From Theory to Practice: Plug and Play with Succinct Data Structures

Simon Gog[1] Timo Beller[2] Alistair Moffat[1] Matthias Petri[1]

[1] Dept. Computing and Information Systems, The University of Melbourne, Victoria 3010, Australia
[2] Inst. Theoretical Computer Science, Ulm University, D-89069, Germany

Abstract. Engineering efficient implementations of compact and succinct structures is time-consuming and challenging, since there is no standard library of easy-to-use, highly optimized, and composable components. One consequence is that measuring the practical impact of new theoretical proposals is difficult, since older baseline implementations may not rely on the same basic components, and reimplementing from scratch can be time-consuming. In this paper we present a framework for experimentation with succinct data structures, providing a large set of configurable components, together with tests, benchmarks, and tools to analyze resource requirements. We demonstrate the functionality of the framework by recomposing two succinct solutions for top-k document retrieval which can operate on both character and integer alphabets.

1 Introduction

The field of succinct data structures (SDSs) has evolved rapidly in the last decade. New data structures such as the FM-Index, Wavelet Tree (WT), Range Minimum Query Structure (RMQ), Compressed Suffix Array (CSA), and Compressed Suffix Tree (CST) have been developed, and been shown to be remarkably versatile. These structures provide the same functionality as the corresponding uncompressed data structures, but do so using space which is asymptotically close to the information-theoretic lower bound needed to store the underlying data or objects. Using standard models of computation, the asymptotic runtime complexity of the operations performed by SDSs is also often identical to their classical counterparts. However, in practice, SDSs tend to be slower than the uncompressed structures, due to more complex memory access patterns on bitvectors, including non-sequential processing of unaligned bits. That is, they come in to their own only when the data scale means that an uncompressed structure would not fit in to main memory (or any particular level of the memory hierarchy), but a compressed structure would.

Accessing (ACCESS), counting (RANK), and selecting (SELECT) bits in bitvectors (BVs) are the fundamental operations from which more intricate operations are constructed. All three foundational operations can be supported in constant time adding only sublinear space. Wavelet trees build on BVs and generalize the three operations to alphabets of size $\sigma > 2$, with a corresponding increase in time to $\mathcal{O}(\log \sigma)$. A further layer up, some CSAs use WTs to realize their functionality; in turn, CSAs are the basis of yet more complex structures such as CSTs. Multiple alternatives exist at each level of this dependency hierarchy. For example, WTs differ in both shape (uniform versus Huffman-shaped) and in the choice made in the lower hierarchy levels (whether compressed or uncompressed BVs are used). The diversity of options allows structures to be composed that have a variety of time-space trade-offs. In practice, many of the complex structures that have been proposed are not yet fully implemented, since the implementations of underlying structures are missing. The use of non-optimized or non-composable substructures then prevents thorough empirical investigations, and makes it difficult to carry out impartial comparisons. The cost of implementing different approaches also creates a barrier that makes it difficult for new researchers, including graduate students, to enter the field.

As part of our investigation into SDSs, a library of flexible and efficient implementations has been assembled, providing a modular "plug and play, what you declare is what you get" approach to algorithm implementation. Instrumentation and visualization tools are also included in the library, allowing space costs to be accurately measured and depicted; as are efficient routines for constructing (in-memory as well as semi-external) and serializing all internal representations, allowing files containing succinct structures to be generated and re-read. We have also incorporated recent hardware developments, so that built-in popcount operations are used when available, and hardware hugepages can be enabled, to bypass address translation costs.

With this resource it is straightforward to *compose* complex structures; *measure* their behavior; and *explore* the range of alternatives. Having an established and robust code base also means that experimental comparisons of new data structures and algorithms can be made more resilient, providing greater consistency between alternative structures, and allowing better baseline systems and hence more reproducible evaluations. For example, different CST components might be appropriate to different types of input sequence (integer alphabet versus byte-based; highly-repetitive or not); and different sampling rates and access methods might be required. Using the library the right combination of components for any given application can be readily determined. The generality embedded in the library means that it is also useful in other fields such as information retrieval, natural language processing, bioinformatics, and image processing.

We illustrate the myriad virtues of the library – version 2 of SDSL – through two case studies: first, a detailed recomposition and re-evaluation of succinct document retrieval systems (Section 2); and second, a detailed examination of the construction processes used to create indexing structures (Section 3).

2 Document Retrieval Recomposed

The *top-k document retrieval problem* is fundamental in information retrieval (IR), and has become an active research topic in the succinct data structures community, see Navarro [11] for an excellent overview. For a collection of N documents $\mathcal{C} = \{d_1, \ldots, d_N\}$ over an alphabet Σ of size σ, a query $\mathcal{Q}$ also a set of strings over Σ, and a ranking function $R : \mathcal{C} \times \mathcal{Q} \rightarrow \mathcal{R}$, the task is to return the k documents with highest values of $R(d_i, \mathcal{Q})$. The simple *frequency* version of the problem assumes that the query consists of a single sequence (term) q and that $R(d_i, q)$ is the frequency $tf(d_i, q)$ of q in d_i. The *tf-idf* version of the problem computes $R(d_i, \mathcal{Q}) = tf(d_i, q) \times \log(N/df(q))$, where $df(q)$ is the *document frequency* of q, the number of documents in which sequence q occurs. Note that the *tf-idf* formulation used here and in similar studies (for example, Sadakane [19]) is still a simplification of the measures used in modern retrieval systems, which incorporate factors like document length, static document components such as pagerank, and queries with multiple terms.

We focus here on the single term *frequency* and single term *tf-idf* versions of the problem. Sadakane [19] devised the first succinct structure for these problems, an approach we refer to as SADA. An alternative mechanism, GREEDY, was presented by Culpepper et al. [2]. Culpepper et al. also describe implementations of SADA and GREEDY, based on components that were available in 2009. Other compressed index representations solving the top-k retrieval problem have been recently proposed. They provide different time and space trade-offs [8, 14], but generally apply more complex storage, compression and query techniques to reduce storage costs or increase query performance. For the purpose of simplicity and highlighting the impact of state-of-the-art components to compose more complex structures, we focus on the two earlier, and comparatively simpler index structures SADA and GREEDY. Insights, techniques and performance gains however are transferable to more recent top-k retrieval techniques which are composed of the same basic succinct structures.

We first briefly explain both index types, and then recompose and reimplement them using the library, to study the impact of state-of-the-art components. For both solutions we use the conventions established by Sadakane and Culpepper et al.: the set of documents is concatenated to form a text $\mathcal{T}$ by appending a sentinel symbol # to each d_i; then joining them all in to one; then appending a further sentinel symbol \$ following d_N. Both sentinels are lexicographically smaller than any symbol in Σ, and $\$ < \#$. We use n to represent $|\mathcal{T}|$.

2.1 SADA and GREEDY revisited

The SADA structure is composed of several components. First, a CSA (denoted `csa_full`) over $\mathcal{T}$ identifies, for any pattern p, the range $[sp..ep]$ of matching suffixes in $\mathcal{T}$, providing the functionality of a SA. Second, a BV is constructed (denoted `border`$[0..(n-1)]$) with `border`$[i] = 1$ iff $\mathcal{T}[i] = \#$; supporting rank and select structures (`border_rank` and `border_select`) are also generated. A *document array* $\mathcal{D}[0..(n-1)]$ that maps the ith suffix in $\mathcal{T}$ to the document that contains it can then be emulated using `border` and `csa_full`, since $\mathcal{D}[i] =$ `border_rank`$(\mathcal{SA}[i])$. Third, to generate all distinct document numbers in a range $[sp..ep]$, an RMQ structure `rminq` is used. It is built over a conceptual array $C[0..(n-1)]$, defined as $C[i] = \max\{j \mid j < i \wedge D[j] = D[i]\}$; that is, $C[i]$ is the index in $\mathcal{D}$ of the last prior occurrence of $\mathcal{D}[i]$. The number of distinct values in $\mathcal{D}[sp..ep]$ corresponds to $df(q)$. Locations of the values are identified by computing $x = \texttt{rminq}(sp, ep)$, the index of the minimum element in $C[sp..ep]$. A temporary BV of size N is used to check if $\mathcal{D}[x]$ was already retrieved; if it wasn't, the counting is continued by recursing into $[sp..(x-1)]$ and $[(x+1)..ep]$. Finally, a second RMQ structure (`rmaxq`) and individual CSAs are built for each document d_i, in order to calculate $\mathit{tf}(d_i, q)$. The top-k items are then calculated by a partial sort on the $\mathit{tf}(d_i, q)$ values. In total, SADA uses $|\text{CSA}(\mathcal{T})| + |\text{BV}(\mathcal{T})| + 2|\text{RMQ}| + \sum_{i=0}^{N-1} |\text{CSA}(d_i)|$ bits. Choosing an H_k-compressed CSA for `csa_full`, a $2n+o(n)$-bit solution for `rminq` and `rmaxq`, and a compressed BV for `border` [15], results in a total bit count bounded above by $nH_k(\mathcal{T})+\sum_{i=0}^{N-1} |\text{CSA}(d_i)| + 4n+o(n)+ N(2+\log(n/N))$. The space for the document CSAs was deliberately not substituted by the cost of a concrete solution, for two reasons: (1) each CSA has an alphabet-dependent overhead, which dominates the space for small documents; and (2) only the inverse SA functionality is used. Thus, in the recomposition, we opt for a *bit-compressed*[3] version of the inverse SA, with almost no overhead per document, and benefiting from constant access time.

The second solution – GREEDY – also uses `csa_full` to translate patterns p to ranges $[sp..ep]$. In this solution the document array $\mathcal{D}$ is explicitly represented by a WT, denoted `wtd`. The total size is then $|\text{CSA}(\mathcal{T})| + |\text{WT}(\mathcal{D})| = nH_k(\mathcal{T}) + nH_0(\mathcal{D})$, using compressed BVs for `wtd` [17]. Culpepper et al. use these structures to solve the frequency variant of the top-k problem. In the first step the pattern is translated to a range in $\mathcal{D}$. A priority queue is then used to store pending nodes in the expansion in WT of the range $[sp..ep]$. At each step the largest node is extracted, and, if it is not a leaf node, its two children are inserted. This process is iterated until k leaves – corresponding to the top-k frequent documents – have emerged. The range size of a leaf corresponds to the term frequency $\mathit{tf}(d_i, q)$. In contrast to SADA, not all documents have to be listed in order to calculate the top-k. However, GREEDY does more work per document, since `wtd` has a height of $\log N$.

2.2 Experimental setup

We adopt the experimental setup employed by Culpepper et al. [2], reusing the PROTEINS file and adding the ENWIKI file[4]. The ENWIKI datafiles come in four versions: parsed as either characters or words; and either a small prefix, or the whole collection, see Table 1. The character version was generated by removing markup from a wikipedia dump file; the word-based version by then applying the parser of the Stanford Natural Language Group.

We generated patterns of different lengths, again following the lead of Culpepper et al. [2], with 200 patterns of each length; reported query times are averages over sets of 200 patterns. All experiments were run on a server equipped with 144 GB of RAM and two Intel Xeon E5640 processors each with a 12 MB L3

[3] That is, each item is encoded as a binary value in $\lceil \log |d_i| \rceil$ bits. [4] We did not use the Wall Street Journal file, since licensing issues meant that we could not make it available for download. Our full setup is available at http://go.unimelb.edu.au/w68n.

cache. The experimental code is available within the benchmark suite of the library, and all used library classes (printed in fixed italic font) are linked to their definitions.

Collection	n	N	n/N	σ	$\lvert\mathcal{T}\rvert$ in MB	$\approx H_k(\mathcal{T})$
character alphabet						
PROTEINS	58,959,815	143,244	412	40	56	0.90
ENWIKI-SML	68,210,334	4,390	15,538	206	65	2.01
ENWIKI-BIG	8,945,231,276	3,903,703	2,291	211	8,535	2.02
word alphabet						
ENWIKI-SML	12,741,343	4,390	2,902	281,577	29	5.03
ENWIKI-BIG	1,690,724,944	3,903,703	433	8,289,354	4,646	4.45

Table 1: Collection statistics: number of characters/words, number of documents, average document length, total collection size, and approximate H_k (determined using `xz -best`) in bits per character/word. The character based collections use one byte per symbol, while $\lceil \log \sigma \rceil$ bits are used the word based case.

2.3 Plug and play

We start by composing an instance of GREEDY. For `csa_full`, SDSL provides several CSA types. We opt for *`csa_wt`*, which is based on a WT. Choosing a Huffman-shaped WT [10] (*`wt_huff`*) and parameterizing it with a suitable BV (*`rrr_vector`*) results in an H_k-compressed CSA. The *`rrr_vector`* implements the on-the-fly decoding recently described by Navarro and Providel [13], which provides low redundancy. Finally, we minimize the space of the CSA by sampling (inverse) SA values only every millionth position. This does not affect the runtime of GREEDY, since it does not require SA access. For `wtd` we choose the alphabet-friendly WT class *`wt_int`* and parameterize it with a fast uncompressed BV (*`bit_vector`*) and small overhead rank structure (*`rank_support_v5`*). No select functionality is required in GREEDY.

We use the same toolbox to assemble a space- and time-efficient version of SADA. The full CSA in SADA has to provide fast element access; "plug-and-play" exploration with different CSAs showed that *`csa_sada`* [18] is the preferred choice in this situation. The suffix sampling rate is set to 32. Similar exploration revealed that a text-order sampling strategy yields a more attractive time-space trade-off than suffix-order sampling, provided the Elias-Fano compressed BV [15] (*`sd_vector`*) is used to mark the sampled suffixes. For components `rminq` and `rmaxq` we select the range min-max-tree based RMQ structure (*`rmq_succinct_sct`*). The inverse SAs of the documents are represented using bit-compressed vectors (*`int_vector`*), and the array of vectors is denoted by `doc_isa`.

Finally, we compose word alphabet versions of GREEDY and SADA by replacing the character alphabet strategy (***byte_alphabet***) in the definition of `csa_full` by the word alphabet strategy (***int_alphabet***).

Collection	character alphabet		word alphabet	
	GREEDY	SADA	GREEDY	SADA
PROTEINS	162 (2.87)	136 (2.42)	– *no word parsing* –	
ENWIKI-SML	130 (2.01)	204 (3.13)	38 (1.32)	50 (1.72)
ENWIKI-BIG	27,043 (3.17)	24,404 (2.86)	6,786 (1.46)	5,703 (1.23)

Table 2: Sizes of indexes, in MB and as a multiple of the collection.

Memory Usage. Table 2 summarizes the space usage of the composed structures. They take considerably less space than those reported by Culpepper et al.; for example, on PROTEINS, their SADA is 6.4 times larger than ours, and their GREEDY is 1.3 times larger. Also note that when the documents are short, our SADA is smaller than GREEDY. These space reductions result from the use of better-engineered components; the only algorithmic change made was the use of bit-compressed inverse SAs, instead of high-overhead CSAs.

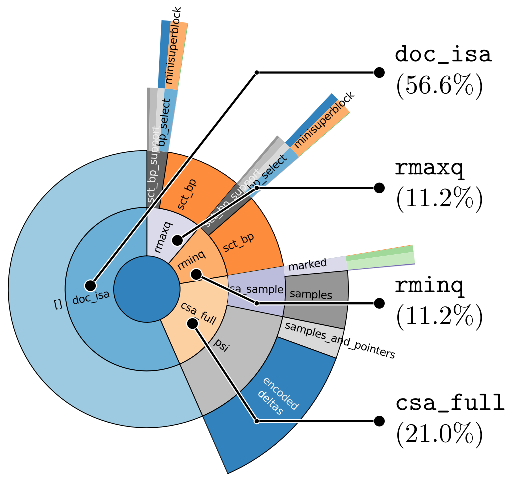

Fig. 1: Sunburst visualization of the memory usage of the character-based SADA on file ENWIKI-BIG. A dynamic version is available at http://go.unimelb.edu.au/ba8n.

Figure 1 depicts a space visualization of the type that can be generated for any SDSL object. It reveals that `doc_isa` takes over half the space; that `rmaxq` and `rminq` are close to the optimal $2n$ bits ($2.6n$ bits); and that the CSA takes 5.3 bits per character. The latter differs from the optimal $H_k = 2.02$ bits reported in Table 2, as samples were added for fast SA access, which account for 2.4 of the 5.3 bits. The largest component of GREEDY (not shown here) is the document array $\mathcal{D}$, which requires $n \log N$ bits plus the overhead of the rank structure, around 92.2% of the total space for ENWIKI-BIG. Plugging in ***rrr_vector*** results in H_0-compression of `wtd` and reduces the overall size to 25,320 MB, while the query time is slowed down by a factor of between 2 and 4. Note that the word versions are smaller than the character-based ones, because n is smaller. SADA

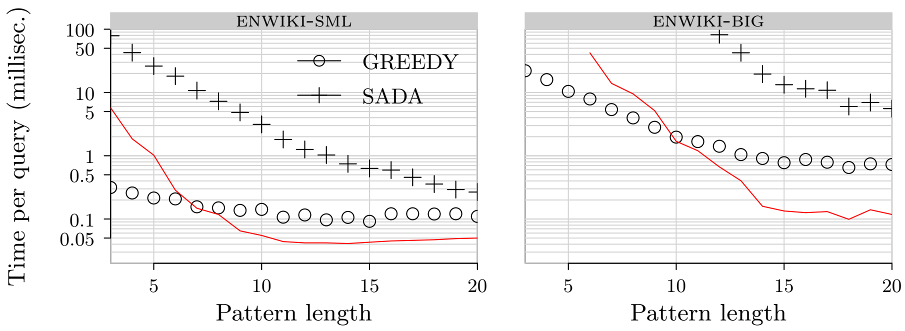

Fig. 2: Average time for top-10 *frequency* queries on the character-based collections. Results are omitted for pattern length ℓ if any single pattern of length ℓ took more than 5 seconds. The solid red line corresponds to SORT, a simple alternative implementation which is explained in the text.

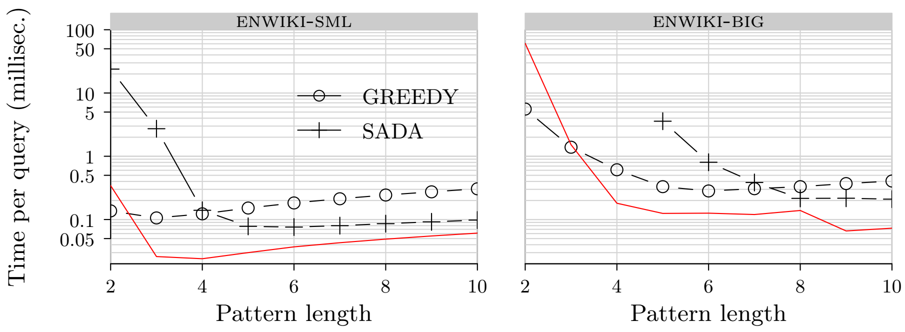

Fig. 3: Average time for top-10 *frequency* queries on the word-based collections.

also benefits from smaller average document lengths. Overall the word indexes are smaller than the character-based indexes, and at around 2/3 of the original text size, are comparable to compressed positional inverted files.

Runtime. Query times are depicted in Figures 2 and 3. We make several observations. First, the runtime of both solutions depends on the collection size, since any given pattern occurs more often in a larger collection. This results in larger ranges, which are especially bad for SADA, since it processes all distinct documents in the range, even when computing top-10 queries. (This same behavior means that SADA can compute complex *tf-idf* queries in very similar times.) GREEDY is also dependent on size of the $[sp..ep]$ range, requiring two orders of magnitudes more time on ENWIKI-BIG than on ENWIKI-SML, matching the difference in their sizes. Second, for long patterns (>15 characters) SADA is now only one order of magnitude slower than GREEDY, in contrast to two orders reported by Culpepper et al. [2]. This is due to the faster extraction of inverse SA values and the use of a Ψ-based CSA instead of a WT-based one. For word indexes

SADA now outperforms GREEDY for long queries, where the result range is very small. In this cases, the pattern matching becomes the dominating cost. In SADA, we used `csa_sada` which implements backward search by binary searches on Ψ using $\mathcal{O}(m \log n)$ time, while `csa_wt` in GREEDY performs $\mathcal{O}(mH_0(\mathcal{T}))$ non-local rank operations on the compressed BV. The GREEDY approach could be made significantly faster if the `rrr_vector` was replaced by an uncompressed BV, but the space would then become much greater than SADA.

We also compare our results to a simple baseline called SORT. Again, `csa_wt` is used as the CSA, but now the document array $\mathcal{D}$ is stored as a bit-compressed vector of $n\lceil \log N \rceil$ bits. After identifying the $[sp..ep]$ interval, the entries of $\mathcal{D}[sp..ep]$ are copied and sorted to generate (d_i, tf) pairs; the standard C++ partial sort is then used to retrieve the top-10. The red lines in Figures 2 and 3 show query times. The sequential or local processing of SORT is always superior to SADA, which, despite careful implementation, suffers from the non-locality of the range minimum/maximum queries. Moreover, GREEDY only dominates SORT in cases involving very wide intervals, emphasizing one of the key messages of this article – that it is only when careful implementations and large test instances are compared that the usefulness of advanced succinct data structures can be properly demonstrated..

We are aware that there are more recent proposals for top-k retrieval systems, such as the scheme of Hon et al. [7], which uses precomputed answers for selected ranges. The compressed suffix tree facility of the library enabled implementation of this solution in other experimentation [3, 16]. The recent mechanism by Navarro and Nekrich [12] provides range-independent query performance; and the character-based implementation described by Konow and Navarro [8] would likely outperform SORT and GREEDY for small interval sizes. The presentation of these two schemas is beyond the scope of this article.

3 Efficient Construction of Complex Structures

Constructing SDSs over small data sets – up to hundreds of megabytes – is not a challenge from an engineering perspective, since commodity hardware supports memory-space many times larger than this. However, SDSs are explicitly intended to replace traditional data structures in resource-constrained environments; which means they are most applicable when the data is too large for uncompressed structures to be used, and hence that construction is also a critical issue. As well, complex structures composed of multiple sub-structures often contain dependencies between sub-structures which further complicate the construction process. For example, to construct a CST, a CSA is required; and to construct a CSA quickly, usually an uncompressed SA is needed. Under memory constraints, it is not possible to hold all of these structures in RAM concurrently. To alleviate this problem the library incorporates semi-external construction algorithms which stream data sequentially to and from disk. To facilitate this, the library provides serialization and save/load functionality for all substructures.

Finally, as already noted, the library includes memory visualization techniques, which analyze space utilization during run-time and construction.

To demonstrate the complexities of the construction process in more detail, we examine the resource utilization during the construction of word-based SADA. Figure 4 shows the resource consumption for the 4.6 GB ENWIKI-BIG collection. In total, the construction process took 5,250 seconds and had a peak requirement of 13 GB. This corresponds to a throughput of 0.88 MB per second. In monetary terms, the SADA index for ENWIKI-BIG can be built for less than one dollar on the Amazon Cloud[5]. The majority of the time (65%) was spent constructing the CSA. First, the plain SA is constructed – phase 1 in Figure 4 – using the algorithm of Larsson and Sadakane [9]. The algorithm uses twice the memory space required by the resulting bit-compressed suffix array, accounting for the peak memory usage (13 GB) of the complete process. After construction, the SA is serialized to disk to construct the Burrows-Wheeler Transform (BWT) (phase 2). Only $\mathcal{T}$ is kept in memory, as it is the only place where random access is required, and the BWT sequence can be written to disk as it is formed. This semi-external construction process of the BWT requires 380 seconds, or 7% of the total time. The remainder of the CSA is constructed in 358 seconds. This includes constructing the Ψ mapping (shown as phase 3), and sampling the SA. The next major construction step, marked as phase 4 in the figure, constructs the N individual inverse SAs (`doc_isa`). Here for each document an SA is constructed, inverted, bit-compressed, and added to `doc_isa`. This process requires 785 seconds, or around 200 microseconds per document. Next, phase 5 constructs the document array $\mathcal{D}$ by streaming the full SA from disk and performing n RANK operations on `border`. Creating the complete array requires 271 seconds or 160 nanoseconds per $\mathcal{D}[i]$ value. This includes reading the SA from disk, performing the RANK on `border`, and storing the resulting $\mathcal{D}[i]$ value. Finally, `rminq` (phase 6) and `rmaxq` (phase 7) are created, including computing temporary C and C' arrays from $\mathcal{D}$. Creating `rminq` requires 360 seconds; computing `rmaxq` a further 396 seconds. Half of that time is spent creating C and C'; the balance on constructing the actual RMQ structures.

Semi-external construction is an important tool to minimize the resource consumption in each phase of the building process of SADA. Keeping all constructed components in memory would significantly increase the memory overhead during the later stages of the construction process. Thus, careful engineering of the construction phase of each individual component of a complex structure is important so that the overall resource requirements of the entire construction process is minimized. In our case, SADA is now so efficient that we can build the structure for the word tokenized GOV2 collection, used in the TREC Terabyte Track, on our experimental machine.

It is important to monitor construction costs, as the modularity of SDSs can lead to unintended resource consumption. For example, in an initial version of our SADA implementation, the CSA was not serialized to disk, but kept in memory. Visualization of the overall construction cost of SADA highlighted

[5] In November 2013 a "High-Memory Extra Large" instance with 17 GB RAM costs USD$0.41 per hour.

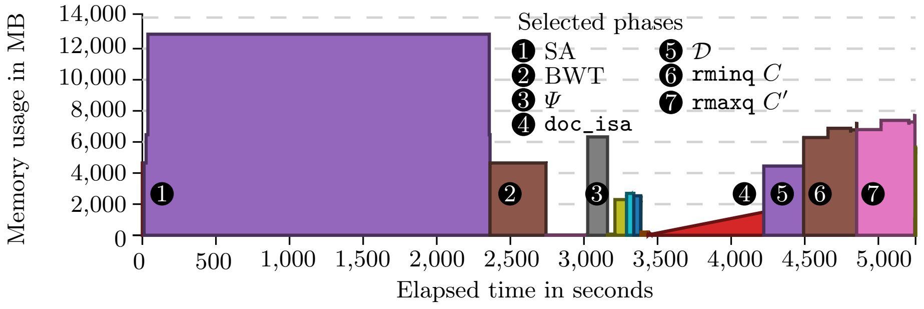

Fig. 4: A memory-time graph for construction of SADA over the word-based sequence ENWIKI-BIG (4.6 GB). The version generated by ***memory_monitor*** is available at http://go.unimelb.edu.au/7a8n.

this inefficiency and allowed it to be rectified. Similarly, when reviewing the character-based ***csa_wt***, it became apparent that optimizing WT construction could not significantly improve the overall process, as that phase only accounts for around 4% of the total cost. An automatic resource tracker (***memory_monitor***) facilitates visualizing the space and time consumption of algorithms in order to provide such insights.

Operations on SDSs can be accelerated if 1 GB pages (hugepages) are used to address memory, rather than the standard 4 kB pages, since address translation becomes a bottleneck when pages are small [5]. The SDSL now includes memory management facilities that allow the use of hugepages during construction as well. We investigate the effect of hugepages on construction time by building ***csa_wt***s of increasing size for prefixes of the character-based ENWIKI-BIG collection (Table 3). For small file sizes, hugepages have a modest effect on construction time, and the 100 MB file is processed only 16% faster. As the file size increases the effect becomes more visible. The index of 1 GB prefix is constructed 24% faster, and the 5 GB index can be built 35% faster. The improvement in construction time then decreases for the full ENWIKI-BIG collection, as the TLB can only maintain a certain number of hugepage entries, after which address translation becomes more costly again.

Prefix	Page size	
(MB)	4 kB	1 GB
10	2	2
100	29	25
1,000	379	307
5,000	2,524	1,877
8,535	5,282	4,482

Table 3: Construction times (seconds) of ***csa_wt*** for prefixes of the character-based ENWIKI-BIG file.

4 Related Work

As part of their experimental work authors often make prototype implementations of their proposed structures available. Additionally, several experimental studies and publicly available libraries focusing on SDSs have emerged. The best-known is the Pizza&Chili corpus[6], which was released alongside an extensive empirical evaluation [4]. The corpus includes a collection of reference data sets, plus implementations of several succinct text index structures accessed via a common interface. The libcds library is also popular, and provides implementations of bitvectors and wavelet trees [1]. It has recently been subsumed by libcds2[7]. Vigna [20] provides bitvector implementations supporting rank and select in the Sux library[8]. The Java version of Sux also implements minimal (monotone) perfect hash functions. Recently Grossi and Ottaviano [6] presented the succinct library[9] which provides bitvectors, succinct tries and an RMQ structure; all structures can be memory mapped.

Compared to these other implementations, sdsl version 2 has a number of distinctive features: all indexes work on both character and *word inputs*; it is optimized for large-scale input (including *dynamic support for hugepages*); it provides coverage of a wide range of alternative structures (including several CSAs and CSTs), that can be composed and substituted in different ways; and it offers *dynamic visualizations* that allow detailed space evaluations to be undertaken. Finally, fully automated tests, *code coverage*, and *various benchmarks* are also included, and can be used by other researchers in the future to check the correctness and performance of further alternative implementations of the various modules. The italicized facets represent the enhancements in version 2 relative to the previous sdsl release [5].

5 Conclusion

We have explored the benefits that flow when modular and composable implementations of succinct data structure building blocks are available, and have showed that efficiency at all levels of the SDS hierarchy can be enhanced by careful attention to low-level detail, and to the provision of precisely-defined interfaces. As a part of that demonstration, we introduced the open source sdsl library, which contains efficient implementations of many SDSs. The library is structured to facilitate flexible prototyping of new high-level structures, and because it offers a range of options at each level of the data structure hierarchy, allows rapid exploration of implementation alternatives. The library is also robust in terms of scale, handling input sequences of arbitrary length over arbitrary alphabets. In addition, we have demonstrated that the use of hugepages can have a notable effect on construction times of large-scale SDSs; and shown that the advanced visualization features of the library provide important insights into the time and space requirements of SDSs.

Acknowledgment. The authors were supported by the Australian Research Council and by the Deutsche Forschungsgemeinschaft.

Software & Experiments. The library code, test suite, benchmarks, and a tutorial, are publicly available at https://github.com/simongog/sdsl-lite.

[6] http://pizzachili.dcc.uchile.cl [7] https://github.com/fclaude/libcds2
[8] http://sux.di.unimi.it [9] https://github.com/ot/succinct

Bibliography

[1] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. SPIRE*, pages 176–187, 2008.

[2] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. ESA*, pages 194–205, 2010.

[3] J. S. Culpepper, M. Petri, and F. Scholer. Efficient in-memory top-k document retrieval. In *Proc. SIGIR*, pages 225–234, 2012.

[4] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *J. Experimental Alg.*, 13, 2008.

[5] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Soft. Prac. & Exp.*, 2013. http://dx.doi.org/10.1002/spe.2198, (to appear).

[6] R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. SEA*, pages 5–17, 2013.

[7] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.

[8] R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. DCC*, pages 5–17, 2013.

[9] N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theor. Comp. Sc.*, 387(3): 258–272, 2007.

[10] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM*, pages 45–56, 2005.

[11] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comp. Surv.*, 2014. To appear.

[12] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. SODA*, pages 1066–1078, 2012.

[13] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. SEA*, pages 295–306, 2012.

[14] G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proc. SEA*, pages 193–205, 2011.

[15] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, 2007.

[16] M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In *Proc. SIGIR*, pages 555–564, 2011.

[17] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.

[18] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Alg.*, 48(2):294–313, 2003.

[19] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comp. Sys.*, 41(4):589–607, 2007.

[20] S. Vigna. Broadword implementation of rank/select queries. In *Proc. WEA*, pages 154–168, 2008.