Practical Performance of Space Efficient Data Structures for Longest Common Extensions

Patrick Dinklage
Department of Computer Science, Technical University of Dortmund, Germany
patrick.dinklage@tu-dortmund.de

Johannes Fischer
Department of Computer Science, Technical University of Dortmund, Germany
johannes.fischer@cs.tu-dortmund.de

Alexander Herlez
Department of Computer Science, Technical University of Dortmund, Germany
alexander.herlez@tu-dortmund.de

Tomasz Kociumaka
Department of Computer Science, Bar-Ilan Unviersity, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Florian Kurpicz
Department of Computer Science, Technical University of Dortmund, Germany
florian.kurpicz@tu-dortmund.de

Abstract

For a text $T[1, n]$, a Longest Common Extension (LCE) query $\mathrm{lce}_T(i, j)$ asks for the length of the longest common prefix of the suffixes $T[i, n]$ and $T[j, n]$ identified by their starting positions $1 \le i, j \le n$. A classic problem in stringology asks to preprocess a static text $T[1, n]$ over an alphabet of size σ so that LCE queries can be efficiently answered on-line. Since its introduction in the 1980's, this problem has found numerous applications: in suffix sorting, edit distance computation, approximate pattern matching, regularities finding, string mining, and many more. Text-book solutions offer $O(n)$ preprocessing time and $O(1)$ query time, but they employ memory-heavy data structures, such as suffix arrays, in practice several times bigger than the text itself.

Very recently, more space efficient solutions using $O(n \log \sigma)$ bits of total space or even only $O(\log n)$ bits of extra space have been proposed: string synchronizing sets [Kempa and Kociumaka, STOC'19, and Birenzwige et al., SODA'20] and in-place fingerprinting [Prezza, SODA'18]. The goal of this article is to present well-engineered implementations of these new solutions and study their practicality on a commonly agreed text corpus. We show that both perform extremely well in practice, with space consumption of only around 10% of the input size for string synchronizing sets (around 20% for highly repetitive texts), and essentially no extra space for fingerprinting. Interestingly, our experiments also show that both solutions become much faster than naive scanning even for finding common prefixes of moderate length, contradicting a common belief that sophisticated data structures for LCE queries are not competitive with naive approaches [Ilie and Tinta, SPIRE'09].

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases text indexing, longest common prefix, space efficient data structures

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.39

Supplementary Material `https://github.com/herlez/lce-test`

Funding *Patrick Dinklage*: Supported by the DFG SPP 1736 "Algorithms for Big Data".
Tomasz Kociumaka: Supported by ISF grants no. 1278/16 and 1926/19, by a BSF grant no. 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Florian Kurpicz: Supported by the DFG SPP 1736 "Algorithms for Big Data".

Acknowledgements Part of this work was carried out during the Dagstuhl Seminar 19241, "25 Years of the Burrows–Wheeler Transform".

1 Introduction and Related Work

The longest common extension (LCE) problem is formally defined as follows:

Given: A text $T[1, n]$ with n symbols from an alphabet of size σ.

Construct: A data structure that allows subsequent on-line queries of the form

$$\mathrm{lce}_T(i, j) := \max\{\ell \geq 0 \ : \ T[i, i + \ell) = T[j, j + \ell)\}.$$

This is a fundamental problem in stringology, with numerous applications in (sparse) suffix sorting [12,19,24], edit distance computation [35,37], approximate pattern matching [2,22,36], regularities finding [4,5,13], string mining [21], and many more. The problem has a classic solution with $O(1)$-time queries based either on the suffix tree of T combined with lowest common ancestor queries [15,26,37], or using the inverse suffix array of T and the longest common prefix array with a data structure for range minimum queries [18,30]. The latter variant is more space efficient in practice but still several times larger than the text itself.

Recently, two notable methods appeared that also partly address the LCE problem: Prezza's *in-place fingerprinting* data structure [44] using a clever replacement of the original text by well-chosen Karp–Rabin fingerprints [31], and data structures relying on local consistency techniques [33,39,46] to construct so-called *string synchronizing sets* (SSS) [12,32].

Although the LCE problem is also of high practical importance, we are only aware of a few experimental papers on the LCE problem [18,29]. However, these studies are now somewhat dated, and naturally none of them includes the recent theoretical advances on the problem. As a result, practitioners (e.g., in bioinformatics) still use those old (slow and/or memory-heavy) data structures (see, e.g., [43,45,47,50]), thereby limiting themselves to problem sizes much smaller than they could actually handle.

Our Contributions and Outline. The goal of this paper is to engineer and evaluate LCE data structures based on the recent theoretical advances in this field [12,32,44]: in-place fingerprinting and string synchronizing sets. We first describe the theory behind these two approaches in Sect. 3, after having introduced some general background in Sect. 2. Sect. 4 describes further details of the implementations, including a simple but practical data structure for the well-known predecessor/successor queries [49]. Finally, in Sect. 5, we evaluate our implementations on a well-established benchmarking set, showing that

(1) our implementations based on SSS are always among the fastest to answer queries and

(2) in-place fingerprinting is very fast in practice and useful to answer queries that have long results.

Further Related Work. Compressed suffix trees with LCA functionality [20] can also be used for LCE queries, but they are certainly too powerful (and hence too space consuming) for the singular problem considered in this paper; this is confirmed experimentally in Sect. 5.

Time-space tradeoffs for LCE queries were considered by many authors [7,9,10,12,24,48]. Kosolobov [34] showed that the product of extra space (in bits) and LCE query time must be at least $\Omega(n \log n)$ under certain assumptions. Furthermore, there are data structures for LCE queries in compressed [6,8,27,28,42,48] and dynamic texts [1,23,39,42].

2 Preliminaries

2.1 Karp–Rabin Fingerprints

The goal of Karp–Rabin fingerprints [31] is to hash substrings to small integers in order to achieve fast comparisons for equality. To construct Karp–Rabin fingerprints, we choose a random prime number $q = \Theta(n^c)$ for some constant $c > 1$. The Karp–Rabin fingerprint $\phi(x, y)$ of the substring $T[x, y]$ is then defined as

$$\phi(x,y) = \left(\sum_{z=x}^{y} T[z] \cdot \sigma^{y-z} \right) \bmod q \ . \tag{1}$$

Observe that the Karp–Rabin fingerprints of matching substrings are equal, i.e., for every integer $\ell \geq 0$, if $T[x, x+\ell] = T[y, y+\ell]$, then $\phi(x, x+\ell) = \phi(y, y+\ell)$. On the other hand, if two substrings do not match, their fingerprints will be different with high probability. Specifically, if $T[x, x+\ell] \neq T[y, y+\ell]$, then $\mathrm{Prob}[\phi(x, x+\ell) = \phi(y, y+\ell)] = O\big(\frac{\ell \log \sigma}{n^c}\big)$. Thus, by choosing large enough constant $c > 1$, we can control the probability of false positives when comparing fingerprints instead of the underlying substrings.

2.2 Static Successor Data Structures

Let U be a universe of u subsequent integers and let $S[0, n-1]$ be a sequence of n integers from U sorted in ascending order. For any $x \in U$, we call $\mathrm{succ}_S(x) := \min\{y \in S \mid y \geq x\}$ the *successor* of x in S, i.e., the smallest value in S larger than or equal to x. Without loss of generality, we assume that $x > S[0]$ (otherwise, its successor is $S[0]$) and $x \leq S[n-1]$ (otherwise, it has no successor). Furthermore, in the following, we are interested only in the *position* i of the successor such that $S[i] = \mathrm{succ}_S(x)$.

The most straightforward ways to find $\mathrm{succ}_S(x)$ are by doing a linear search in time $O(n)$ or, since S is sorted, by doing a binary search in time $O(\log n)$. Both require $O(\log n)$ bits of extra space for keeping track of the current position or the search interval and pivot, respectively. Alternatively, we can construct a bit vector B_S of size $u' = S[n-1] - S[0] + 1$ bits, in which we set $B_S[x - S[0]] := 1$ if and only if $x \in S$. Enhancing B_S to support constant-time rank queries (see [41, p. 75]) then allows for constant-time successor queries, because $\mathrm{succ}_S(x) = \mathrm{rank}_1(B_S, x - S[0])$. However, this method requires $u' + o(u')$ bits of memory and only works for static sets. Matsuoka et al. [38, Corollary 1] showed how to achieve the same space bounds in a dynamic setting, raising the query time to $O(\log \log u)$.

One can also interpret the numbers from S as binary strings and store them in a trie; some nodes of that trie can then be sampled to speed up the successor search. Such *universe-based sampling* has already been used in the *van Emde Boas tree* [49], which answers successor queries in time $O(\log \log u)$ and requires $O(n \log u)$ bits of memory using hashing. Dementiev et al. [14] described a careful implementation of this idea for 32-bit universes, called *STree*. However, the STree is designed for *dynamic* sequences and contains components to support insertion and deletion into the set, which are not needed in the static case.

3 LCE Data Structures

3.1 In-Place Fingerprinting

Prezza [44] showed how to store the Karp–Rabin fingerprints from Sect. 2.1 succinctly so that the fingerprints *replace* the original text characters, but the original characters can still be recovered.

To facilitate explanation, let us assume a byte-alphabet ($\sigma = 256$) and a computer with word size $w := 64$ bits. We group characters from T into blocks of size $\tau := \Theta(w/\log\sigma)$; in our case, $\tau = 8$. Call the resulting array $B[1, n/\tau]$, with $B[i] := T[(i-1)\tau + 1, i\tau]$ for all $i = 1, \ldots, n/\tau$ (assume for simplicity that τ divides n). Note that τ characters fit into a computer word, and can hence be transferred to/from memory and arithmetically manipulated in constant time. We also choose a random prime q such that $\frac{1}{2}\sigma^{\tau} \leq q < \sigma^{\tau}$. We then compute the fingerprints $\phi(1, i\tau)$ for all $i = 1, \ldots, n/\tau$. (The original description actually computes the values $(\phi(1, i\tau) + \bar{s}) \bmod q$, where $0 \leq \bar{s} < q$ is a random seed, which we omit in the following.) Since $D[i] := \lfloor B[i]/q \rfloor \in \{0, 1\}$, all we need to recover the full contents of block $B[i]$ are the two fingerprints $\phi(1, (i-1)\tau)$ and $\phi(1, i\tau)$, and the value $D[i]$. Prezza then shows how to choose q such that w.h.p. the fingerprints $\phi(1, i\tau)$ all have their most significant bit (MSB) set to 0; thus, the array D can be stored instead of those MSBs. (If this doesn't hold, one can recompute the data structure with a different value of q until it holds.) This also allows us to compute the fingerprints for *arbitrary* substrings of T (not necessarily aligned with the block boundaries), also in $O(1)$ time. Prezza shows that with the above choices of q and τ, the fingerprints are collision free with high probability $1 - n^{-\Omega(1)}$.

The advantage of replacing the original text characters with fingerprints is that arbitrarily long substrings of the text can be tested for equality in constant time, using the fingerprints of those substrings. This can be applied for longest common extension queries as follows: To compute $\ell := \mathrm{lce}_T(i, j)$ for any $1 \leq i, j \leq n$, we do an exponential search by comparing $\phi(i, i + 2^k)$ with $\phi(j, j + 2^k)$ for increasing exponents k until the fingerprints mismatch; the actual position of the first mismatch between $T[i, n]$ and $T[j, n]$ is then found by a further binary search on an interval of size $O(\ell)$. The whole process takes $O(\log \ell)$ time, assuming that we have access to a precomputed table of all necessary powers of σ modulo q, which adds another $O(w \log n)$ bits of space (negligible in practice).

3.2 String Synchronizing Sets

The idea behind a *string synchronizing set* (SSS for short) is to designate a (hopefully small) set of positions from T such that this choice of positions is *locally consistent*, meaning that in sufficiently long matching substrings of T, the chosen positions are the same (relative to the beginnings of these substrings). We use the following simplified definition of SSS that is sufficient for our purposes:

▶ **Definition 1.** *For a positive integer $\tau \leq n/2$, the τ-synchronizing set of T is defined as*

$$S = \{i \in [1, n-2\tau+1] \;:\; \min\{\phi(j, j+\tau-1) \;:\; j \in [i, i+\tau]\} \in \{\phi(i, i+\tau-1), \phi(i+\tau, i+2\tau-1)\}\}.$$

The τ-synchronizing set satisfies the following *consistency property*: for all $i, j \in [1, n-2\tau+1]$, if $T[i, i+2\tau-1] = T[j, j+2\tau-1]$, then $i \in S \iff j \in S$. The original definition [32] also includes a *density property* that is necessary to guarantee a small SSS (of size $O(n/\tau)$) even for highly repetitive parts of T, which we do not consider in this paper, as it would make the data structure and the query procedure significantly more complicated. (Omitting the density property can only make our data structure larger, but it remains correct).

We can use SSS for LCE data structures as follows. We first build a successor data structure on S. Let $n' := |S|$ denote the size of the SSS, and let $s_1 < s_2 < \cdots < s_{n'}$ denote the positions of S in increasing order. Define a new text T' of length n' by setting $T'[i] := T[s_i, s_i + 3\tau - 1]$ for all $1 \leq i \leq n'$. Then, build a data structure for constant-time LCE queries on T', e.g., the RMQ-based solution mentioned in the introduction [18].

In order to answer an $\mathrm{lce}_T(i,j)$ query for arbitrary $1 \leq i,j \leq n$, first compare at most $3\tau+1$ characters from $T[i,i+3\tau]$ and $T[j,j+3\tau]$. If this comparison yields a mismatch, we are done. If not, compute $s_{i'} := \mathrm{succ}_S(i)$, $s_{j'} := \mathrm{succ}_S(j)$, and $\ell' := \mathrm{lce}_{T'}(i',j')$. Then,

$$\mathrm{lce}_T(i,j) = s_{i'+\ell'} - i + \mathrm{lce}_T(s_{i'+\ell'}, s_{j'+\ell'}) \ , \tag{2}$$

and the latter lce_T value can again be computed naively as it is at most 3τ due to $T'[i'+\ell'] \neq T'[j'+\ell']$. Overall, we get $O(\tau)$ query time. (The original description [32] sets $\tau = \Theta(\log_\sigma n)$ and uses the bit-vector approach from Sect. 2.2 for successor queries, as well as word-packing and bit-fiddling techniques, to achieve $O(1)$ query time, still in $O(n \log \sigma)$ bits of space.)

An advantage of this LCE data structure is that it naturally combines fast naive scanning for small LCE values (up to 3τ) with more sophisticated data structures guaranteeing a good worst-case performance. Taken together with the fact that the data structure is easily tunable by adjusting τ, it is an excellent candidate for a practical LCE data structure.

4 Implementation

In this section, we describe our `C++` implementations of the data structures from Sect. 3. The code is optimized for byte-alphabets ($\sigma = 256$) and can handle texts of arbitrary length, with the restriction for the algorithms described in Sect. 4.4 that the synchronizing set S can contain at most $|S| < 2^{32}$ elements.

4.1 A Simple but Fast Static Successor Data Structure

We first describe our implementation of a fast practical data structure for successor queries on a static sorted integer sequence $S[0,n-1]$ of length n over a universe U of size $u = 2^w$. To the best of our knowledge, there is no systematic study of such data structures. Our data structure is a simple index that combines universe-based sampling, binary search, and linear search to find the successor $\mathrm{succ}_S(x)$ for any given $x \in U$.

Let $k < w$ be a trade-off parameter and let c be the number of elements from U that fit into a cache line. We build an index P over S such that we can, in constant time, look up an interval in S of length at most 2^k+1 in which $\mathrm{succ}_S(x)$ is guaranteed to be contained. In this interval, we proceed with a binary search up to the point when the size of the current search interval becomes at most c. In this final small interval, which fully fits into a cache line, we look for $\mathrm{succ}_S(x)$ using linear search. This successor query takes $O(k+c)$ time.

It remains to construct P. Let $\mathrm{hi}_k(x) = \lfloor x/2^k \rfloor$. We define an array I_P of $2^{w-k} = u/2^k$ intervals such that, for every non-negative integer $q < u/2^k$, the entry $I_P[q]$ is defined as

$$[\max\{i \in [0,n-1] \ : \ \mathrm{hi}_k(S[i]) < q\} + 1,\ \max\{j \in [0,n-2] \ : \ \mathrm{hi}_k(S[j]) \leq q\} + 1] \ .$$

If any maximum does not exist, we set the corresponding boundary to 0. As $\mathrm{hi}_k(s) \leq u/2^k$ for each $s \in S$, the right boundary of the last interval is always $n-1$. Informally, we split the universe U into 2^{w-k} intervals of size 2^k, find the corresponding boundaries in S, and store them in I_P with the right boundary shifted forward by one. The successor of x is then guaranteed to be contained in the interval $I_P[\mathrm{hi}_k(x)]$ of S, which is of length at most 2^k+1.

Since, for $1 < q < u/2^k$, the right boundary of $I_P[q-1]$ equals the left boundary of $I_P[q]$, in our final index P, we only store the left boundaries and append $n-1$ (the right boundary of the last interval) at the end. Then, the interval of S in which the successor of x is contained can be expressed as $[P[\mathrm{hi}_k(x)],\ P[\mathrm{hi}_k(x)+1]]$. We can construct P in time $O(n+u/2^k)$ with one scan over S; observe that P takes $(u/2^k+1)\lceil \lg n \rceil$ bits of space.

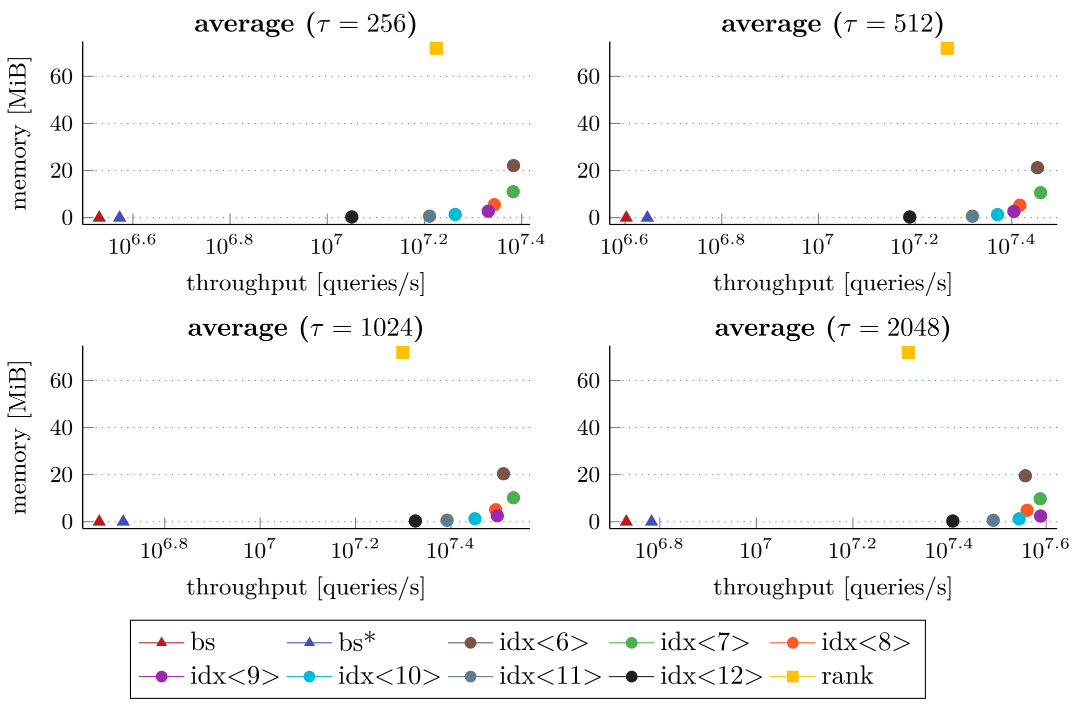


Figure 1 Query throughput versus space usage for static successor data structures built on top of string synchronizing sets with various values of τ, averaged over all texts in the corpus (Tbl. 1).

Note that the space requirement is not optimal in theory and very wasteful if $u \gg n$. However, in our use case of τ-synchronizing sets, the universe is relatively small and gaps between two subsequent integers in S are of length at most τ, making our index practical.

In a preliminary experiment, we looked for the best configuration of the parameter k for our successor data structure. For this experiment, we constructed the string synchronizing sets S for our input texts (see Tbl. 1) and compared the following successor data structures:

1. simple binary search (*bs*),
2. binary search that switches to linear search (*bs**) for intervals of length at most c,
3. the simple data structure from Sect. 2.2, using a bit vector supporting constant-time *rank* queries, and
4. our index P for different parameters k (*idx<k>*).

We measured the memory consumption of the successor data structures and the median query throughput for ten million successor queries over five iterations. The results in Figure 1 indicate that $k := 7$ yields the most beneficial trade-off between query throughput and memory consumption. Therefore, we used this configuration in further experiments. Appendix A shows a detailed breakdown of the results for all input texts.

4.2 Naive LCE Queries

Let us now turn our attention to LCE data structures. An **ultra naive** LCE query algorithm compares characters one by one until a mismatch is detected. This can be sped up significantly by using `uint_128` variables – nowadays supported by most CPUs – to compare blocks of 16 characters at once. Once we have a mismatch, we compare the last block character by character. This algorithm is called **naive**.

4.3 In-Place Fingerprinting

We group 8 consecutive bytes of the input text into a block, reverse the order of the characters in each block by simply swapping the characters (to speed up arithmetic operations when querying the data structure), compute the fingerprint up to the end of the block using Eq. (1), and overwrite the block with this fingerprint. Reversing the order of characters has no significant effect on the construction time, as this construction algorithm is still the fastest on all tested inputs (ignoring the naive approaches that do not require any construction at all), see Sect. 5. We use the smallest prime $q > 2^{63}$, which is $q = 2^{63} + 29$; this worked without any collision detected and with MSB $= 0$ for all stored fingerprints across all tested texts. We also use `uint_128` and the appropriate extended processor instructions for multiplying two 64-bit fingerprints.

Answering LCE queries using the procedure explained in Sect. 2.1 is done block-wise, i.e., in steps by 8, 16, ... characters. Within a block, instead of reconstructing and comparing fingerprints of short substrings, we first translate the fingerprint of the block back to its original contents (characters from T) and scan these characters naively. Because this is faster for LCE values smaller than 256, we also do this at the start of every LCE query. That means that we only start the exponential search after scanning 8 blocks of 8 characters each. We do the same when the final binary search reaches an interval shorter than 256 characters.

4.4 String Synchronizing Sets

While constructing a synchronizing set S, we only need to keep the fingerprints of a window of size 2τ in memory. For reasons of speed, we do so by using a ring buffer of size $2^{\lceil \lg 2\tau \rceil}$: the power of two allows us to compute the positions in the buffer using fast bit operations.

When sorting the suffixes of T', we first use sequential radix sort (`bingmann_msd_CI3_sb`) by Bingmann et al. [11] to reduce the alphabet size of T' (its characters are length-3τ substrings of T). Then, we compute the suffix array using SAIS-LITE [40], because it is the fastest for integer alphabets [3]. According to Sect. 3.2, we now have to compute the LCP array of T'. However, since in the end we are interested only in the LCP values within the original text T, we deviate slightly from Eq. (2) and compute a *sparse* LCP array of T, using the positions from S as indexed positions. For this, we rely on [32, Fact 5.1] that the lexicographic order of suffixes of T' coincides with the order of the corresponding suffixes of T. Apart from eliminating the need to map positions from T' back to T, this also saves us from performing the 3τ naive character comparisons at the end of the query in Eq. (2).

For the actual LCE computation for positions in S, we also build an RMQ data structure on the sparse LCP array; we apply a data structure by Ferrada and Navarro [16] for this purpose. We use a trick to speed up RMQ computations: since we know the length of the interval on which an RMQ is performed, we can make use of the fact that scanning small intervals is faster than an RMQ [29]. In our implementation, we only use the complicated RMQ machinery for intervals larger than 1024; this value yielded the fastest query times. Smaller intervals are simply scanned for the minimum. We implemented two variants of LCE queries:

Prefer-short corresponds to the description in Sect. 3.2: First, scan 3τ characters from T naively; if they are equal, compute the successors, and then apply Eq. (2) (with the modification described in the preceding paragraph). This variant should be used when *small* LCE values are expected, as the procedure is likely to stop already in the initial scan (before computing the successors).

Prefer-long, on the other hand, is optimized for a setting where *large* LCE values are expected. Here, the important observation is that the initial naive scan could reach synchronized positions much earlier than after 3τ comparisons. From that point on, one could immediately resort to LCE computation on T'. We therefore swap the computation of the successors ($s_{i'}$ and $s_{j'}$) and the naive scan. The naive scan then only has to verify that $T[i, s_{i'})$ and $T[j, s_{j'})$ indeed match.

5 Experimental Evaluation

We tested the data structures from Sect. 4: **ultra_naive** and **naive** (Sect. 4.2), **our-rk** (fingerprinting from Sect. 4.3), and **sss**$_\tau$ (string synchronizing sets from Sect. 4.4 with τ = 256, 512, 1024, 2048) in the both variants (*pl* denotes the prefer-long variant). Our implementations are available at `https://github.com/herlez/lce-test`. We also compared our implementations with existing implementations: **prezza-rk** (`https://github.com/nicolaprezza/rk-lce`) and data structures using the compressed suffix trees (CST) **sada** and **sct3** contained in SDSL [25] (`https://github.com/simongog/sdsl-lite`). Other existing implementations were excluded due to their much higher space consumption.

Our experiments were conducted on a computer with two Intel Xeon E5-2640v4 CPUs (each with 10 cores at 2.4 GHz base frequency (3.4 GHz maximum turbo frequency) and cache sizes: 32 KB L1, 256 KB L2, 25.6 MB L3 cache) and 64 GB RAM. All algorithms are sequential and only use a single core, and the results are averages of five runs. The code was compiled with GCC 9.2.0 and compiler flags `-O3`, `-ffast-math`, and `-march=native`.

We ran all algorithms on the input shown in Tbl. 1. The texts were taken from the well-established benchmark suite Pizza&Chili [17]. Data sets from the top half of the table are from the *regular corpus*, whereas those from the bottom half are from the *repetitive corpus*.

Apart from several characteristics of the input texts, like length and alphabet size, Tbl. 1 also shows the sizes of the resulting string synchronizing sets for different values of τ. These numbers confirm our claim from Sect. 3.2 that the additional measures for keeping the SSS small are not necessary in practice, as it can be observed that the growth of SSS size is almost always perfectly proportional to the decrease of the values τ (growth rate only slightly less than 1). The only notable exception is the data set "cere", one of the highly repetitive texts, which contains long runs (of the character `N`). There, for example, when halving τ from 512 to 256, the SSS grows only by a factor of ≈ 1.21 (instead of the expected ≈ 2).

Table 1 Additional information about inputs used in evaluation: name, size n, alphabet size σ, and sizes of the corresponding synchronizing sets $|S_\tau|$ for τ = 256, 512, 1024, and 2048.

name	n	σ	$\|S_{256}\|$	$\|S_{512}\|$	$\|S_{1024}\|$	$\|S_{2048}\|$
dblp.xml	296 135 874	97	2 304 012	1 153 799	578 703	288 758
dna	403 927 746	16	3 141 655	1 575 668	788 113	393 823
english.1024MB	1 073 741 824	239	8 354 560	4 187 042	2 095 466	1 047 924
proteins	1 184 051 855	27	9 215 429	4 616 809	2 341 374	1 155 364
sources	210 866 607	230	1 640 498	821 450	417 457	205 792
cere	461 286 644	5	31 619 034	26 205 215	20 013 847	12 699 848
coreutils	205 281 778	236	1 596 863	800 826	422 577	205 539
Escherichia_Coli	112 689 515	15	876 378	439 647	222 292	109 797
einstein.de.txt	92 758 441	117	721 365	361 616	181 080	90 702
einstein.en.txt	467 626 544	139	3 640 253	1 823 454	998 765	547 184
influenza	154 808 555	15	1 204 011	604 046	301 601	151 142
kernel	257 961 616	160	2 008 714	1 006 209	505 080	252 391

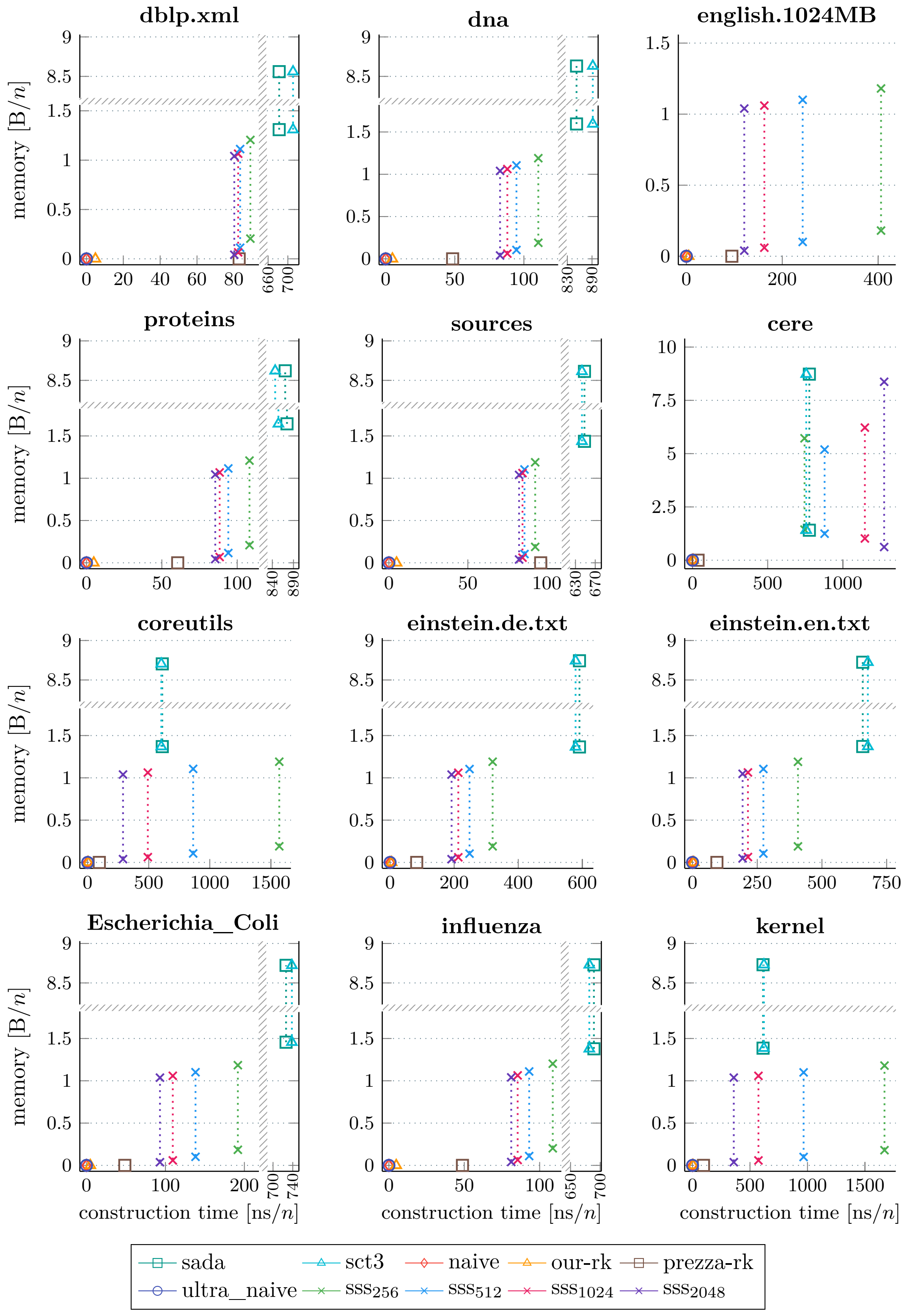


Figure 2 Construction time in nanoseconds and memory consumption in bytes per character of the input for the LCE data structures. The upper mark is the memory peak during construction, and the lower mark is the memory required for the final data structure. (Needs colors for viewing.)

5.1 Construction and Space Usage

Fig. 2 shows the construction times and the space usage of the data structures. For the SSS and CST data structures, that figure shows two numbers: the final size of the data structure (bottom mark) and the memory peak during construction time (top mark). The difference of these two numbers is therefore the additional space at construction time, which results from the intermediate steps using additional data structures (as described in Sect. 4.4 for SSS).

The other data structures need no extra space during construction, as they are either in-place ("prezza-rk" and "our-rk") or do not do any preprocessing at all ("ultra_naive" and "naive"). Also, "our-rk" is significantly faster to build than "prezza-rk" on all inputs; it is 19.76 times faster on "sources". The CST could not be computed for "english.1024MB". Overall, the CST data structures require the most memory. They are also the slowest to construct on all texts but "cere", "coreutils", and "kernel", where SSS is slower for some τ.

The first thing to note is that both time and space requirements grow for SSS with decreasing parameter τ. For space, this is what one would expect immediately, whereas for the running time, this needs further explanation: the times for *sorting* the characters of T', which are substrings of T, depend both on their length 3τ and their number $|S|$. One could now conjecture that, in all cases, roughly the same amount of characters has to be sorted, resulting in construction times mostly independent of τ. However, as our string sorting procedure is *prefix aware* (it only considers the strings up to their distinguishing prefixes), the 3τ-long substrings are usually not inspected in full. This implies that the number of strings can have a higher impact on the running times than their total length, despite the fact that the total length of the strings remains the same (e.g., "english.1024MB"). For other texts (like "dblp.xml"), the construction times are very similar for different values of τ, but are generally faster with growing τ. The notable exception is again the data set "cere", where the order on the time-axis is reversed (but not on the space-axis). This can be explained as follows: we already said before that the growth of the size of the SSS is much less pronounced with decreasing values of τ than for the other data sets. The string sorter is likely to inspect a number of characters proportional to the length of the strings. Therefore, the sorting times for the cere-substrings are influenced more by their total lengths than for the other data sets.

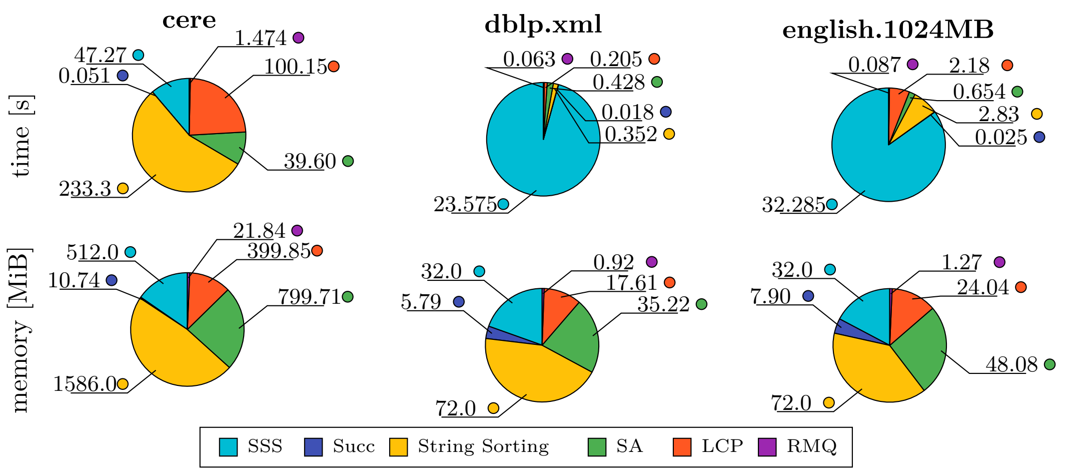


Figure 3 Snapshots of construction time and memory usage during construction for different phases of sss_{512} (same as $\text{sss}^{\text{pl}}_{512}$).

The construction of the SSS structure can be broken down into several steps. Fig. 3 shows examples of how much time is needed for the individual components (building SSS, sorting the length-3τ substrings, building the successor data structure, the suffix array, the LCP array, and the RMQ data structure). Some components differ significantly from text to text: for "cere", the string sorting takes almost half of the total time, whereas for "dblp.xml" (and partly also "english"), the construction of the SSS itself takes almost all of the time. The charts also indicate if and where further engineering efforts can pay off: for example, improving the successor data structure further will most likely neither speed up the final construction times significantly nor improve the space usage. On the contrary, speeding up string sorting or SSS construction (e.g., by parallelization) could result in an overall speedup.

5.2 Query Times

Now, we are interested in query times for different LCE values. To this end, we precomputed queries whose results are in the interval $[2^k, 2^{k+1})$ for all k, allowing us to get a detailed look at query times depending on the LCE value and use the same queries for all data structures.

We now first evaluate the query throughput of the different SSS data structures (with varying values for τ). The results are shown in Fig. 4, grouped by the lengths of the actual LCEs. It can be observed that the initial scanning of characters generally results in a visible drop of the throughput for longer LCEs; smaller values of τ lead to faster query times; the prefer-long variants are indeed faster for longer LCEs, whereas they are slower for shorter ones; and the throughput of all variants does not drop below a threshold for very long LCEs. Indeed, for many data sets, very long LCEs become faster, which can be explained by our method for answering RMQs: for very long LCEs, the corresponding LCP values are likely to be close together in the LCP array, so our query procedure is likely to use the fast scanning for the minimum instead of invoking the heavy $O(1)$-time machinery. Given that the results for $\tau = 512$ and $\tau = 256$ are almost equally fast (in particular for "prefer-long"), we choose the larger of the two values for the following tests, as it results in a smaller data structure.

Fig. 5 shows the query throughput of all data structures (for SSS only with $\tau = 512$). We observe that "ultra_naive" is always slower than "naive" (this comes at no surprise); for short LCEs (roughly up to 2^8), "ultra_naive", "naive", and "sss_{512}" are fastest due to the simple initial scanning; for longer LCEs (greater than 2^{12}), "$\text{sss}^{\text{pl}}_{512}$" becomes the fastest data structure (also faster than "sss_{512}", which becomes faster than the naive approaches for LCEs longer than $\approx 2^{12}$); "our-rk" is slower than Prezza's original implementation "prezza-rk" except for medium sized LCEs around 2^5–2^{10}; and SSS's are much faster than the fingerprinting or naive methods for LCEs longer than $\approx 2^{12}$. The CST data structures "sada" and "sct3" are of similar speed and faster than "ultra_naive" only for long LCEs, around 2^{16} on most inputs. On the repetitive texts "einstein.de.txt" and "einstein.en.txt" both are faster than the fingerprint data structures for medium-size LCEs around $\approx 2^{13}$.

6 Conclusions

We conclude from the experiments that string synchronizing sets (SSS) are the method of choice if their 10–20% overhead on top of the original text size fits into RAM, as they naturally combine the best of two worlds: they answer short LCEs equally fast as naive scanning methods, but are much faster for long LCEs and have a guaranteed worst-case query time. This threshold is much earlier than previously conjectured: even for LCE values larger than $\approx 2^9$, the additional effort for constructing the data structure pays off. If even the little extra space for the SSS is too much and worst case query times have to be guaranteed for long LCEs, then one should use an in-place fingerprinting method.

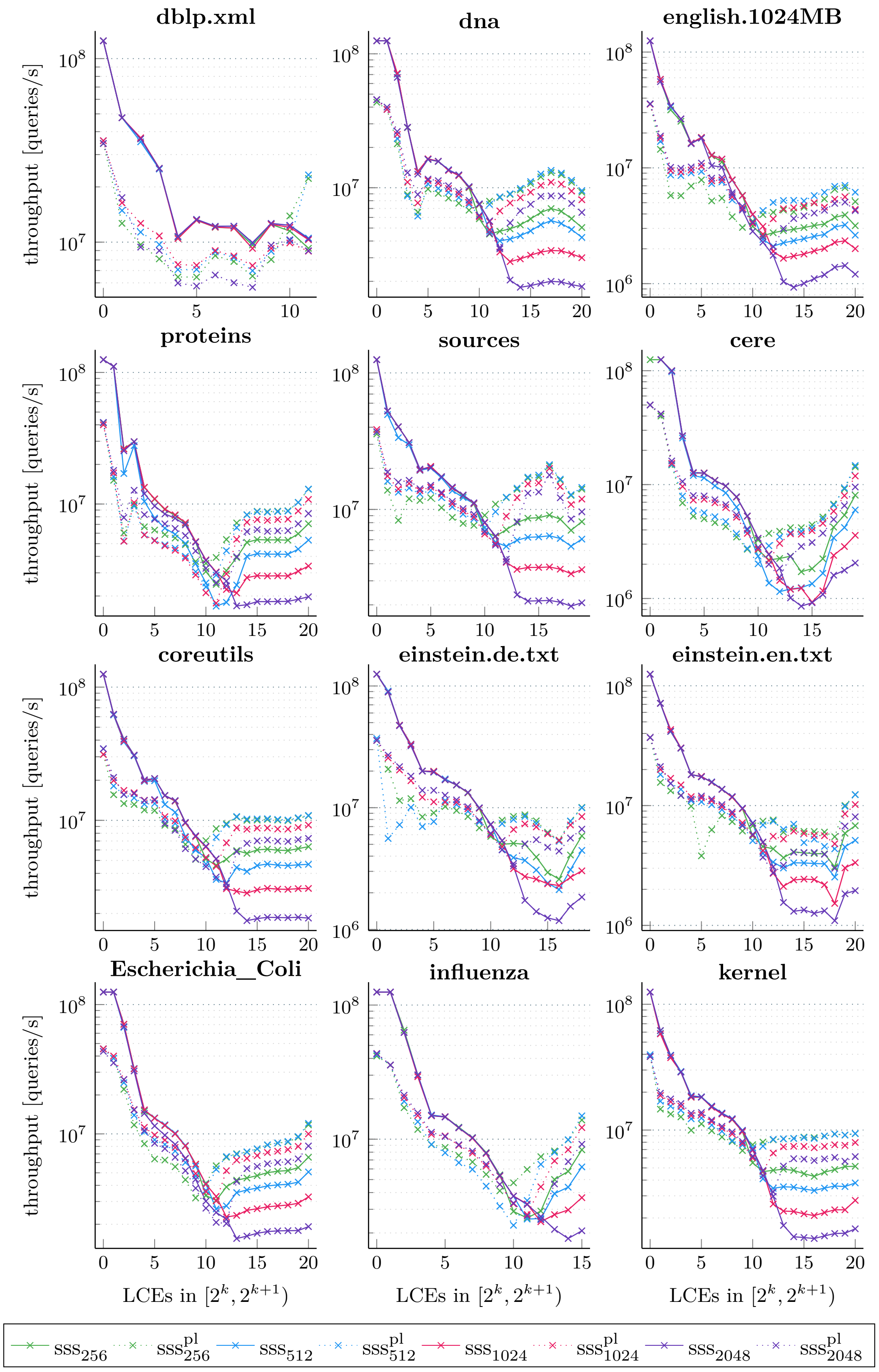


Figure 4 Comparing query throughput of our SSS LCE data structures.

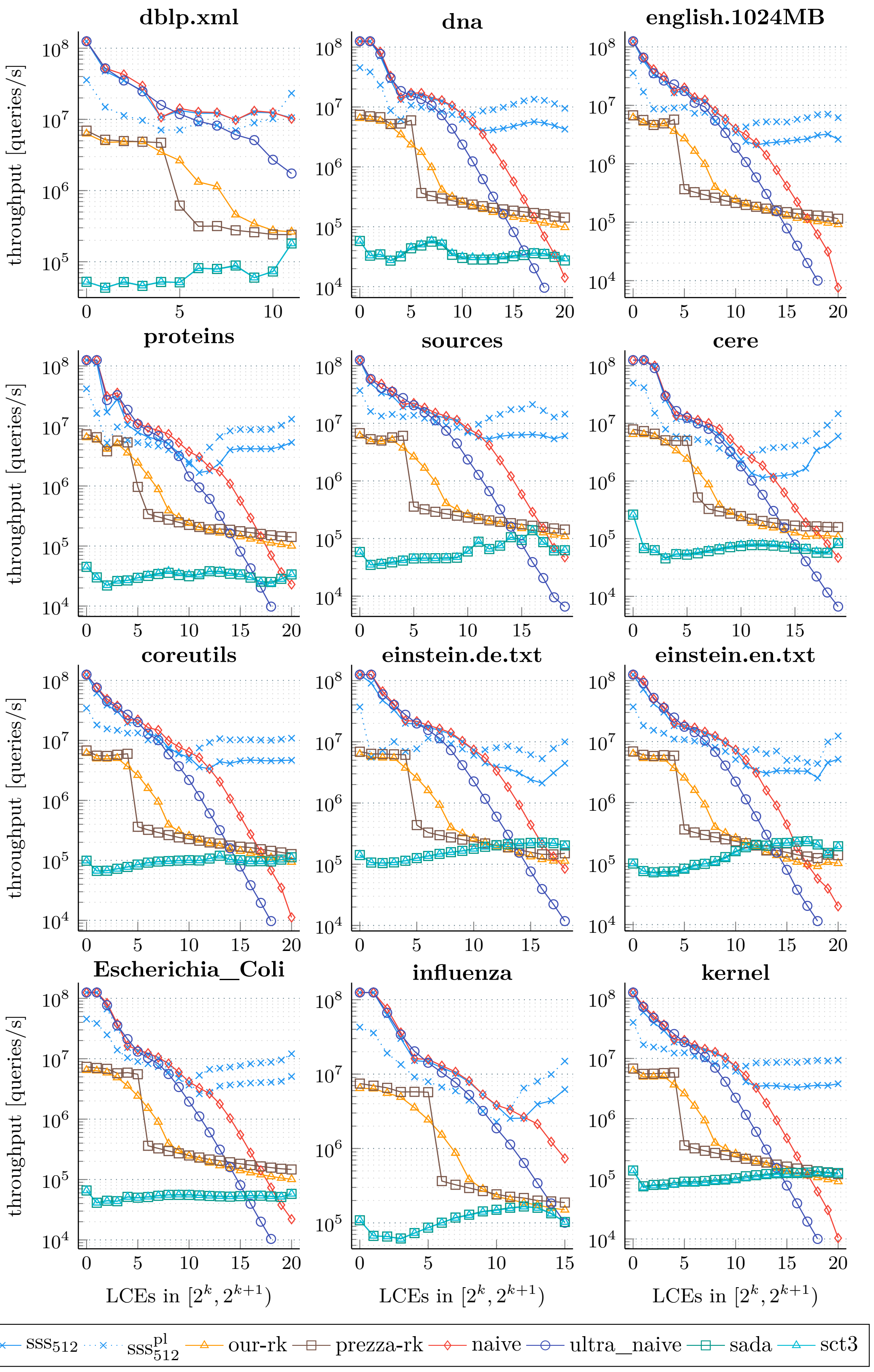


Figure 5 Query throughput of the LCE data structures, only including the fastest SSS ($\tau = 512$).

References

1 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 819–828. ACM/SIAM, 2000. URL: `http://dl.acm.org/citation.cfm?id=338219.338645`.

2 Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004. `doi:10.1016/S0196-6774(03)00097-X`.

3 Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Johannes Fischer, Hermann Foot, Florian Grieskamp, Florian Kurpicz, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. Sacabench: Benchmarking suffix array construction. In *26th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 11811 of *Lecture Notes in Computer Science*, pages 407–416. Springer, 2019. `doi:10.1007/978-3-030-32686-9_29`.

4 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. `doi:10.1137/15M1011032`.

5 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.CPM.2017.22`.

6 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. *Theory of Computing Systems*, 62(8):1715–1735, 2018. `doi:10.1007/s00224-017-9839-9`.

7 Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms*, 12(3):39:1–39:19, 2016. `doi:10.1145/2836166`.

8 Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017. `doi:10.1016/j.jcss.2017.01.002`.

9 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9133 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2015. `doi:10.1007/978-3-319-19929-0_6`.

10 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. `doi:10.1016/j.jda.2013.06.003`.

11 Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *Algorithmica*, 77(1):235–286, 2017. `doi:10.1007/s00453-015-0071-1`.

12 Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 607–626. SIAM, 2020. `doi:10.1137/1.9781611975994.37`.

13 Maxime Crochemore, Roman Kolpakov, and Gregory Kucherov. Optimal bounds for computing α-gapped repeats. *Information and Computation*, 268, 2019. `doi:10.1016/j.ic.2019.104434`.

14 Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit key. In *Sixth Workshop on Algorithm Engineering and Experiments (ALENEX) and the First Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 142–151. SIAM, 2004.

15 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. `doi:10.1145/355541.355547`.

16 Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, 2017. `doi:10.1016/j.jda.2016.09.002`.

17 Paolo Ferragina and Gonzalo Navarro. Pizza&Chili corpus: Compressed indexes and their testbeds. URL: `http://pizzachili.dcc.uchile.cl/`.

18 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In *17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006. `doi:10.1007/11780441_5`.

19 Johannes Fischer, Tomohiro I, and Dominik Köppl. Deterministic sparse suffix sorting on rewritable texts. In *12th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 9644 of *Lecture Notes in Computer Science*, pages 483–496. Springer, 2016. `doi:10.1007/978-3-662-49529-2_36`.

20 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009. `doi:10.1016/j.tcs.2009.09.012`.

21 Johannes Fischer, Veli Mäkinen, and Niko Välimäki. Space efficient string mining under frequency constraints. In *8th IEEE International Conference on Data Mining (ICDM)*, pages 193–202. IEEE Computer Society, 2008. `doi:10.1109/ICDM.2008.32`.

22 Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986. `doi:10.1145/8307.8309`.

23 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1509–1528. SIAM, 2018. `doi:10.1137/1.9781611975031.99`.

24 Paweł Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 425–439. SIAM, 2017. `doi:10.1137/1.9781611974782.27`.

25 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In Joachim Gudmundsson and Jyrki Katajainen, editors, *13th International Symposium on Experimental Algorithms, SEA 2014*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014. `doi:10.1007/978-3-319-07959-2_28`.

26 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. `doi:10.1017/cbo9780511574931`.

27 Tomohiro I. Longest common extensions with recompression. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.CPM.2017.18`.

28 Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Information and Computation*, 240:74–89, 2015. `doi:10.1016/j.ic.2014.09.009`.

29 Lucian Ilie and Liviu Tinta. Practical algorithms for the longest common extension problem. In *16th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5721 of *Lecture Notes in Computer Science*, pages 302–309. Springer, 2009. `doi:10.1007/978-3-642-03784-9_30`.

30 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. `doi:10.1145/1217856.1217858`.

31 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. `doi:10.1147/rd.312.0249`.

32 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 756–767. ACM, 2019. `doi:10.1145/3313276.3316368`.

33 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 532–551. SIAM, 2015. `doi:10.1137/1.9781611973730.36`.

34 Dmitry Kosolobov. Tight lower bounds for the longest common extension problem. *Information Processing Letters*, 125:26–29, 2017. doi:10.1016/j.ipl.2017.05.003.

35 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.

36 Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. doi:10.1016/0304-3975(86)90178-7.

37 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

38 Yoshiaki Matsuoka, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Semi-dynamic compact index for short patterns and succinct van Emde Boas tree. In *26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9133 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2015. doi:10.1007/978-3-319-19929-0_30.

39 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

40 Yuta Mori. sais: An implementation of the induced sorting algorithm. URL: https://sites.google.com/site/yuta256/.

41 Gonzalo Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016. doi:10.1017/cbo9781316588284.

42 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.72.

43 Cinzia Pizzi. Missmax: alignment-free sequence comparison with mismatches through filtering and heuristics. *Algorithms for Molecular Biology*, 11:6, 2016. doi:10.1186/s13015-016-0072-x.

44 Nicola Prezza. In-place sparse suffix sorting. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1496–1508. SIAM, 2018. doi:10.1137/1.9781611975031.98.

45 Wei Quan, Bo Liu, and Yadong Wang. SALT: a fast, memory-efficient and snp-aware short read alignment tool. In *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1774–1779. IEEE, 2019. doi:10.1109/BIBM47256.2019.8983162.

46 Süleyman Cenk Sahinalp and Uzi Vishkin. On a parallel-algorithms method for string matching problems. In *Second Italian Conference on Algorithms and Complexity (CIAC)*, volume 778 of *LNCS*, pages 22–32. Springer, 1994. doi:10.1007/3-540-57811-0_3.

47 Avi Srivastava, Hirak Sarkar, Nitish Gupta, and Robert Patro. Rapmap: a rapid, sensitive and accurate tool for mapping rna-seq reads to transcriptomes. *Bioinformatics*, 32(12):192–200, 2016. doi:10.1093/bioinformatics/btw277.

48 Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 1:1–1:10. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.1.

49 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

50 Chen Zhou, Hao Chi, Leheng Wang, You Li, Yan-Jie Wu, Yan Fu, Ruixiang Sun, and Si-Min He. Speeding up tandem mass spectrometry-based database searching by longest common prefix. *BMC Bioinformatics*, 11:577, 2010. doi:10.1186/1471-2105-11-577.

A Detailed Results for Predecessor Queries

(a) $\tau = 256$.

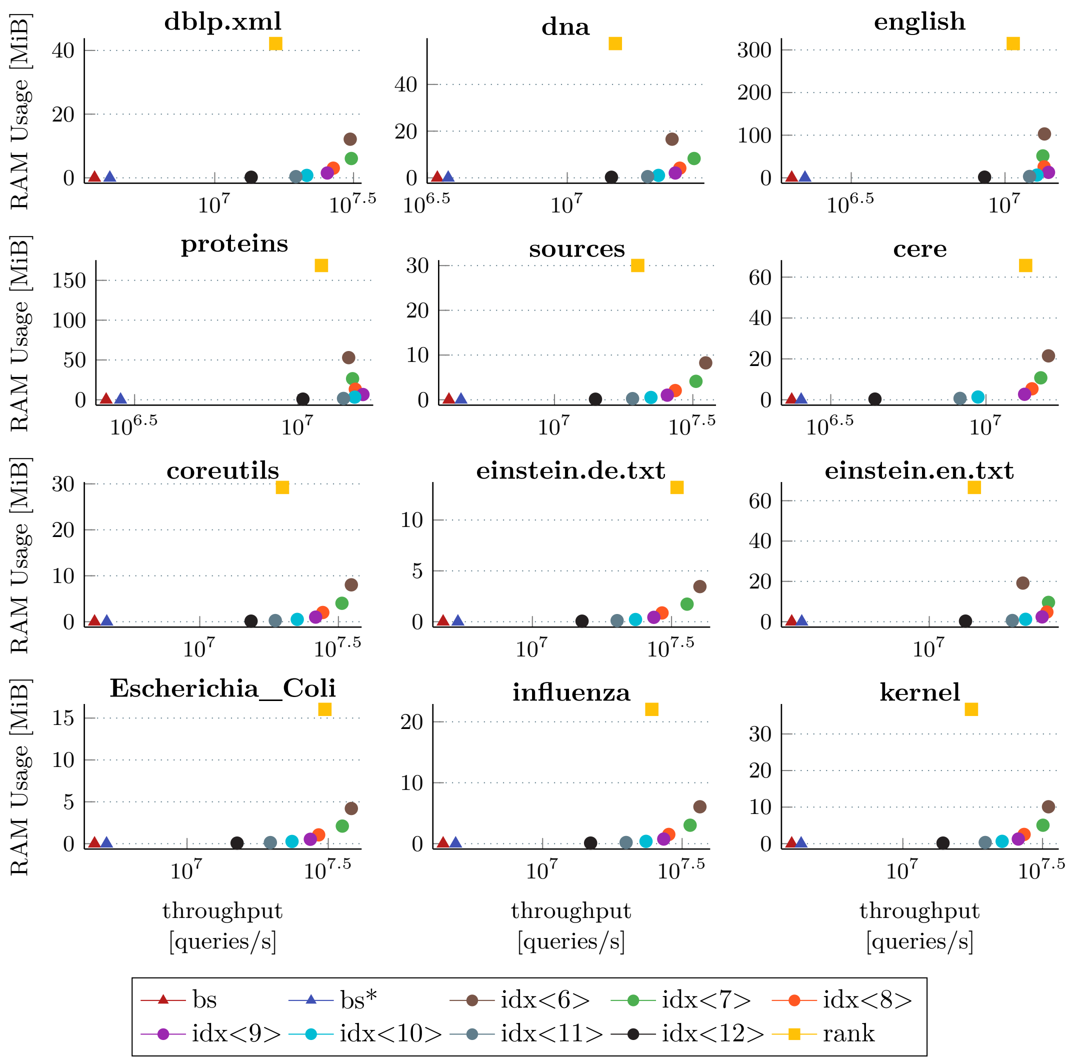
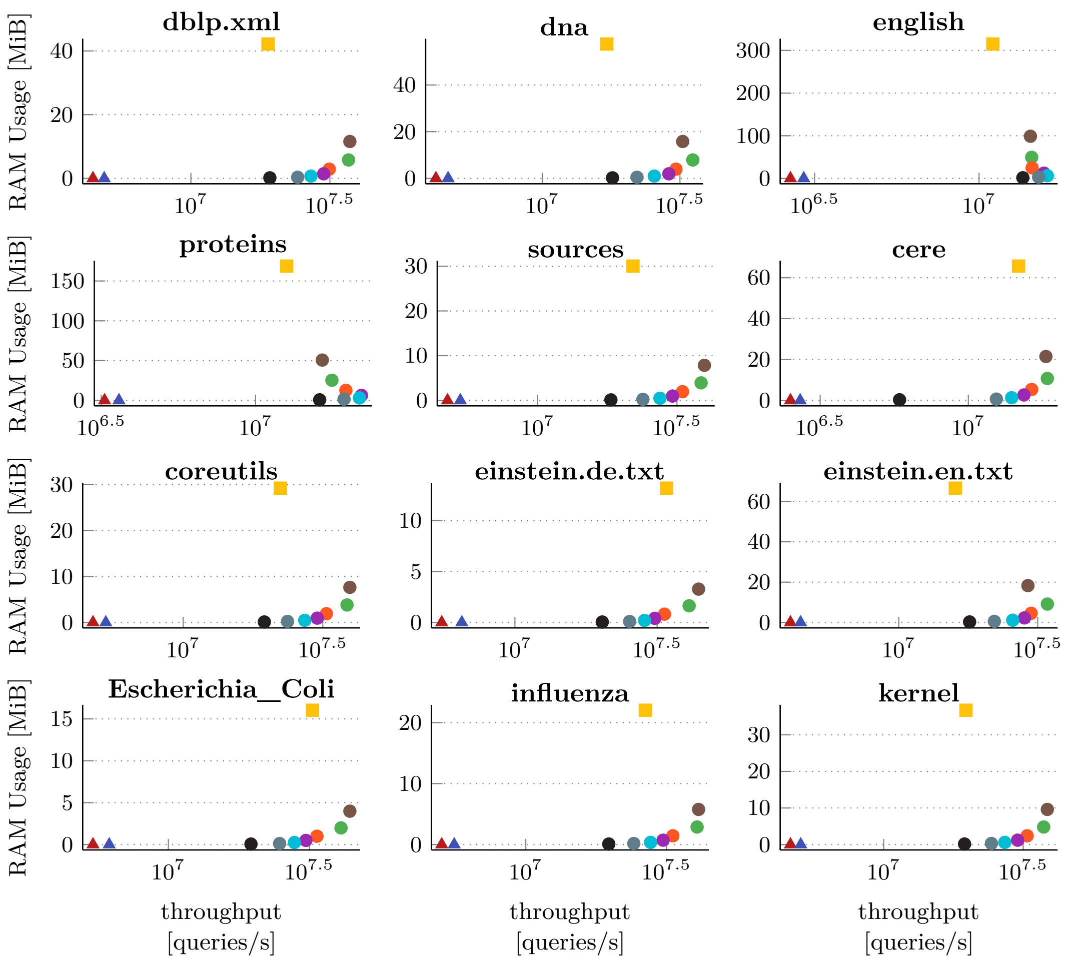


Figure 6 Query throughput and space usage for static successor data structures on SSS's.

(a) $\tau = 512$.



🟨 **Figure 7** Query throughput and space usage for static successor data structures on SSS's (cont.).

(a) $\tau = 1024$.

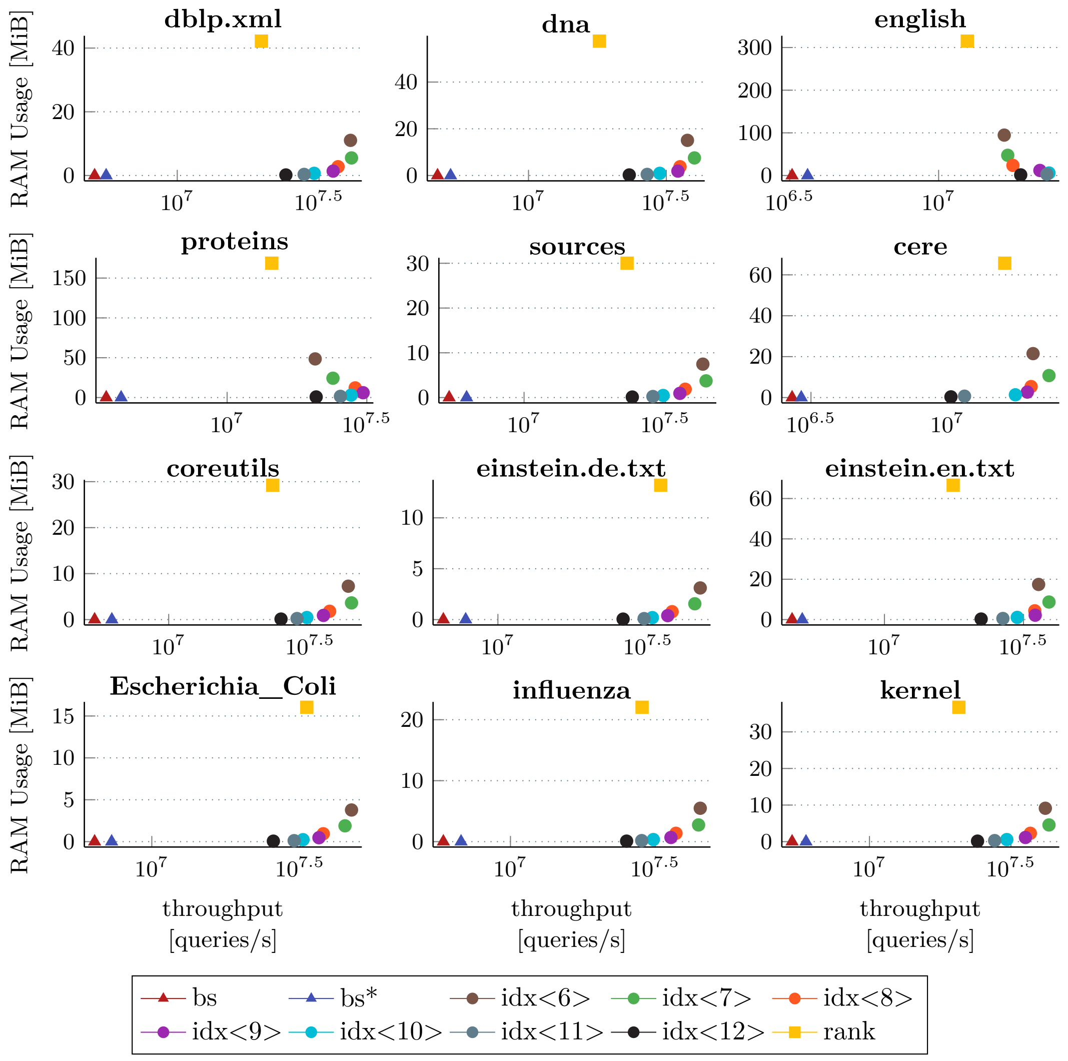


Figure 8 Query throughput and space usage for static successor data structures on SSS's (cont.).

(a) $\tau = 2048$.

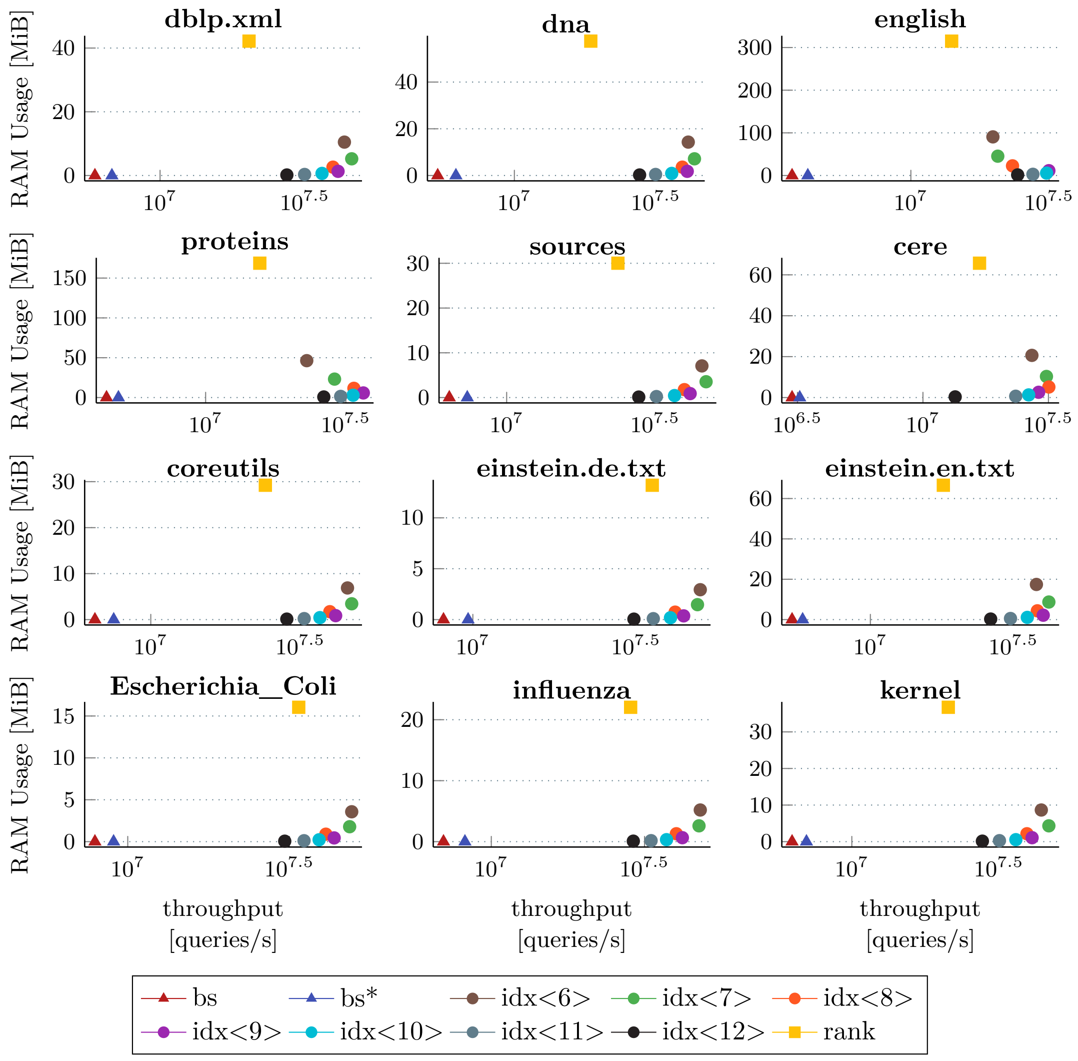


Figure 9 Query throughput and space usage for static successor data structures on SSS's (cont.).