Revisiting the Folklore Algorithm for Random Access to Grammar-Compressed Strings

Alan M. Cleary ✉[1], Joseph Winjum[2], Jordan Dood[3], and Shunsuke Inenaga[4]

[1] National Center for Genome Resources, Santa Fe, NM, USA
acleary@ncgr.org
[2] Montana State University, Bozeman, MT, USA
joseph.winjum@ecat1.montana.edu
[3] Hyalite Technologies LLC, Bozeman, MT, USA
hyalitetechnologies@gmail.com
[4] Department of Informatics, Kyushu University, Fukuoka, Japan
inenaga.shunsuke.380@m.kyushu-u.ac.jp

Abstract. Grammar-based compression is a widely-accepted model of string compression that allows for efficient and direct manipulations on the compressed data. Most, if not all, such manipulations rely on the primitive *random access* queries, a task of quickly returning the character at a specified position of the original uncompressed string without explicit decompression. While there are advanced data structures for random access to grammar-compressed strings that guarantee theoretical query time and space bounds, little has been done for the *practical* perspective of this important problem. In this paper, we revisit a well-known folklore random access algorithm for grammars in the Chomsky normal form, modify it to work directly on general grammars, and show that this modified version is fast and memory efficient in practice.

Keywords: grammar-based compression · random access · straight-line programs

1 Introduction

Random access on grammar-compressed strings has been used as a key primitive in a number of efficient algorithms that directly work on compressed strings, including pattern matching [21,27,23,35,3], compressed-string indexing [10], q-gram frequencies [16,17], detection of palindromes and repetitions [26,19,20], convolutions [32], finger searches [2], and Lempel-Ziv factorizations in compressed space [11].

Given a grammar $\mathcal{G}$ in the Chomsky normal form for a text T, a folklore algorithm for this problem first computes and stores the length of the string that each non-terminal derives in $O(\mathsf{size}(\mathcal{G}))$-time and space, where $\mathsf{size}(\mathcal{G})$ denotes the total size of the productions in $\mathcal{G}$. Then, given a position p in T, one can climb down the corresponding path to $T[p]$ in the derivation tree for $\mathcal{G}$ in $O(\mathsf{h}(\mathcal{G}))$-time, where $\mathsf{h}(\mathcal{G})$ denotes the height of the derivation tree of $\mathcal{G}$. While $\mathsf{h}(\mathcal{G})$ can be as small as $\Theta(\log n)$ for some highly repetitive strings of length n with balanced grammars $\mathcal{G}$, $\mathsf{h}(\mathcal{G})$ can be as large as $\Theta(n)$ in the worst case. Bille et al. [4] showed how to preprocess $\mathcal{G}$ in $O(\mathsf{size}(\mathcal{G}))$-time and space so that later random access queries can be answered in $O(\log n)$-time, irrespective of $\mathsf{h}(\mathcal{G})$. Garnardi et al. [14] showed how to convert a given grammar $\mathcal{G}$ into another grammar $\mathcal{G}'$, with $\mathsf{size}(\mathcal{G}') = O(\mathsf{size}(\mathcal{G}))$ and $\mathsf{h}(\mathcal{G}') = O(\log n)$, that derives the same string as the original grammar $\mathcal{G}$, thus achieving $O(\log n)$-time random access using $O(\mathsf{size}(\mathcal{G}))$-space. Garnardi et al. [14] also presented an $O(\log n/\log\log n)$-time random access data structure with $O(\mathsf{size}(\mathcal{G})\log^{\epsilon} n)$-space for any $\epsilon > 0$, by generalizing the result of Belazzougui et al. [1]. This matches the cell-probe lower bound shown by Verbin and Yu [33], for strings that are only polynomially compressible in n.

While random access on grammars has been extensively studied in the theoretical perspective, as shown above, the only practical results that we are aware of are the works by Maruyama et al. [25] and Gagie et al. [12]. However, these approaches require specific grammar encodings and only work on RePair style grammars, making them incompatible with recent grammar-based compression algorithms [29,9,7]. In fact, we are not aware of a general random access algorithm that will work on any grammar.

In this work, we revisit the folklore algorithm for random access to grammar compressed strings and show that it can be improved to use significantly less space and generalized to operate directly on any grammar. Our experiments show that this modified folklore algorithm achieves state-of-the-art performance in both its space requirements and run-time.

2 Preliminaries

In this section, we define syntax and review information related this paper. Indexes start at 1.

2.1 Strings

Let Σ be an alphabet of size σ. An element in Σ^* is called a *string*. The length of a string T is denoted by $|T|$ and $n = |T|$. The empty string ε is the string of length 0, i.e. $|\varepsilon| = 0$. The pth character in a string T is denoted by $T[p]$ for $1 \leq p \leq n$, and the substring of T that begins at position p and ends at position q is denoted by $T[p..q]$ for $1 \leq p \leq q \leq n$. For convenience, let $T[p..q] = \varepsilon$ for $p > q$.

2.2 Grammar-Based Compression

A *context-free grammar* is a set of recursive rules that describe how to form strings from a language's alphabet. A context-free grammar is called an *admissible grammar* if the language it generates consists only of a single string.

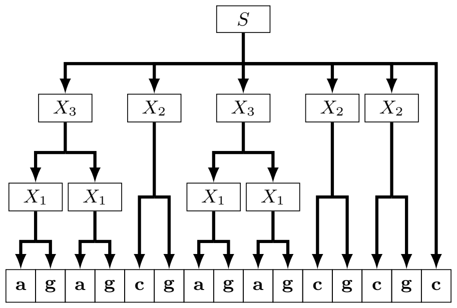
Grammar-based compression is a compression technique that computes an admissible grammar for a given string such that the computed grammar can be stored in less space than the original string. In what follows, we will call admissible grammars simply *grammars.*

Let $\mathcal{G} = \langle X, \Sigma, R, S \rangle$ be a grammar that generates T, where X is a set of non-terminal characters, Σ is the alphabet of T (i.e. terminal characters) and is disjoint from X, R is a finite relation in $X \times (X \cup \Sigma)^*$, and S is the symbol in X that should be used as the *start rule* when using $\mathcal{G}$ to generate T. R defines the *rules* of $\mathcal{G}$ as a set of m productions $\{X_i \to \mathrm{expr}_i \mid 1 \leq i \leq m\}$ such that each X_i is a non-terminal in X and expr_i is a non-empty sequence from $(\Sigma \cup \{X_1, \ldots, X_{i-1}\})^+$. The *size* of grammar $\mathcal{G}$ is the total length of the right-hand sides of the productions and is denoted $\mathsf{size}(\mathcal{G}) = \sum_{i=1}^{m} |\mathrm{expr}_i|$. We say that a non-terminal X_i *represents* a string w if w is the (unique) string that X_i derives. We only consider grammars with no useless rules and symbols, unless stated otherwise. $\mathsf{h}(\mathcal{G})$ denotes the height of the derivation tree of grammar $\mathcal{G}$.

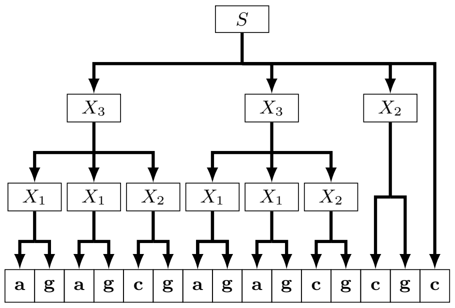
In this work we will discuss three types of grammars: *straight-line programs (SLPs)*, *Chomsky normal form (CNF) grammars*, and *RePair grammars.* An SLP $\mathcal{G}_{\mathrm{SLP}}$ is simply an admissible grammar. A CNF grammar $\mathcal{G}_{\mathrm{CNF}}$ is an SLP in the Chomsky normal form, i.e. every rule (including start rule S) is of the form $X_i \to c$ ($c \in \Sigma$) or $X_i \to X_\ell X_r$ ($\ell, r < i$).[5] A RePair grammar $\mathcal{G}_{\mathrm{RePair}}$, proposed in [22], is a CNF grammar in which the start rule can have arbitrarily many symbols in its righthand side, i.e. $S \to \mathrm{expr}$ with $\mathrm{expr} \in ((X \setminus S) \cup \Sigma)^+$. Any SLP can be converted to an equivalent CNF grammar [24] and any CNF grammar can be converted to an equivalent balanced CNF grammar with height $O(\log n)$ [14], where n is the length of the uncompressed string. See Figure 1 (a) and (b) for example RePair and SLP grammars, respectively.

Let $\mathcal{G}$ be a grammar that represents a string T of length n. A *random access query* on grammar $\mathcal{G}$ is, given a query position p ($1 \leq p \leq n$), return the pth character $T[p]$ in the uncompressed string T. The *random access problem on grammar-compression* is to preprocess a given grammar $\mathcal{G}$ and build a space-efficient (i.e. compressed) data structure on $\mathcal{G}$ that can quickly return the desired character $T[p]$ for query positions p. In practice, random access queries can be for entire substrings $T[p..q]$, where $1 \leq p \leq q \leq n$. Our experiments include results for substring queries but our algorithm descriptions only consider the single character version of the problem. This is because in our approach only the first character of the substring needs to be located in $\mathcal{G}$'s derivation tree; the rest of the substring can be generated by simply traversing the derivation tree from that location.

[5] While the term SLP is often used for grammar-compression in the Chomsky normal form, in this paper, for clarity, we use SLP to denote general admissible grammars and CNF to denote grammars in the Chomsky normal form.

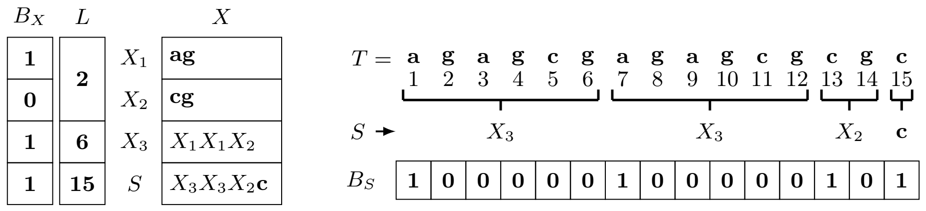


(a) The derivation tree for $\mathcal{G}_{\text{RePair}}$



S
X_3 X_3 X_2
X_1 X_1 X_2 X_1 X_1 X_2
a g a g c g a g a g c g c g c

(b) The derivation tree for $\mathcal{G}_{\text{SLP}}$



B_X	L		X
1	**2**	X_1	**ag**
0		X_2	**cg**
1	**6**	X_3	$X_1X_1X_2$
1	**15**	S	$X_3X_3X_2$**c**

$T =$	**a**	**g**	**a**	**g**	**c**	**g**	**a**	**g**	**a**	**g**	**c**	**g**	**c**	**g**	**c**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S \rightarrow$	X_3						X_3						X_2		**c**
B_S	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**

(c) Text T and data structures for the modified folklore algorithm on $\mathcal{G}_{\text{SLP}}$

Fig. 1: Example grammars and data structures for the modified folklore algorithm on string T = agagcgagagcgcgc. (a) and (b) depict the derivation trees for a RePair grammar $\mathcal{G}_{\text{RePair}}$ and an SLP grammar $\mathcal{G}_{\text{SLP}}$, respectively. (c) depicts the data structures used by our modified version of the folklore random access algorithm. Note that the subscript i for each non-terminal X_i conveys the array index of the non-terminal in the array of arrays representation of the grammars. For simplicity, terminal characters are used directly without a proxy non-terminal character.

3 Algorithms

In this section, we present novel algorithms for random access to grammar-compressed strings. As before, indexes start at 1.

3.1 The Folklore Algorithm

The folklore algorithm for random access to grammar-compressed strings is for CNF grammars. Given a grammar $\mathcal{G}_{\text{CNF}}$, the algorithm works by first computing an array A that stores the length of the string that each non-terminal represents. Then, given a position p in T, the corresponding path to $T[p]$ in the derivation tree is followed by looking up the string length of each non-terminal's left and right characters in A to determine which character contains the position, requiring $O(\log n)$-time when the grammar is balanced and $O(\mathsf{h}(\mathcal{G}_{\text{CNF}}))$-time otherwise. See Algorithm 1 for details.

Algorithm 1: Folklore Random Access

```
Data: Grammar $\mathcal{G}_{\mathrm{CNF}}$, i.e. an array of integer arrays
Data: Start rule $S$, i.e. an index in $\mathcal{G}_{\mathrm{CNF}}$
Data: Length lookup table $A$
Input: Position $p$ in $T$
Output: Character $T[p]$
1  $X_i \leftarrow S$
2  while $X_i$ is a non-terminal $(X_i \rightarrow X_\ell X_r)$ do
3      $leftLength \leftarrow A[X_i]$
4      if $p \leq leftLength$ then
5          $X_i \leftarrow \mathcal{G}_{\mathrm{CNF}}[X_i]_\ell$      ▷ $X_\ell$ denotes the first character in $X_i$'s array
6      else
7          $p \leftarrow p - leftLength$
8          $X_i \leftarrow \mathcal{G}_{\mathrm{CNF}}[X_i]_r$      ▷ $X_r$ denotes the second character in $X_i$'s array
9      end
10 end
11 return $X_i$
```

While this algorithm is fast both asymptotically and in practice, it requires much additional space. For instance, if the algorithm is given a non-CNF grammar $\mathcal{G}$, then an equivalent CNF grammar $\mathcal{G}_{\mathrm{CNF}}$ must be computed, which may require introducing useless rules that increase the size of the grammar. Additionally, using the naïve array of arrays grammar encoding [31] where each non-terminal character is an array index, the folklore algorithm requires:

(a) $2m \lg(m+\sigma)$ bits to represent the grammar and
(b) $m \lg(n)$ bits to represent the rule string lengths,

where m denotes the number of non-terminals in $\mathcal{G}_{\mathrm{CNF}}$, in which there are σ rules of form $X_i \rightarrow c$ ($c \in \Sigma$). Note that the bits to represent the rule string lengths require half as much space as the grammar itself!

3.2 Modified Folklore Algorithm

In our modified folklore algorithm to follow, the non-terminals are required to be sorted (and subsequently numbered) in increasing order of their expansion lengths. Given a grammar $\mathcal{G}$ with m non-terminals, we can simply sort them in $O(m \log m)$-time and renumber them in $O(\mathsf{size}(\mathcal{G}))$-time with $O(m \log(n))$ bits of working space by any suitable comparison-based sorting algorithm. The $O(m \log(n))$ bits of information is discarded after this preprocessing. If the non-terminals in $\mathcal{G}$ are already sorted, this step can be skipped.

To improve the folklore algorithm, we first observe that it can be easily extended to work on any SLP. Specifically, given an SLP $\mathcal{G}_{\mathrm{SLP}}$, the length of the string each non-terminal represents is computed and stored, as before. Then the position in T of each non-terminal character in start rule S is computed and stored as well. Now, given a position p in T, the algorithm will first look up the

Algorithm 2: Modified Folklore Random Access

```
Data: Ordered grammar $\mathcal{G}_{\text{SLP}}$, i.e. an array of integer arrays
Data: Start rule $S$, i.e. an index in $\mathcal{G}_{\text{SLP}}$
Data: Bitvector $B_X$
Data: Bitvector $B_S$
Data: Unique length array $L$
Input: Position $p$ in $T$
Output: Character $T[p]$
1  $r \leftarrow \text{rank}(B_S, p)$
2  $X_i \leftarrow \mathcal{G}_{\text{SLP}}[S][r]$
3  $p \leftarrow p - \text{select}(B_S, r)$
4  while $X_i$ is a non-terminal ($X_i \rightarrow \text{expr}_i$ with $\text{expr}_i \in (\Sigma \cup \{X_1, \ldots, X_{i-1}\})^+$)
    do
5      for $X_j$ in $\mathcal{G}_{\text{SLP}}[X_i]$ do
6          $r \leftarrow \text{rank}(B_X, X_j)$
7          $length \leftarrow L[r]$
8          if $p \leq length$ then
9              $X_i \leftarrow X_j$
10             break
11         else
12             $p \leftarrow p - length$
13         end
14     end
15 end
16 return $X_i$
```

character in the start rule that contains p and then descend the derivation tree from this character. However, since each rule in an SLP can have arbitrarily many symbols in its righthand side, instead of checking the left and right characters to determine which character contains the query position, a rule's characters are iterated until the character containing the position is found.

Using the naïve array of arrays grammar encoding, this extended version of the folklore algorithm requires:

(c) $\mathsf{size}(\mathcal{G}_{\text{SLP}}) \lg(m + \sigma)$ bits to represent the grammar and
(d) $(|S| + m) \lg(n)$ bits to represent the rule string lengths and start character positions.

Note that when $\mathcal{G}_{\text{SLP}}$ is a CNF grammar, (c) is equivalent to (a).

We observe that the following changes can be made to significantly reduce the space requirements of (d):

1. Using the naïve grammar representation of (c) where each non-terminal character is an array index, order (and subsequently renumber) the rules by string length – shortest to longest – with the start rule S last.
2. Create a sorted array L of the unique rule string lengths.

3. Create a length m bitvector B_X with a bit i set for the first rule X_i of each string length in the grammar array.
4. Create a length n bitvector B_S with a bit p set at the text position of each character in start rule S.

The modified folklore algorithm can then be used by first looking up the start rule character via a paired *rank-select* query on B_S. The algorithm can then descend the derivation tree by using *rank* queries on B_X to look up each rule's string length in L. See Algorithm 2 for details.

Using a standard bitvector the *rank* and *select* operations can be done in $O(1)$-time [34,6]. However, a standard bitvector requires $64\lceil |B|/64+1\rceil$ bits of space, where $|B|$ is the length of the bitvector. This is impractical since $|B_S| = n$.

To minimize the space required by bitvectors B_S and B_X, we propose using the sparse bitvector [30]. This bitvector answers *rank* queries in $O(\log \frac{|B|}{b})$-time and select queries in $O(1)$-time while requiring no more than $b(2+\log \frac{|B|}{b})$ bits of space, where b is the number of set bits. Using the sparse bitvector, the modified folklore algorithm requires no more than $|S|(2+\log \frac{n}{|S|})+|L|(2+\log \frac{m}{|L|})+|L|\lg |L|$ bits to represent the rule string lengths and start character positions.

We observe that the modified folklore algorithm is effectively equivalent to the original folklore algorithm when given a CNF grammar. Since any SLP can be converted to an equivalent balanced CNF grammar [24,14], this implies that when using standard bitvectors the modified folklore algorithm runs in $O(\log n)$-time, and when using sparse bitvectors it runs in $O(\log \frac{n}{|S|} + \log n \log \frac{m}{|L|})$-time. Without converting to the Chomsky normal form or balancing the grammar, the worst-case run-time is $O(n)$. However, algorithms that produce RePair grammars tend to create balanced grammars, and so the expected run-time on RePair grammars is $O(\log n)$. Although the nature of SLPs in general is less predictable, the grammars of the MR-RePair algorithm we use in our experiments have been shown to be isomorphic to grammars produced by the actual RePair algorithm [9], so we expect these grammars to have an $O(\log n)$ run-time as well, despite having to iterate rules' characters when descending the derivation tree.

4 Results

In this section, we describe our implementation and experimental results.

4.1 Implementation

Our implementation of the modified folklore algorithm is called *FRAS* - Folklore Random Access for SLPs. We implemented FRAS in C++. The bitvectors and their respective *rank* and *select* data structures were implemented using the Succinct Data Structure Library (SDSL) [15]. We used the naïve array of arrays encoding to represent grammars [31] and implemented the sparse bitvector variation of the folklore algorithm described in Section 3. The source code is available at `https://github.com/alancleary/FRAS`.

4.2 Experiments

We performed experiments on two corpora of data: the *Pizza&Chili* corpus[6] and a collection of pangenomes. For each corpus, we generated grammars and benchmarked our modified folklore algorithm (FRAS) against the original folklore algorithm (Folklore), the algorithm of [25] (FOLCA), and the algorithm of [12] (ShapedSLP[7]), measuring encoding size and random access run-time. For each grammar generated, we accessed substrings of length 1, 10, 100, and 1,000 at pseudo-random positions.[8] We performed this procedure 10,000 times for each substring length and computed the average run-time. See the following subsections for details.

Experiments were run on a server with two AMD EPYC 7543 32-Core 2.8GHz (3.7GHz max boost) processors and 2TB of 8-channel DDR4 3200MHz memory running CentOS Stream release 9. Note that this server is excessively overpowered for these experiments and was used for the purpose of stability and accuracy of measurement. Similar results can be achieved on a consumer laptop.

Pizza&Chili For the Pizza&Chili corpus, we generated grammars for all data sets from the *real* and *artificial* collections. For each data set, we generated grammars using Gonzalo Navarro's implementation[9] of RePair [22] and Isamu Furuya's implementation[10] of MR-RePair [9]. MR-RePair generates SLPs that are isomorphic to RePair grammars. We included these grammars to test our hypothesis that the run-times of our FRAS algorithm should be approximately equivalent on an SLP as on an equivalent RePair grammar. The Folklore, FOLCA, and ShapedSLP algorithms were not benchmarked on MR-RePair SLPs as they only work on RePair grammars. See Table 5 in Appendix A for information about the Pizza&Chili data sets and their respective grammars. The space used by each algorithm is listed in Table 1 and the run-times of each algorithm are listed in Table 2.

We found that FRAS consistently used less space than Folklore, especially on the MR-RePair grammars. However, FRAS was also consistently slower than Folklore. This is expected as both FOLCA and ShapedSLP also use less space than Folklore but are slower. FRAS, however, consistently used more space than FOLCA and ShapedSLP but was also much faster. Moreover, FRAS was faster than FOLCA and ShapedSLP on every data set and query size, and it was the only algorithm of the three to achieve sub-microsecond run-times.

Interestingly, FRAS on MR-RePair grammars did not use much more space than FOLCA and ShapedSLP but it was the fastest of all the algorithms, ex-

[6] `https://pizzachili.dcc.uchile.cl/`

[7] Note that in [12] "SLP" refers to RePair grammars, thus ShapedSLP is only compatible with RePair grammars.

[8] Pseudo-random numbers were generated on a uniform distribution using the xoroshiro128+ generator [5]. The generator was seeded so that the same numbers were used by every algorithm.

[9] `https://users.dcc.uchile.cl/~gnavarro/software/repair.tgz`

[10] `https://github.com/izflare/MR-RePair`

cluding Folklore. This suggests that although the size of the rules in MR-RePair grammars is unbounded FRAS is acheiving the expected run-time and is faster than FRAS on RePair grammars simply because MR-RePair grammars are typically smaller.

Pangenomes A *pangenome* is a collection of genomes from the same species. The ability to efficiently store and access these data at scale is an important problem in bioinformatics [8]. Grammar-based compression is particularly well suited to compressing these data as the size of the grammars scales relative to information content, rather than input size [28]. To demonstrate the practicality of our algorithm, we generated grammars for three pangenomes: the 12 yeast assemblies from the Yeast Population Reference Panel (YPRP) [36]; the 25 Maize assemblies from the nested association mapping (NAM) population [18]; and 1000 copies of Human chromosome 19 (c1000) used by Gagie et al. in [12]. Grammars were generated using BigRePair as it is currently the only grammar-based compression algorithm that can generate grammars for the NAM and c1000 data sets [13]. See Table 6 in Appendix A for information about these data sets and their respective grammars. The space used by each algorithm is listed in Table 3 and the run-times of each algorithm are listed in Table 4.

As with the Pizza&Chili corpus, we found that FRAS consistently used less space than Folklore and was consistently slower. And, again, FRAS consistently used more space than FOLCA and ShapedSLP but was also much faster, beating FOLCA and ShapedSLP on every data set and query size.

5 Conclusion

In this work, we showed how the folklore algorithm for random access to grammar-compressed strings can be modified to work on any grammar and achieve good space and run-time performance in practice. We believe this is the first random access algorithm for grammar-compressed strings that works directly on any grammar, thus further enhancing the usefulness of the algorithm. In future work we would like to further improve the modified folklore algorithm by representing the grammar in a manner that uses less space with minimal effect on run-time performance.

Acknowledgments. The work of Alan M. Cleary, Joseph Winjum, and Jordan Dood was support by NSF award number 2105391. The work of Shunsuke Inenaga was supported by JSPS KAKENHI grant numbers JP 20H05964, JP23K24808, JP23K18466. We would like to thank the authors of [12] for sharing the c1000 data set with us. You saved us much time and computation.

Table 1: The space used by the random access algorithms benchmarked on the *Pizze&Chili* corpus. **Data Set** is the names of the data sets and what collection they belong to. The **MR-FRAS**, **FRAS**, **Folklore**, **FOLCA**, and **ShapedSLP** columns are the space used by each algorithm, where **MR-FRAS** is **FRAS** run on MR-RePair grammars; all other results are on RePair grammars. All space is in megabytes.

Data Set		**MR-FRAS**	**FRAS**	**Folklore**	**FOLCA**	**ShapedSLP**
real	*Escherichia_Coli*	17.2	22.9	41.1	13.8	14.0
	cere	16.3	23.7	37.4	13.1	14.0
	coreutils	9.8	16.6	22.7	7.9	9.0
	einstein.de.txt	0.4	0.5	0.7	0.3	0.3
	einstein.en.txt	0.9	1.2	1.9	0.7	0.6
	influenza	8.0	8.8	17.6	6.1	5.5
	kernel	5.6	9.2	13.0	4.5	5.0
	para	21.3	29.5	49.3	17.4	18.2
	world_leaders	1.7	2.1	3.4	1.1	1.1
artificial	*fib41*	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1
	rs.13	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1
	tm29	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1

Table 2: Random access run-times for grammars built on the *Pizze&Chili* corpus. **Data Set** is the names of the data sets and what collection they belong to. The **MR-FRAS**, **FRAS**, **Folklore**, **FOLCA**, and **ShapedSLP** columns are the algorithms benchmarked and their query run-times, where **MR-FRAS** is **FRAS** run on MR-RePair grammars; all other results are on RePair grammars. For the run-times, **1**, **10**, **100**, and **1,000** are the lengths of the substrings queried. All run-time are in microseconds.

Data Set		**MR-FRAS**				**FRAS**				**Folklore**				**FOLCA**				**ShapedSLP**			
		1	**10**	**100**	**1,000**	**1**	**10**	**100**	**1,000**	**1**	**10**	**100**	**1,000**	**1**	**10**	**100**	**1,000**	**1**	**10**	**100**	**1,000**
real	*Escherichia_Coli*	1.8	1.9	3.4	17.7	5.8	6.0	8.4	29.2	1.1	1.2	2.2	12.0	25.0	26.3	42.0	191.2	21.0	22.7	42.5	238.7
	cere	1.3	1.5	3.0	16.2	4.4	4.7	7.2	28.5	1.0	1.1	2.1	10.9	17.7	19.4	34.7	182.3	14.2	16.3	36.5	235.8
	coreutils	17.5	17.8	19.5	34.5	42.2	42.7	46.3	69.1	6.4	7.2	9.5	20.9	156.0	158.3	180.7	355.4	242.9	247.2	276.7	491.9
	einstein.de.txt	0.8	1.0	2.4	15.2	1.4	1.6	3.4	19.0	0.5	0.7	1.6	10.0	10.6	12.3	28.1	176.7	6.6	8.6	27.8	217.6
	einstein.en.txt	1.0	1.2	2.8	16.5	1.3	1.5	3.4	19.2	0.7	0.8	1.8	10.2	11.8	13.6	30.2	184.2	6.2	8.3	27.7	217.9
	influenza	0.3	0.5	2.2	17.1	0.5	0.6	2.3	17.0	0.6	0.7	1.6	10.4	10.6	12.3	27.3	172.1	3.0	5.0	24.1	213.6
	kernel	5.1	5.3	7.0	21.6	15.2	15.9	18.1	36.0	1.9	2.2	3.4	13.3	51.7	54.6	72.3	241.3	60.5	64.3	85.7	289.7
	para	0.9	1.1	2.6	17.4	1.8	2.2	4.8	27.4	0.8	1.0	2.1	11.9	13.2	14.9	30.7	182.3	5.8	7.7	27.0	218.8
	world_leaders	0.5	0.7	1.9	13.0	1.1	1.3	2.7	14.5	0.6	0.7	1.3	7.6	11.8	13.3	28.6	166.7	5.7	7.7	26.7	214.0
artificial	*fib41*	0.9	1.0	1.2	7.4	1.0	1.0	1.2	7.4	0.2	0.3	0.6	3.6	2.6	3.3	10.4	81.3	4.2	5.7	20.8	172.1
	rs.13	0.9	1.0	1.3	7.9	0.9	1.0	1.2	7.4	0.3	0.3	0.6	3.8	3.1	3.8	11.2	85.8	4.7	6.2	22.0	179.1
	tm29	0.9	0.9	1.2	7.0	0.8	0.9	1.3	7.6	0.2	0.3	0.6	3.7	2.8	3.6	10.6	80.9	4.7	6.3	22.5	183.1

Table 3: The space used by the random access algorithms benchmarked on the *Pangenome* corpus. **Data Set** is the names of the data sets and the **FRAS**, **Folklore**, **FOLCA**, and **ShapedSLP** columns are the space used by each algorithm. All results are on BigRePair grammars. All space is in megabytes.

Data Set	**FRAS**	**Folklore**	**FOLCA**	**ShapedSLP**
YPRP	49.9	75.4	25.9	31.4
Maize	3779.6	6400.4	2529.8	3020.2
c1000	122.3	217.4	86.5	80.6

Table 4: Random access run-times for grammars built on the *Pangenome* corpus. **Data Set** is the names of the data sets and the **FRAS**, **Folklore**, **FOLCA**, and **ShapedSLP** columns are the algorithms benchmarked and their query run-times. For the run-times, **1**, **10**, **100**, and **1,000** are the lengths of the substrings queried. All results are on BigRePair grammars. All run-time are in microseconds.

Data Set	**FRAS**				**Folklore**				**FOLCA**				**ShapedSLP**			
	1	**10**	**100**	**1,000**	**1**	**10**	**100**	**1,000**	**1**	**10**	**100**	**1,000**	**1**	**10**	**100**	**1,000**
YPRP	1.8	2.2	5.7	37.4	0.9	1.1	2.5	14.7	11.5	13.3	29.6	187.1	2.9	4.9	24.9	220.1
Maize	3.9	4.4	8.5	44.5	3.1	3.6	5.7	24.0	22.3	24.3	44.3	233.6	9.0	11.2	35.1	269.5
c1000	3.2	3.6	7.4	41.0	2.5	2.7	4.8	21.2	16.4	18.3	34.8	192.2	6.9	9.2	31.1	242.2

References

1. Belazzougui, D., Cording, P.H., Puglisi, S.J., Tabei, Y.: Access, rank, and select in grammar-compressed strings. In: Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9294, pp. 142–154 (2015)
2. Bille, P., Cording, P.H., Gørtz, I.L.: Compressed subsequence matching and packed tree coloring. Algorithmica **77**(2), 336–348 (2017)
3. Bille, P., Gørtz, I.L., Cording, P.H., Sach, B., Vildhøj, H.W., Vind, S.: Fingerprints in compressed strings. J. Comput. Syst. Sci. **86**, 171–180 (2017)
4. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM J. Comput. **44**(3), 513–539 (2015)
5. Blackman, D., Vigna, S.: Scrambled linear pseudorandom number generators. ACM Trans. Math. Softw. **47**(4) (sep 2021)
6. Clark, D.: Compact pat trees (1997)
7. Cleary, A., Dood, J.: Constructing the CDAWG CFG using LCP-intervals. In: 2023 Data Compression Conference (DCC). pp. 178–187 (2023)
8. Consortium, T.C.P.G.: Computational pan-genomics: status, promises and challenges. Briefings in Bioinformatics **19**(1), 118–135 (10 2016)
9. Furuya, I., Takagi, T., Nakashima, Y., Inenaga, S., Bannai, H., Kida, T.: Practical grammar compression based on maximal repeats. Algorithms **13**(4), 103 (2020)
10. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: A faster grammar-based self-index. In: LATA 2012. Lecture Notes in Computer Science, vol. 7183, pp. 240–251. Springer (2012)
11. Gagie, T., Goga, A., Jez, A., Navarro, G.: Space-efficient conversions from slps. In: LATIN 2024. Lecture Notes in Computer Science, vol. 14578, pp. 146–161 (2024)
12. Gagie, T., I, T., Manzini, G., Navarro, G., Sakamoto, H., Benkner, L.S., Takabatake, Y.: Practical random access to SLP-compressed texts. In: SPIRE 2020. Lecture Notes in Computer Science, vol. 12303, pp. 221–231 (2020)
13. Gagie, T., I, T., Manzini, G., Navarro, G., Sakamoto, H., Takabatake, Y.: Rpair: Rescaling RePair with rsync. In: Brisaboa, N.R., Puglisi, S.J. (eds.) String Processing and Information Retrieval. pp. 35–44. Springer International Publishing, Cham (2019)
14. Ganardi, M., Jez, A., Lohrey, M.: Balancing straight-line programs. J. ACM **68**(4), 27:1–27:40 (2021)
15. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) Experimental Algorithms. pp. 326–337. Springer International Publishing, Cham (2014)
16. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Computing q-gram non-overlapping frequencies on SLP compressed texts. In: SOFSEM 2012. Lecture Notes in Computer Science, vol. 7147, pp. 301–312 (2012)
17. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Fast q-gram mining on SLP compressed strings. J. Discrete Algorithms **18**, 89–99 (2013)
18. Hufford, M.B., Seetharam, A.S., Woodhouse, M.R., Chougule, K.M., Ou, S., Liu, J., Ricci, W.A., Guo, T., Olson, A., Qiu, Y., Coletta, R.D., Tittes, S., Hudson, A.I., Marand, A.P., Wei, S., Lu, Z., Wang, B., Tello-Ruiz, M.K., Piri, R.D., Wang, N., won Kim, D., Zeng, Y., O'Connor, C.H., Li, X., Gilbert, A.M., Baggs, E., Krasileva, K.V., Portwood, J.L., Cannon, E.K.S., Andorf, C.M., Manchanda, N., Snodgrass, S.J., Hufnagel, D.E., Jiang, Q., Pedersen, S., Syring, M.L., Kudrna, D.A., Llaca,

V., Fengler, K., Schmitz, R.J., Ross-Ibarra, J., Yu, J., Gent, J.I., Hirsch, C.N., Ware, D., Dawe, R.K.: De novo assembly, annotation, and comparative analysis of 26 diverse maize genomes. Science **373**(6555), 655–662 (2021)
19. I, T., Matsubara, W., Shimohira, K., Inenaga, S., Bannai, H., Takeda, M., Narisawa, K., Shinohara, A.: Detecting regularities on grammar-compressed strings. Inf. Comput. **240**, 74–89 (2015)
20. I, T., Nishimoto, T., Inenaga, S., Bannai, H., Takeda, M.: Compressed automata for dictionary matching. Theor. Comput. Sci. **578**, 30–41 (2015)
21. Karpinski, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions. Nord. J. Comput. **4**(2), 172–186 (1997)
22. Larsson, N., Moffat, A.: Off-line dictionary-based compression. Proceedings of the IEEE **88**(11), 1722–1732 (2000)
23. Lifshits, Y.: Processing compressed texts: A tractability border. In: Ma, B., Zhang, K. (eds.) CPM 2007. Lecture Notes in Computer Science, vol. 4580, pp. 228–240. Springer (2007)
24. Lohrey, M.: Algorithmics on slp-compressed strings: A survey. Groups - Complexity - Cryptology **4**(2), 241–299 (2012)
25. Maruyama, S., Tabei, Y., Sakamoto, H., Sadakane, K.: Fully-online grammar compression. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) String Processing and Information Retrieval. pp. 218–229. Springer International Publishing, Cham (2013)
26. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theor. Comput. Sci. **410**(8-10), 900–913 (2009)
27. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight line programs. Journal of Discrete Algorithms **1**(1), 187–204 (2000)
28. Navarro, G.: Indexing highly repetitive string collections, part ii: Compressed indexes. ACM Computing Surveys **54**(2) (feb 2021)
29. Nunes, D.S.N., Louza, F., Gog, S., Ayala-Rincón, M., Navarro, G.: A grammar compression algorithm based on induced suffix sorting. In: 2018 Data Compression Conference. pp. 42–51 (2018)
30. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: ALENEX 2007. pp. 60–70
31. Tabei, Y., Takabatake, Y., Sakamoto, H.: A succinct grammar compression. In: Fischer, J., Sanders, P. (eds.) Combinatorial Pattern Matching. pp. 235–246. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
32. Tanaka, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Computing convolution on grammar-compressed text. In: DCC 2013. pp. 451–460. IEEE (2013)
33. Verbin, E., Yu, W.: Data structure lower bounds on random access to grammar-compressed strings. In: CPM 2013. Lecture Notes in Computer Science, vol. 7922, pp. 247–258 (2013)
34. Vigna, S.: Broadword implementation of rank/select queries. In: McGeoch, C.C. (ed.) WEA 2008. pp. 154–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
35. Yamamoto, T., Bannai, H., Inenaga, S., Takeda, M.: Faster subsequence and don't-care pattern matching on compressed texts. In: CPM 2011. Lecture Notes in Computer Science, vol. 6661, pp. 309–322 (2011)
36. Yue, J.X., Li, J., Aigrain, L., Hallin, J., Persson, K., Oliver, K., Bergström, A., Coupland, P., Warringer, J., Lagomarsino, M.C., Fischer, G., Durbin, R., Liti, G.: Contrasting evolutionary genome dynamics between domesticated and wild yeasts. Nature Genetics **49**(6), 913–924 (Apr 2017)

A Appendix

Table 5: Data sets used from the *Pizze&Chili* corpus. **Data Set** is the names of the data sets and what collection they belong to, **Size** is the number of characters in each data set, and **MR-RePair** and **RePair** are information about the grammars generated for these data sets. For the grammars, **Rules** is the number of rules in the grammars, **Depth** is the maximum depth of the grammars, **Start** is the size of the start rules, and **Size** is the total lengths of the right-hand sides of the rules in each grammar, excluding the start rule.

Data Set		**Size**	**MR-RePair**				**RePair**			
			Rules	**Depth**	**Start**	**Size**	**Rules**	**Depth**	**Start**	**Size**
real	*Escherichia_ Coli*	112,689,515	712,228	23	712,484	1,595,881	2,012,087	3,279	1,601,482	4,024,174
	cere	461,286,644	836,956	29	648,659	3,392,707	2,561,292	1,359	655,298	5,122,584
	coreutils	205,281,778	437,054	30	153,775	2,270,187	1,821,734	43,728	153,346	3,643,468
	einstein.de.txt	92,758,441	21,787	42	12,683	71,709	49,949	269	12,665	99,898
	einstein.en.txt	467,626,544	49,565	48	62,591	150,233	100,611	1,355	62,473	201,222
	influenza	154,808,555	429,027	28	897,657	1,088,872	643,587	366	886,836	1,287,174
	kernel	257,961,616	246,596	34	69,537	1,304,343	1,057,914	5,822	69,427	2,115,828
	para	429,265,758	1,079,287	30	1,134,361	4,157,167	3,093,873	487	1,147,650	6,187,746
	world_ leaders	46,968,181	100,293	30	98,397	309,222	206,508	463	94,327	413,016
artificial	*fib41*	267,914,296	38	40	3	76	38	40	3	76
	rs.13	216,747,218	55	45	121	26	66	47	24	132
	tm29	268,435,456	51	29	6	126	75	45	6	150

Table 6: Data sets used from the *Pangenome* corpus. **Data Set** is the names of the data sets, **Size** is the number of characters in each data set, and **BigRePair** is information about the grammars generated for these data sets. For the grammars, **Rules** is the number of rules in the grammars, **Depth** is the maximum depth of the grammars, **Start** is the size of the start rules, and **Size** is the total lengths of the right-hand sides of the rules in each grammar, excluding the start rule.

Data Set	**Size**	**BigRePair**			
		Rules	**Depth**	**Start**	**Size**
YPRP	143,169,450	5,911,887	37	642,828	11,823,774
Maize	55,270,577,570	432,651,138	47	79,378,700	865,302,276
c1000	59,125,115,010	12,898,128	45	4,495,360	21,300,896