subtree branch needs to be selected at each internal node, based on cumulative sums and child sizes.

These branch selections form a root to leaf path.

`DYNAMIC` uses a linear scan for branch selection. This is extremely fast for low branching factors.

Our branch selection is based on branchless binary search. However, instead of the typical `cmov`-based implementations, we chose to limit the allowed universe size to allow us to use the "sign-bit" of the partial sums for calculating branching.

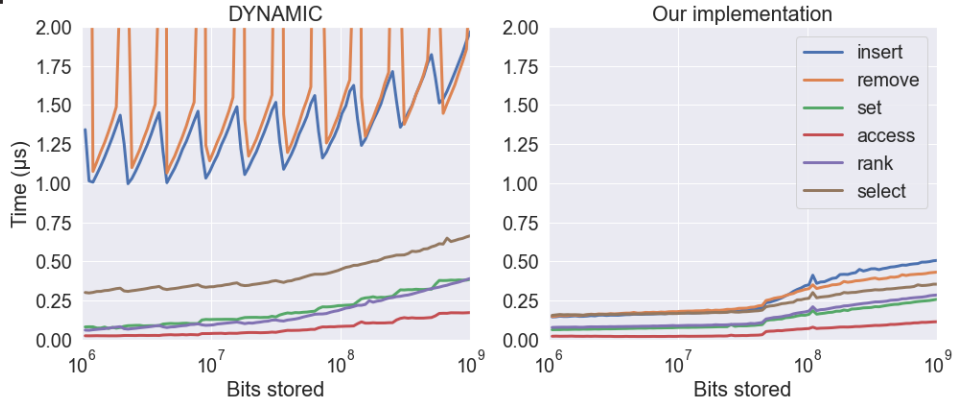
CONTRIBUTIONS – BRANCHING

```
uint8_t find(dtype q) const {
    idx = (uint8_t(1) << 5) - 1;
    idx ^= (dtype((elems_[idx] - q) & SIGN_BIT) >> (num_bits - 6)) | (uint8_t(1) << 4);
    idx ^= (dtype((elems_[idx] - q) & SIGN_BIT) >> (num_bits - 5)) | (uint8_t(1) << 3);
    idx ^= (dtype((elems_[idx] - q) & SIGN_BIT) >> (num_bits - 4)) | (uint8_t(1) << 2);
    idx ^= (dtype((elems_[idx] - q) & SIGN_BIT) >> (num_bits - 3)) | (uint8_t(1) << 1);
    idx ^= (dtype((elems_[idx] - q) & SIGN_BIT) >> (num_bits - 2)) | uint8_t(1);
    return idx ^ (dtype((elems_[idx] - q) & SIGN_BIT) >> (num_bits - 1));
}
```

This along with prefetching allows us to use a higher branching factor without significant penalty to branch selection.

RESULTS I

Performance comparison to DYNAMIC.



Mean operation times for random operations as function of number of bits stored.

CONTRIBUTIONS – BUFFERING

Instead of writing to the leaf directly, we register the operation in a buffer and write multiple insertions / removals at once.

We add a small b-element buffer or 32 bit integers to each ℓ element leaf node.

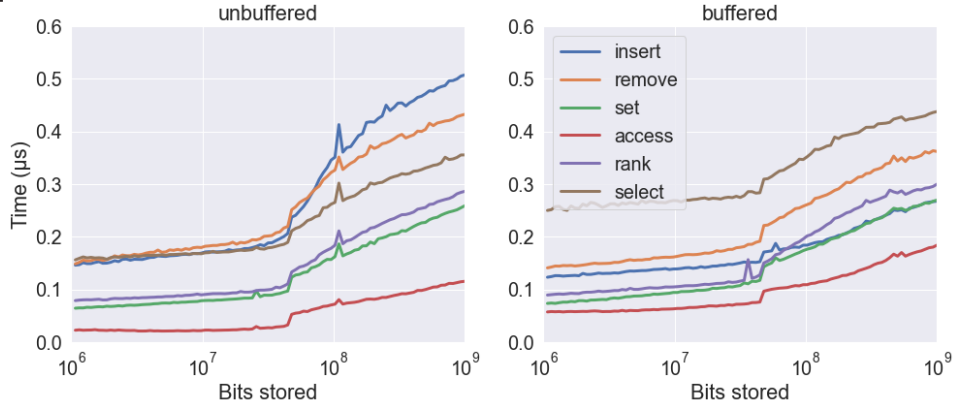
This changes the time complexity of insertions and removals (at the leaves) from $\mathcal{O}(\ell)$ to $\mathcal{O}(b + \ell/b)$.

Unfortunately this also adds a $\mathcal{O}(b)$ component to all queries.

Would ideally improve overall performance, especially for modification heavy workloads.

RESULTS II

Effects of buffering.



Insertions and removals speed up significantly. Select slows down

CONTRIBUTIONS – SUPPORT STRUCTURES

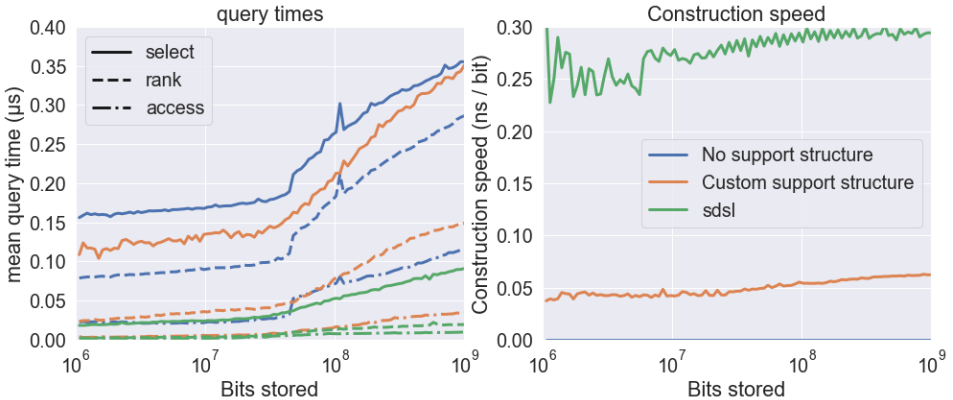
Create a temporary static support structure for the bitvector to speed up queries.

In practice the support structure is either an SDSL* bitvector with associated support structures, or a custom support structure build specifically for our dynamic bitvector.

*S. Gog, T. Beller, A. Mffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In Proc. SEA, LNCS, pages 326-337. Springer, 2014.

RESULTS – III

Construction time / speed trade-off.



Using SDSL takes and additional $\sim 1.4n$, and the the custom structure an additional $\sim 0.2n$ bits of

CONTRIBUTION – COMPRESSION

Inspired by Kärkkäinen et al.*, we implemented a hybrid encoding scheme for the leaves.

Currently the encoding for each leaf is dynamically chosen to be plain or run-length-encoded.

RLE was chosen due to the relative simplicity of implementation in the current architecture.

*J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Hybrid compression of bitvectors for the FM-index. In Proc. DCC, pages 302-311. IEEE, 2014.

CONTRIBUTIONS – COMPRESSION

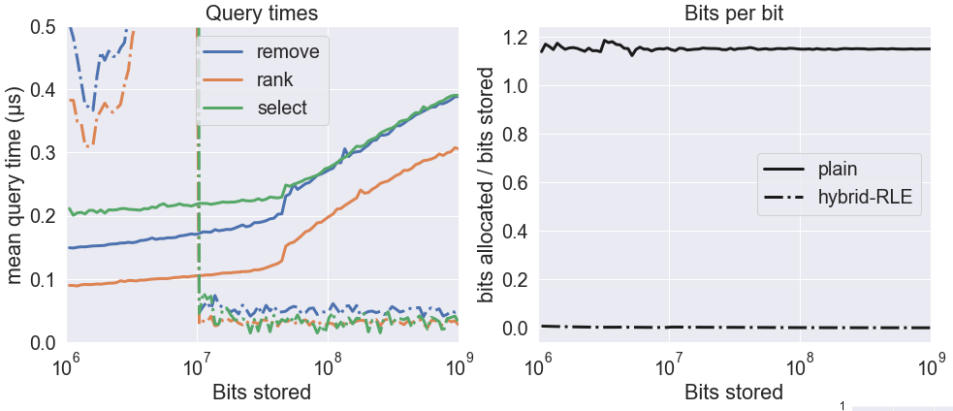
The encoding for a leaf can be changed any time the buffer of the leaf is committed due to being full.

This has the effect of eliminating the need to repeatedly change leaf encodings.

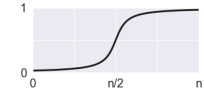
Current implementation is very resistant to splitting the first few leaves, leading to some strange behaviour for some “small” data structure sizes.

RESULTS – IV

Performance of hybrid RLE.



Probability of 1-bit goes from very low to very high:



RESULTS – V

	kernel 20M			cere		
cw-bwt	DYNAMIC	plain	hybrid-RLE	DYNAMIC	plain	hybrid-RLE
Time (s)	164	51	147	1982	817	1521
RSS (kB)	21312	18472	13024	184304	179032	86760
h0-LZ77						
Time (s)	641	647	1492	4650	2494	4572
RSS (kB)	21276	20432	10796	227908	217524	52896
Allocated (kB)	18561	17347	5892	213075	205349	45998

Table: Performance of the DYNAMIC's `suc_bv` bitvector and our plain and hybrid-rle compressed bitvectors on the cw-bwt and h0-LZ77 benchmarks from the DYNAMIC library.

FUTURE

There is still much to do with our dynamic bitvector.

There are multiple practical and quality of life improvements required to make the current implementation more usable

In additon, there are multiple unexplored avenues for research.

PRACTICAL STUFF

At least the following need to be “fixed”:

- Code quality improvements
- Strange behaviour with small hybrid-RLE bitvectors should be resolved
- Codepath and cache performance optimization should be done for the support structures and the hybrid approach

FURTHER RESEARCH

- At least minority bit compression should be added to the hybrid approach
- The current `malloc`-based allocation should be evaluated and possibly replaced
- We have ideas for alternatives to the current buffering implentation
- Some use-cases generate repeated queries that could be serviced by query caching
- Buffering at internal nodes could be beneficial but practical implementation is unclear
- Research effects of further vectorization with e.g. AVX512.

CONCLUSIONS

Performance comparison to `DYNAMIC`.