Advantages of Shared Data Structures for Sequences of Balanced Parentheses

Simon Gog
Institut für Theoretische Informatik
Universität Ulm, Germany
simon.gog@uni-ulm.de

Johannes Fischer
Center for Bioinformatics (ZBIT)
Universität Tübingen, Germany
fischer@informatik.uni-tuebingen.de

Abstract

We propose new data structures for navigation in sequences of balanced parentheses, a standard tool for representing compressed trees. The most striking property of our approach is that it shares most of its internal data structures for all operations. This is reflected in a large reduction of space, and also in faster navigation times. We exhibit these advantages on two examples: succinct range minimum queries and compressed suffix trees. Our data structures are incorporated into a ready-to-use C++-library for succinct data structures.

I. Introduction

Sequences of balanced parentheses are at the core of all tree-based compressed data structures [1]–[4]. They are used to describe the topology of an ordered (or binary) tree in a succinct manner, namely using $2n$ bits for an n-node tree [5], [6]. In order to simulate navigation in the represented tree, a lot of effort has been made to support more and more complex operations: subtree-size, depth, parent, and sibling are among the simpler ones; level ancestor queries or the i-th child operation among the more difficult ones. In all cases, $o(n)$ bits on top of the $2n$ from the sequence suffice to carry out the operations in $O(1)$ time [3], [6], [7].

A fundamental operation in many applications is that of computing the *lowest common ancestor* (LCA for short) of two given nodes. Up to now, for constant-time LCAs [2] one has to employ specialized data structures for *range minimum queries* (RMQs) over bit-arrays, similar to the famous LCA-solution in non-succinct trees [8]. The disadvantage of this RMQ-based approach is that is uses entirely new data structures, while the other operations can be implemented to share the most space-consuming parts [7].

We resolve this problem by showing that $O(1)$-LCAs can also be implemented by sharing the space with the other operations. The key idea is a new operation called *range restricted enclose*, whose definition is analogous to operations already present in succinct trees (enclose, findclose, findopen). Our work is quite similar in spirit to a recent development due to Sadakane and Navarro [9], who also noted that there is a need for re-using data structures.

This article is organized as follows. Having introduced the necessary background (Sect. II), in Sect. III we describe our theoretical contributions (solution to range restricted enclose). A large part of our work, however, is devoted to practical applications: in Sect. IV, we show that our new data structure leads to significant improvements (in terms of both time and space) in state-of-the-art schemes for RMQs, attesting again the strong connection between RMQs and LCAs. Finally, in Sect. V, we apply our solution to compressed suffix trees, and show again that we can improve on time and space.

We point out that all our data structures have been implemented and included in the first author's Succinct Data Structure Library (*sdsl*), available at http://www.uni-ulm.de/in/theo/research/sdsl.

II. Definitions and Previous Results

We use the standard *word-RAM* model of computation, where we have a computer with word-width w, where $\log n = O(w)$ for problem size n. Fundamental arithmetic operations on w-bit wide words can be computed in $O(1)$ time.

A. Rank and Select on Binary Strings

Consider a *bit-string* $S[0, n-1]$ of length n. We define the fundamental *rank*- and *select*-operations on S as follows: $\mathit{rank}_1(S, i)$ gives the number of 1's in the prefix $S[0, i-1]$, and $\mathit{select}_1(S, i)$ gives the position of the i'th 1 in S, reading S from left to right ($1 \leq i \leq n$). Operations $\mathit{rank}_0(S, i)$ and $\mathit{select}_0(S, i)$ are defined similarly for 0-bits. There are data structures of size $O(\frac{n \log\log n}{\log n})$ bits in addition to S that support $O(1)$-rank- and select-operations [10]. If the number of ones in S is small, namely $O(n/\log n)$, we can store S in $O(\frac{n \log\log n}{\log n})$ bits of space, *including* structures for $O(1)$-rank- and select-operations [11].

B. Succinct Tree Representations

A string $S[0, 2n-1]$ of n opening parentheses '(' and n closing parentheses ')' is called *balanced* if in each prefix $S[0, i]$, $0 \leq i < 2n$, the number of ')'s is no more than the number of '('s. Such balanced sequences can be used in at least two different ways to represent an ordered tree T on n nodes succinctly in $2n$ bits: BPS ("balanced parentheses sequence") and DFUDS ("depth first unary degree sequence"). The BPS is obtained by a depth-first traversal of T, writing an opening parenthesis '(' when visiting a node for the first time, and a closing parenthesis ')' when all of its subtrees have been traversed. Thus, a node v is represented by a pair of matching parentheses $(\ldots)$. As an example, the tree in Fig. 1 has BPS ((()(()()))()(()())), and the pair of parentheses representing node 2 is underlined.

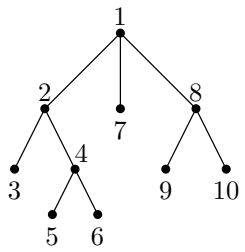

Figure 1. An ordered tree.

For DFUDS, we also visit the nodes in depth-first order, and encode a node with out-degree w as '$(^w)$'. If an initial open parenthesis '(' is prepended, then the resulting sequence is also balanced. For example, the tree in Fig. 1 has DFUDS ((((()(()))(()))))(())), again with node 2 underlined.

C. Navigation in Succinct Trees

In order to navigate in succinctly represented trees, we need additional data structures on top of the $2n$ bits from Sect. II-B. It is known [6] that all basic navigational operations (parent, firstChild, sibling, depth, isLeaf, isAncestor, etc.) can be expressed by the operations from the top half of Tbl. I. We have already seen in Sect. II-A that rank_a and select_a (and hence also *excess*) can be solved with $O(\frac{n \log\log n}{\log n})$ additional bits of space (interpreting a '(' as a '1' and a ')' as a '0'). For operations *find_open*, *find_close*, and *enclose*, Geary et al. [7] give a neat recursive data structure which we are going to describe next using the example of the *find_close*-operation; the solutions to *find_open* and *enclose* are similar.

Let S be the sequence of n balanced parentheses pairs. First, S is subdivided into blocks of fixed size $s = \frac{1}{2}\log n$. Let $\beta(i) = \lfloor i/s \rfloor$ denote the block of i. We say that there are two kinds of parentheses: *near* and *far*. The matching parenthesis $\mu(i)$ of a near parenthesis i lies in the same block as i, $\beta(i) = \beta(\mu(i))$. All other parentheses are called far. To answer *find_close* for near parentheses, we construct a look-up table of size $2^s \times s \times \log s = o(n)$ bits (number of different blocks $\times$ number of query positions $\times$ space for answers). This table can also be used to identify far parentheses, by storing a special value $\perp$ as the answer if there is no near parenthesis. Unfortunately, we cannot store all answers for far parentheses,

Table I
FUNDAMENTAL OPERATIONS ON A SEQUENCE $S[0, 2n-1]$ OF BALANCED PARENTHESES.

operation	description
$\mathit{rank}_a(S, i)$	Returns the number of symbol $a \in \{(,)\}$ in $S[0..i-1]$.
$\mathit{select}_a(S, i)$	Returns the index of the ith $a \in \{(,)\}$ in S (counting starts at 1).
$\mathit{excess}(S, i)$	Returns the number of opening parentheses minus the number of closing parentheses in $S[0, i]$.
$\mathit{find\_close}(S, i)$	If parenthesis i is opening, $\mathit{find\_close}(i)$ returns the position of the matching closing parentheses, defined as $\min\{j \mid j > i \wedge \mathit{excess}(j) = \mathit{excess}(i) - 1\}$.
$\mathit{find\_open}(S, i)$	If parenthesis i is closing, $\mathit{find\_open}(i)$ returns the position of the matching opening parentheses, defined as $\max\{j \mid j < i \wedge \mathit{excess}(j) = \mathit{excess}(i) + 1\}$.
$\mu(S, i)$	Combines the last two operations. If i is opening, $\mu(i) = \mathit{find\_close}(i)$. Otherwise, $\mu(i) = \mathit{find\_open}(i)$.
$\mathit{enclose}(S, i)$	Let i be opening. $\mathit{enclose}(i)$ returns the opening parenthesis of the pair $(j, \mu(j))$ that encloses $(i, \mu(i))$ most tightly, or $\perp$ if no such pair exists.
$\mathit{double\_enclose}(S, i, j)$	Let i and j be opening parentheses of two parentheses pairs $(i, \mu(i))$ and $(j, \mu(j))$ with $\mu(i) < j$. *double_enclose* returns the opening parenthesis of the pair $(k, \mu(k))$ that encloses $(i, \mu(i))$ and $(j, \mu(j))$ most tightly, or $\perp$ if no such pair exists.
$\mathit{rr\_enclose}(S, i, j)$	Let i and j be opening parentheses of two parenthesis pairs $(i, \mu(i))$ and $(j, \mu(j))$ with $\mu(i) < j$. The *range restricted enclose* operation *rr_enclose* returns the opening parenthesis of the pair $(k, \mu(k))$ with $k = \min\{\ell > \mu(i) \mid (\ell, \mu(\ell))$ encloses $(j, \mu(j))\}$, or $\perp$ if no such pair exists.

as there can be $\Theta(n)$ far parentheses in S, and so it would take $\Theta(n \log n)$ bits of space to store all the answers.

So we need a different approach. To this end, let $M_{far} = \{i \mid \beta(i) \neq \beta(\mu(i))\}$ be the set of far parenthesis in S. The subset

$$M_{pio} = \{i \mid i \in M_{far} \wedge \nexists j : (\beta(j) = \beta(i) \wedge \beta(\mu(j)) = \beta(\mu(i)) \wedge \mathit{excess}(j) < \mathit{excess}(i))\}$$

of M_{far} is called the set of *pioneers* of S. It has three nice properties:

(a) The set is symmetric with regard to opening and closing pioneers. That is, if parenthesis i is a pioneer, $\mu(i)$ is also a pioneer.
(b) The sub-sequence S' of S consisting only of the parentheses from M_{pio} is again balanced.
(c) The size of M_{pio} depends on the number of blocks (say b) in S, as there are $\leq 4b-6$ pioneers [5].

Now we show how to answer *find_close* queries for far parentheses. First, we add a *pioneer bitmap* P of length $2n$ to the data structure which indicates whether a parenthesis is in the set M_{pio} or not (see Fig. 2). Because of property (c), P needs $O(\frac{n \log\log n}{\log n})$ bits of space, including structures for rank and select (see Sect. II-A). Now let parenthesis i be opening and far. If it is a pioneer itself, we return $\mu(i)$. Otherwise, by consulting P we determine the maximal pioneer $p < i$, which must be opening. We determine $\mu(p)$, which lies per definition in the same block (say b) as $\mu(i)$. As all excess values in the range $[i, \mu(i) - 1]$ are greater than $\mathit{excess}(i) - 1$, we have to find the minimal position, say k, in block b with $\mathit{excess}(k) = \mathit{excess}(i) - 1$. This is done again by look-up tables.

It remains to show how to determine $\mu(p)$, the position of the matching parenthesis of pioneer p. Instead of storing these values in a plain array (using $O(n/s \times \log n) = O(n)$ bits), we use properties (a) and (b) to get a recursive data structure. Recall that S' denotes the reduced (balanced) sequence of pioneers. It need not be represented explicitly, as $S'[i] = S[\mathit{select}_1(P, i+1)]$. Because S' is a subsequence of S, it holds that the answers to μ in S' are the same as in the original S. Hence, we can use the data structure from the previous paragraph to answer $\mu(\cdot)$ recursively.

There are two options to end the recursion. First, we can stop on a fixed level $k \geq 2$ and store the answers explicitly on level k, in which case one can show that the total space is $o(n)$ bits. The second possibility is to end the recursion when $n_k \leq s$, where n_i denotes the number of parentheses on level i,

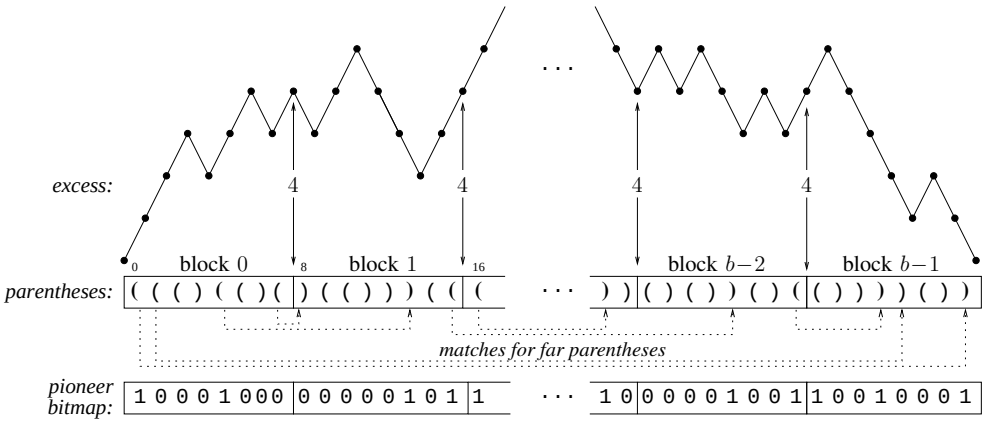

Figure 2. Illustration to Geary et al.'s succinct tree representation. The pioneer bitmap S is stored only implicitly using $o(n)$ bits, and the bottommost array (storing block numbers) is represented recursively by the same data structure.

and then use table lookups to find the answers on level k. Because $n_k = 4^k(\frac{2n}{s^k} - 2) + 2$ and $s = O(\log n)$ for *all* levels, we get about $k = \frac{\log n}{\log\log n}$ levels, and the *find_close*-operation takes $O(\frac{\log n}{\log\log n})$ time. This latter variant is particularly easy to implement.

III. Range Restricted Enclose

The main advantage of the recursive approach from Sect. II-C over previous solutions [6] is that the most space consuming part, the pioneer bitmaps, is shared by *find_close*, *find_open*, and *enclose*. However, these functions cannot be used for solving the important LCA-operation. It seems that it had already been tried to cope with LCA by means similar to the solution of *find_open*, as Munro and Raman introduced the *double_enclose*-operation in the conference version [12] of their work (see bottom half of Tbl. I). However, this approach contained a bug, and was therefore dropped in the final version [6].

```
min_excess_pos(S, ℓ, r)
01  if ℓ ≥ r
02    return ⊥
03  k ← ⊥
04  if β(ℓ) = β(r)
05    k ← near_min_excess_pos(S, ℓ, r)  ①
06  else
07    min_ex ← excess(r) + 2 · (S[r] = ')')  ②
08    p'_ℓ ← rank_1(P, ℓ)
09    p'_r ← rank_1(P, r)                    ③
10    k' ← min_excess_pos(S', p'_ℓ, p'_r)  ④
11    if k' ≠ ⊥
12      k ← select_1(P, k' + 1)  ④'
13      min_ex ← excess(k)
14    else
15      k' ← near_min_excess_pos(S, β(r)·s, r)
16      if k' ≠ ⊥                              ⑤
17        k ← k'
18        min_ex ← excess(k)
19    k' ← near_min_excess_pos(S, ℓ, (β(ℓ)+1)·s)
20    if k' ≠ ⊥ and excess(k') < min_ex       ⑥
21      k ← k'
22  return k
```

Figure 3. The *min_excess_pos*-operation.

It was Sadakane [2] who quietly introduced the first $o(n)$-bit solution for LCAs in BPS-encoded trees. His approach is quite different, as it is based on solving RMQs on the excess-values of the BPS. Because the sequence of excess-values has the property that subsequent elements differ by only 1, this is in fact a restricted version of RMQ, called ± 1RMQ. Sadakane's data structure for ± 1RMQ needs $O(\frac{n \log^2 \log n}{\log n})$ bits, which was recently lowered to $O(\frac{n \log\log n}{\log n})$ bits [13]. A similar RMQ-based approach for LCA works also for DFUDS [3]. However, all of these

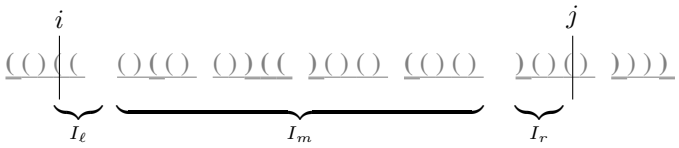

Figure 4. Splitting the interval $[i, j-1]$ into three subintervals.

solutions have the disadvantage that they do *not* share any component of the pioneer-structure from Sect. II-C, and thus constitute a large practical overhead in space.

In this section, we show that the addition of a new operation, called *rr_enclose* for *range restricted enclose*, solves the above-mentioned problems. Method *rr_enclose* takes two opening parentheses i, j ($i < j$) and calculates the leftmost opening parentheses $k \in [\mu(i)+1, j-1]$ such that $(k, \mu(k))$ encloses $(j, \mu(j))$, or $\perp$ if no such parentheses pair exists. We get a nice recursive formulation of the method if we change the parameters slightly: Method *min_excess_pos* takes two (not necessarily opening) parentheses ℓ and r ($\ell < r$), and calculates the leftmost opening parenthesis $k \in [\ell, r-1]$ with $\mu(k) \geq r$. If no $k \in [\ell, r-1]$ exists, $\perp$ is returned. See Fig. 3 for the code of this method, which is invoked by *min_excess_pos*$(S,$ *find_close*$(i)+1, j)$ to solve *rr_enclose*(S, i, j).

To prove the correctness of the algorithm, we show that *min_excess_pos*(S, ℓ, r) returns the leftmost parenthesis $k \in [\ell, r-1]$ with $\mu(k) \geq r$, or $\perp$ if no such k exists. We will prove this by induction on n. If $n \leq s$, ℓ and r lie in the same block and the subroutine *near_min_excess_pos* will find the correct answer by making a look-up in precomputed tables of size $O(s^2 \cdot 2^s \cdot \log s) = O(\sqrt{n}\,\mathrm{polylog}\, n)$ bits (step ①). Otherwise, $n > s$ and $\beta(\ell) \neq \beta(r)$. Let $\hat{k}$ be the correct solution of *min_excess_pos*(S, ℓ, r). We divide the interval $I = [\ell, r-1]$ into three subintervals $I_\ell = [\ell, (\beta(\ell)+1)\cdot s - 1]$, $I_m = [(\beta(\ell)+1)\cdot s, \beta(r)\cdot s - 1]$, and $I_r = [\beta(r)\cdot s, r-1]$; see Fig. 4.

We will show that if $\hat{k}$ lies in I_m, it follows that $\hat{k}$ is a pioneer. To prove that, for the sake of contradiction suppose that $\hat{k}$ is *not* a pioneer. As $\hat{k} \in I_m$ and $\mu(\hat{k})$ does not belong to I_m, it follows that $\hat{k}$ has to be a far parenthesis. Since $\hat{k}$ is a far parenthesis but no pioneer, there has to be an pioneer $\tilde{k} < \hat{k}$ in the same block. Consequently, $(\tilde{k}, \mu(\tilde{k}))$ encloses $(\hat{k}, \mu(\hat{k}))$. This is a contradiction to the assumption that $\hat{k}$ is the right solution.

By induction, we know that if $k' \neq \perp$ after step ④, the corresponding pioneer k in S (calculated by a select query in step ④') lies in I, and $\mu(k) \geq r$.

The look-up in the interval I_r (see step ⑤) for a solution is unnecessary if a pioneer k is found in step ④'. To show this, let $k_r \in I_r$ be the solution from step ⑤. If k_r is not a pioneer, $(k_r, \mu(k_r))$ is enclosed by $(k, \mu(k))$; otherwise it was already considered in step ④. Finally, we have to check the interval I_ℓ for the solution $\hat{k}$, because if $\hat{k}$ lies in I_ℓ it does not necessarily follow that $\hat{k}$ is a pioneer. This is due to the fact that there can be a pioneer in $\beta(\hat{k}) < \ell$.

We have shown the algorithm in the variant that keeps recursing until the length of the sequence is less than s, which has a running time of $O(\log n / \log\log n)$ (see Sect. II-C). As before, we could also stop recursing on a fixed level $k \geq 2$, and store the answers to *min_excess_pos* explicitly on the last level. As there are n_k^2 queries on level k, storing *all* answer would not result in sub-linear space. However, we can use similar ideas as in the sparse-table algorithm from Bender and Farach-Colton [8] to get a final space consumption of $O(n_2 \log n_2) = O(\frac{n}{(\log n)^2} \log n_2) = O(\frac{n}{\log n})$ bits. We formulate a final theorem that summarizes this section:

Theorem 1. *On top of the* $O(\frac{n \log\log n}{\log n})$ *bits from the data structures for* find_open, find_close, *and*

enclose, there are data structures for rr_enclose *with either* $O(\frac{\log n}{\log\log n})$ *query time and* $O(\sqrt{n}\,\mathrm{polylog}\,n)$ *bits of space, or with* $O(1)$ *query time and space* $O(\frac{n}{\log n})$ *bits.*

A. Computing Lowest Common Ancestors

We can use *rr_enclose* to compute LCAs in a BPS $S[0, 2n-1]$ as follows: given two nodes $i < j$ (we identify nodes with the position of their *opening* parenthesis), first compute $k = \mathit{rr\_enclose}(S, i, j)$. Now if $k \neq \perp$, return $\mathit{enclose}(S, k)$ as $\mathrm{LCA}(i, j)$. The other possibility is that $k = \perp$, which happens if and only if j is the son of $\mathrm{LCA}(i, j)$. In this case, we return $\mathit{enclose}(S, j)$.

In DFUDS, computing LCAs is also simple if we "reverse" the definition of *min_excess_pos* from $\min_{i \le k < j}\{k : \mu(k) \ge j\}$ to $\mathit{min\_excess\_pos}'(S, i, j) = \max_{i \le k < j}\{k : \mu(k) < i\}$. It is easily verified that the algorithm above can be adapted to deal with this modified definition. Then

$$k = \mathit{find\_open}(S, \mathit{min\_excess\_pos}'(S, i, j))$$

brings us directly to the description of $\mathrm{LCA}(i, j)$ (in DFUDS, we identify nodes with the position of their *closing* parenthesis). We can thus jump to the appropriate closing parenthesis by $\mathrm{LCA}(i, j) = \mathit{select}_)(S, \mathit{rank}_)(S, k+1) + 1)$.

IV. Application I: Range Minimum Queries

A. Problem Definition and Existing Algorithms

For an array $A[0, n-1]$ of n natural numbers or other objects from a totally ordered universe, a *range minimum query* $\mathrm{RMQ}_A(i, j)$ for $i \le j$ returns the *position* of a minimum in the sub-array $A[i, j]$; in symbols, $\mathrm{RMQ}_A(i, j) = \mathrm{argmin}_{i \le k \le j}\{A[k]\}$. This fundamental algorithmic problem has numerous applications, for example in data compression [14]; see recent articles on RMQs [8], [13] for more examples. In all of these applications, the array A in which the RMQs are performed is static and known in advance. Hence it makes sense to preprocess A into a *scheme* such that future RMQs can be answered quickly.

Preprocessing schemes for $O(1)$-RMQs can be classified into two different types: *systematic* and *non-systematic*. Systematic schemes must store the input array A verbatim along with the additional information for answering the queries. In such a case the query algorithm can consult A when answering the queries; this is indeed what all systematic schemes make heavy use of. On the contrary, non-systematic schemes must be able to obtain their final answer without consulting the array. This latter type is important for cases where only the *position* of the minimum matters, and not the *value* itself. The table in Fig. 5 summarizes existing data structures for RMQs; the top half displays the best systematic schemes, whereas the bottom half shows all known non-systematic schemes. Among the systematic ones, *compr* is a compressed scheme building on *succ*; there, H_k is the *empirical entropy of order* k, which is a common measure of compression. All of the non-systematic methods rely on constant-time LCA-computation in some succinctly encoded tree, either implicitly or explicitly.

name	space in bits
succ [15]	$2n + o(n) + \|A\|$
compr [16]	$nH_k + o(n) + \|A\|$
sada [17]	$4n + o(n)$
2dmin [13]	$2n + o(n)$
sct [18]	$2n + o(n)$

Figure 5. State-of-the art preprocessing schemes for $O(1)$-RMQs, where $|A|$ denotes the space for the read-only input array.

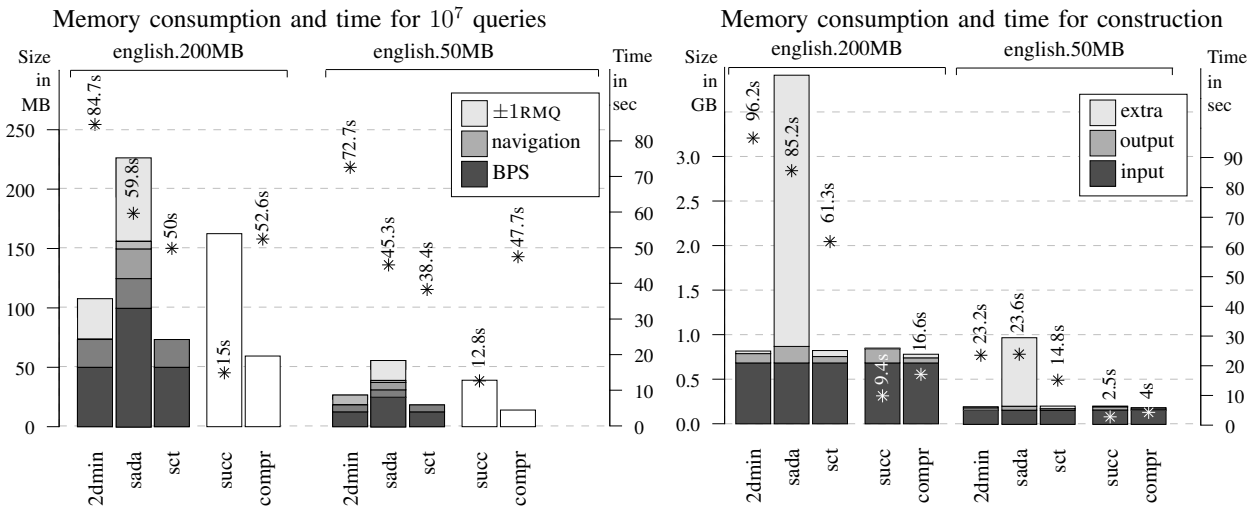

Figure 6. Left: sizes and query times of RMQ-structures on two LCP-arrays of texts from the *Pizza& Chili*-Corpus. Right: running time and memory consumption for the construction of the data structures.

B. Empirical Tests

The aim of this section is to show the practical advantages of using an LCA-method based on *rr_enclose* instead of ± 1RMQ. In principle, *rr_enclose* could be used in any of the non-systematic methods (as a substitute for ± 1RMQ). However, we opted for implementing previous methods [13], [17] as they were described at the time of their writing, and use *rr_enclose* only in the sct-scheme [18] that is based on the Super-Cartesian Tree [19]. We use the $O(\frac{\log n}{\log \log n})$-variant from Thm. 1. For reasons of completeness, we also included the two systematic schemes [15], [16] from Fig. 5 in our tests. We do not include the ± 1RMQ-solution of Sadakane and Navarro [9] in our tests, as their implementation appeared only after our work was conducted.

We implemented all data structures from Fig. 5 in C++. For a fair comparison we use the same support data structures (e.g., ± 1RMQ,rank/select/findopen,...) wherever possible. We compiled each data structure with `g++` version 4.1.2 with options `-O3 -DNDEBUG`. All tests were executed on an AMD Opteron 2220 (2.8 GHz) equipped with 24GB of RAM. We ran tests on random integer arrays and on LCP-arrays of all texts of size 50MB and 200MB from the *Pizza&Chili* site (http://pizzachili.di.unipi.it). As the results were all rather similar, we only display those for two LCP-arrays on English texts. The results of the complete test suite are available on the *sdsl* site.

The left bar plot of Fig. 6 shows the final space of the schemes; for a deeper analysis of the non-systematic schemes, we partition the bars into different components:

2dmin: From bottom to top, the boxes correspond to the $2n$ bits of the BPS of the 2d-Min-Heap [13], structures for navigation in the 2d-Min-Heap (see Sect. II-C), and ± 1RMQ-information on the excess-array of the BPS.

sada: The boxes — again from bottom to top — represent the space for: the BPS of the extended Cartesian Tree ($4n$ bits), rank structures for the pattern '()', rank structures for ')', a select structure for the pattern '()', and a ± 1RMQ structure on the excess sequence.

sct: Because we replace ± 1RMQ by *rr_enclose*, it consists of only two parts (again from bottom to top): the BPS of the Super-Cartesian Tree ($2n$ bits), and navigational information (Sect. II-C and Sect. III).

The final size of the non-systematic schemes highly depends on the size of the balanced parentheses

sequence (depicted as the lowest box of the stacks in Fig. 6). However, the space for the $o(n)$-terms is non-negligible, and we can indeed observe a clear advantage of the method that was pimped up with *rr_enclose*, namely scheme *sct*. Note again that this is *not* due to an inherent supremacy of the Super-Cartesian Tree — scheme *2dmin* (and, to a lesser extent, also *sada*) would show the same reduction in space if its ± 1RMQ were replaced with *rr_enclose*. Compared with the two systematic schemes (*succ* and *compr*), *sct* is almost as small as *compr*, the best of these[1] — this becomes even more amazing when we take into account that *the bars for succ and compr in the left plot of Fig. 6 do not include the space of the input array*, although it is needed at query time!

The running times for 10^7 random queries are also shown on the left of Fig. 6. Scheme *2dmin* is slowest, as it needs to invoke the *find_open*-operation for each query. The systematic scheme *succ* is fastest, which is not surprising given that it does not use any compressed data structures (all arrays are byte-aligned). The other three schemes (*sada*, *sct*, and *compr*) are about equally fast, showing that *rr_enclose* is also competitive in time.

The construction space (we measured the resident size) is depicted in the right plot of Fig. 6. In all bars, the lower box (dark gray) represents the space for the input array (we actually packed the LCP-values into $\lceil \log M \rceil$-bit wide words for arrays with maximum LCP-value M). Due to the explicit construction of the Cartesian Tree, *sada* takes about 1.0GB or 4.0GB for the LCP-array over a text of length 50MB or 200MB, respectively. Both other non-systematic implementations, *2dmin* and *sct*, do not need a construction of a tree and therefore take only the space of the input, the output, and a stack to build the 2d-Min-Heap or the Super-Cartesian tree. However, because the non-systematic implementations have to construct bit vectors and many support structures, the construction takes significant longer in contrast to the systematic schemes.

V. Application II: Compressed Suffix Trees

A. Description of Compressed Suffix Trees

Given a text $T \in \Sigma^\star$ of length n, the *suffix tree* of T is a rooted tree S with edge labels from $\Sigma^\star$, such that exactly all of T's sub-strings $T_{i..j}$ are spelled out when reading off edge labels from S (starting at the root). *Compressed suffix trees* (CSTs) can be viewed as a data structure with the same functionality as a plain suffix tree (access to edge labels, navigation, etc.), but with lower space occupancy [2], [4], [18], [20]. CSTs are all based on the *suffix array* of T, which is an array $A[0, n-1]$ of integers s.t. $T_{A[i]...n-1} < T_{A[i+1]...n-1}$ for all $0 \leq i < n-1$; i.e., A describes the lexicographic order of T's suffixes by "enumerating" them from the lexicographically smallest to the largest. Suffix arrays can be compressed to occupy less than the $n\lceil \log n \rceil$ bits that they would need normally; however, the time to access an arbitrary entry $A[i]$ rises to $\omega(1)$.

The LCA-operation is central to all compressed suffix trees. Indeed, it is used not only as an 'end-user' routine to actually compute lowest common ancestors, but also as a tool for other fundamental suffix-tree-specific operations like string depth, suffix links, moving to a child, etc.

B. Empirical Tests

All CSTs consist of at least 4 parts: a compressed suffix array (called csa henceforth), a compressed LCP-table (lcp), a BPS to represent the tree topology (bp), and navigational information on the BPS (bp_support). The original proposal of Sadakane (called *cst_sada*) uses $4n$ bits for bp and has three additional components: rank- and select-structures for patterns '10' (called rank10 and select10), and

[1]The reason that the space for the compressed scheme *compr* is significantly lower than that of *succ* is that we now use much smaller data structures for rank and select than in the original implementation [16].

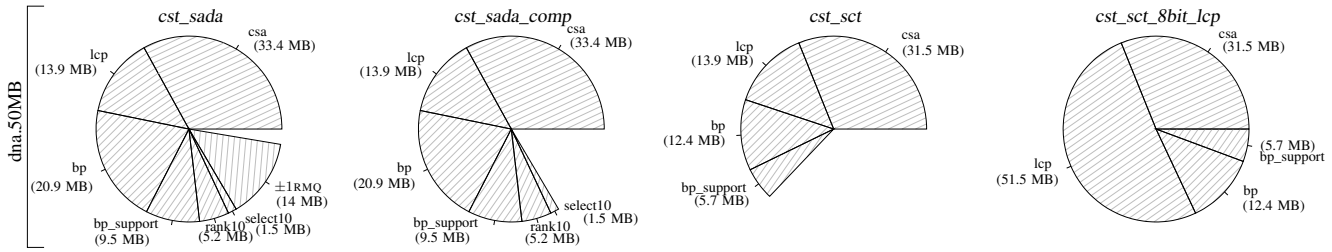

Figure 7. Sizes of different CST-implementations and their components on a DNA-file of size 50MB.

structures for ± 1RMQ on the (implicit) excess sequence. Fig. 7 shows the sizes of each part of *cst_sada* for a 50MB DNA-file. The structure for ± 1RMQ takes about a fifth of the whole *cst_sada*. This structure is now obsolete, as we can use *rr_enclose* instead. We call the resulting CST *cst_sada_comp*, and we can observe a significant gain in space. An even smaller CST (called *cst_sct*) was presented by Ohlebusch and Gog [18]. It uses a kind of DFUDS of the Super-Cartesian Tree [19] of the LCP-array for the tree topology (similar to the CST with constant-time previous-/next-smaller-value queries [4, Sect. 4.1]), which has only $2n$ bits. Therefore, both bp and bp_support for *cst_sct* have only half the size compared to *cst_sada*.

Concerning navigation time, we opted for testing the suffix-link method, which is a rather complex operation needing RMQs, access to LCP, and much more [4]. We got the following query times for 10^7 (truly) random suffix link operations: *cst_sada*: 31.6s, *cst_sada_comp*: 26.7s, and *cst_sct*: 579.0s. Comparing *cst_sada* with *cst_sada_comp*, we can observe that the use of *rr_enclose* also leads to faster navigation times. Concerning *cst_sct*, the huge increase in space is explained by the fact that its suffix-link operation depends on access to the LCP-array (this is not the case for *cst_sada* and *cst_sada_comp*). To balance out this bias, we also implemented a byte-aligned LCP-table and plugged it into *cst_sct*; the resulting CST is called *cst_sct_8bit_lcp*. It needs 30.1s for the suffix-link operations, showing that the slow-down in *cst_sct* originates indeed from the slow LCP-table, and *not* from the *rr_enclose*-operation.

ACKNOWLEDGMENT

We thank Enno Ohlebusch and the anonymous reviewers for their insightful comments.

REFERENCES

[1] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Structuring labeled trees for optimal succinctness, and beyond," in *Proc. FOCS*. IEEE Computer Society, 2005, pp. 184–196.

[2] K. Sadakane, "Compressed suffix trees with full functionality," *Theory of Computing Systems*, vol. 41, no. 4, pp. 589–607, 2007.

[3] J. Jansson, K. Sadakane, and W.-K. Sung, "Ultra-succinct representation of ordered trees," in *Proc. SODA*. ACM/SIAM, 2007, pp. 575–584.

[4] J. Fischer, V. Mäkinen, and G. Navarro, "Faster entropy-bounded compressed suffix trees," *Theor. Comput. Sci.*, vol. 410, no. 51, pp. 5354–5364, 2009.

[5] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. FOCS*. IEEE Computer Society, 1989, pp. 549–554.

[6] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses and static trees," *SIAM J. Comput.*, vol. 31, no. 3, pp. 762–776, 2001.

[7] R. F. Geary, N. Rahman, R. Raman, and V. Raman, "A simple optimal representation for balanced parentheses," *Theor. Comput. Sci.*, vol. 368, no. 3, pp. 231–246, 2006.

[8] M. A. Bender and M. Farach-Colton, "The LCA problem revisited," in *Proc. LATIN*, ser. LNCS, vol. 1776. Springer, 2000, pp. 88–94.

[9] K. Sadakane and G. Navarro, "Fully-functional succinct trees," 2009, CoRR, abs/0905.0768. To be presented at SODA'10.

[10] A. Golynski, "Optimal lower bounds for rank and select indexes," *Theor. Comput. Sci.*, vol. 387, no. 3, pp. 348–359, 2007.

[11] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," *ACM Transactions on Algorithms*, vol. 3, no. 4, p. Article No. 43, 2007.

[12] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs," in *Proc. FOCS*. IEEE Computer Society, 1997, pp. 118–126.

[13] J. Fischer, "Optimal succinctness for range minimum queries," 2008, CoRR, abs/0812.2775. To be presented at LATIN'10.

[14] G. Chen, S. J. Puglisi, and W. F. Smyth, "LZ factorization using less time and space," *Mathematics in Computer Science*, vol. 1, no. 4, pp. 605–623, 2007.

[15] J. Fischer and V. Heun, "A new succinct representation of RMQ-information and improvements in the enhanced suffix array," in *Proc. ESCAPE*, ser. LNCS, vol. 4614. Springer, 2007, pp. 459–470.

[16] J. Fischer, V. Heun, and H. M. Stühler, "Practical entropy bounded schemes for $O(1)$-range minimum queries," in *Proc. DCC*. IEEE Press, 2008, pp. 272–281.

[17] K. Sadakane, "Succinct data structures for flexible text retrieval systems," *J. Discrete Algorithms*, vol. 5, no. 1, pp. 12–22, 2007.

[18] E. Ohlebusch and S. Gog, "A compressed enhanced suffix array supporting fast string matching," in *Proc. SPIRE*, ser. LNCS, vol. 5721. Springer, 2009, pp. 51–62.

[19] J. Fischer and V. Heun, "Finding range minima in the middle: Approximations and applications," *Mathematics in Computer Science*, preprint, 27 Nov. 2009, doi:10.1007/s11786-009-0007-8.

[20] L. M. S. Russo, G. Navarro, and A. L. Oliveira, "Fully-compressed suffix trees," in *Proc. LATIN*, ser. LNCS, vol. 4957. Springer, 2008, pp. 362–373.