Constructing the CDAWG CFG using LCP-Intervals

Alan Cleary* and Jordan Dood†

*National Center for Genome Resources
Santa Fe, NM, USA
acleary@ncgr.org

†Montana State University
Bozeman, MT, USA
jordan.dood@montana.edu

Abstract

It is known that a context-free grammar (CFG) that produces a single string can be derived from the compact directed acyclic word graph (CDAWG) for the same string. In this work, we show that the CFG derived from a CDAWG is deeply connected to the maximal repeat content of the string it produces and thus has $\mathcal{O}(m)$ rules, where m is the number of maximal repeats in the string. We then provide a generic algorithm based on this insight for constructing the CFG from the LCP-intervals of a string in $\mathcal{O}(n)$ time, where n is the length of the string. This includes a novel data-structure to support stabbing queries on LCP-intervals in $\mathcal{O}(1+k)$ time after $\mathcal{O}(n)$ preprocessing time, where k is the number of intervals stabbed. These results connect the CFG to properties of the string it produces and relates it to other string data-structures, allowing it to be studied independently of the CDAWG and providing opportunity for innovation of grammar-based compression algorithms.

Introduction

Grammar-based compression is the method of compressing a string by computing a context-free grammar (CFG) that produces the string (and no others) such that the size of the grammar is smaller than the string. A CFG that produces a single string is called a straight-line grammar (SLG). The problem of computing the smallest SLG for a string is known as the *smallest grammar problem* (SGP) and is NP-complete [1]. Despite the hardness of the SGP, grammar-based compressors have remained of interest due to their effective compression in practice [1–4].

There is a variety of grammar-based compression algorithms, many of which have optimal asymptotic run-time complexity and bounded approximation ratios [1]. Unfortunately, these heuristics are generally not related to other string algorithms and structures, precluding any benefit from the rich body of existing literature on stringology. For instance, the seminal RePair algorithm greedily adds pairs of characters to the CFG in most-frequent-pair-first order [2]. Although it was shown in [4] that this heuristic can be equivalent to greedily selecting the most frequent maximal repeat at each iteration of the algorithm, these maximal repeats may include non-terminal characters of the CFG and may not actually be maximal in the original string.

In [5] the authors study how the compact directed acyclic word graph (CDAWG) may be augmented to support additional string operations without increasing space complexity asymptotically. As an aside, they observe that an SLG that produces the original string may be derived from their CDAWG and that such a CFG "might have independent interest." Indeed, this CFG is of great interest as its relationship to the CDAWG transitively connects it to other string data-structures and properties of the

2375-0359/23/$31.00 ©2023 IEEE
DOI 10.1109/DCC55655.2023.00026

string it produces, such as the Ziv-Lempel decomposition, maximal and tandem repeats, minimal unique substrings, and minimal absent words [6–8]. This provides an opportunity for existing algorithms based on these data-structures and string properties to be ported to the CFG and for the development of new algorithms specifically for the CFG. Conversely, this CFG connects these data-structures and string properties to *string attractors* and may serve as a pathway to better connecting them to dictionary compressors in general [3]. Unfortunately, the authors of [5] do not provide an algorithm for constructing the CFG independently of their augmented CDAWG.

In this work, we further explore the properties of the CFG derived from the CDAWG of [5] and show that it is deeply connected to the maximal repeat content of the string it produces and thus has $\mathcal{O}(m)$ rules, where m is the number of maximal repeats in the string. We then provide a generic algorithm based on this insight for constructing the CFG from the LCP-intervals of a string in $\mathcal{O}(n)$ time, where n is the length of the string. This includes a novel data-structure to support stabbing queries on LCP-intervals in $\mathcal{O}(1 + k)$ time after $\mathcal{O}(n)$ preprocessing time, where k is the number of intervals stabbed. We use LCP-intervals because they can be computed from a variety of string data-structures, further relating the CFG to these data-structures. Moreover, these results allow the CFG to be studied independently of the CDAWG and provide opportunity for innovation of grammar-based compression algorithms. Due to space limitations we omit some proofs.

Preliminaries

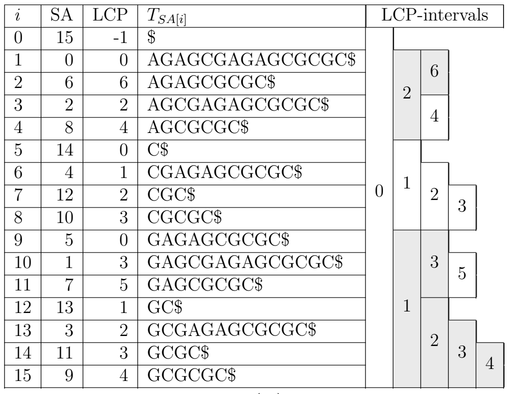
In this section, we define syntax and review information related this paper, including definitions, existing theorems, and simple corollaries. Indexes start at 0.

Strings

Let T be a string over an alphabet Σ, where $n = |T|$ and $\sigma = |\Sigma|$. We assume that all strings end with a unique character $\$ \notin \Sigma$ that is lexicographically smaller than any character in Σ. A *repeat* ω is a substring of T that occurs at least twice. A *left-extension* of repeat ω is a substring $\alpha\omega$ in T, where $\alpha \in \Sigma$. Similarly, a *right-extension* of repeat ω is a substring $\omega\alpha$ in T. A repeat ω is *left-maximal* if every left-extension $\alpha\omega$ occurs fewer times in T than ω. Similarly, a repeat ω is *right-maximal* if every right-extension $\omega\alpha$ occurs fewer times in T than ω. A repeat ω is a *maximal repeat* if it is both left- and right-maximal. There is at most $n - 1$ maximal repeats in a string [9]. Since the number of maximal repeats is a measure of the repetitiveness of a string and it tends to be much smaller than $n-1$ in practice, we denote the number of maximal repeats as m to make this distinction. The *longest common prefix* of two strings is the longest substring that prefixes both strings.

Suffix Arrays and LCP-intervals

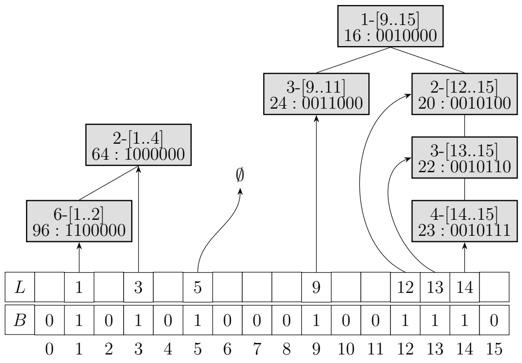
We assume that the reader is already familiar with the suffix tree [6]. The *suffix array* (SA) of a string T is an array containing the start positions of the suffixes of T in lexicographic order. We denote the suffix of T starting at index j as T_j, so the suffix at index i of SA would be denoted $T_{\mathrm{SA}[i]}$. The *longest common prefix array* (LCP)



i	SA	LCP	$T_{SA[i]}$
0	15	-1	$
1	0	0	AGAGCGAGAGCGCGC$
2	6	6	AGAGCGCGC$
3	2	2	AGCGAGAGCGCGC$
4	8	4	AGCGCGC$
5	14	0	C$
6	4	1	CGAGAGCGCGC$
7	12	2	CGC$
8	10	3	CGCGC$
9	5	0	GAGAGCGCGC$
10	1	3	GAGCGAGAGCGCGC$
11	7	5	GAGCGCGC$
12	13	1	GC$
13	3	2	GCGAGAGCGCGC$
14	11	3	GCGC$
15	9	4	GCGCGC$

LCP-intervals
0
2
6
4
1
2
3
1
3
5
2
3
4

(a)



(b)

Figure 1: Data-structures for the string $T = AGAGCGAGAGCGCGC\$$. (a) The suffix array (SA), longest common prefix array (LCP), suffixes, and LCP-intervals. The LCP-intervals of maximal repeats are highlighted in grey. (b) The data-structure for performing interval stabbing queries on an LCP-interval tree. The LCP-interval tree in (b) is composed of the maximal repeat LCP-intervals from (a). Each interval has an unsigned integer identifier used as a bit vector when computing stabbing queries.

is an array that stores the length of the longest common prefix of all consecutive pairs of suffixes in the suffix array. More formally, let $lcp(T_1, T_2)$ be a function that returns the length of the longest common prefix between two strings T_1 and T_2. Then $\text{LCP}[i] = lcp(T_{\text{SA}[i-1]}, T_{\text{SA}[i]})$ for every $1 \leq i < n$ and $\text{LCP}[0] = -1$. An *LCP-interval* represents a subarray of SA where every suffix has a common prefix, the length of which is called the interval's *LCP-value*. More formally, an LCP-interval $[i..j]$, where $0 \leq i < j < n$, with LCP-value l has the following properties:

$$\text{LCP}[i] < l \tag{1}$$
$$\text{LCP}[k] \geq l \text{ for all } k \text{ with } i+1 \leq k \leq j \tag{2}$$
$$\text{LCP}[k] = l \text{ for at least one } k \text{ with } i+1 \leq k \leq j \tag{3}$$
$$\text{LCP}[j+1] < l \tag{4}$$

We denote the LCP-interval $[i..j]$ with LCP-value l as l-$[i..j]$. We overload the term *LCP-interval* to refer to an LCP-interval and its LCP-value unless stated otherwise. There is at most $n-1$ LCP-intervals for a string. By definition the LCP-intervals form a tree structure of nested intervals known as the *LCP-interval tree* [6].

Theorem 1. *There is a bijection between the nodes and edges of the LCP-interval tree and the internal nodes and edges of the suffix tree [6].*

Indeed, the LCP-interval tree can be thought of as the suffix tree without leaf nodes. As such, traversing the suffix tree can be emulated by traversing the LCP-interval tree. The advantage of the LCP-interval tree is that it is conceptual — LCP-intervals can be computed in a tree-traversal order without the tree or the LCP array. Additionally, LCP-intervals can be computed from a variety of data-structures, including the more succinct FM-index [7] and run-length Burrows-Wheeler transform [8]. An example of a suffix array and corresponding LCP data-structures is shown in Figure 1.

CDAWG

The *compact directed acyclic word graph* (CDAWG) is the minimal compact automaton that recognizes all the suffixes of a string. Although it may be constructed directly from a string, it is useful to note that the CDAWG is a minified suffix tree.

Lemma 2. *The CDAWG is a suffix tree that has been minified by collapsing together all isomorphic subtrees and replacing all leaf nodes with a single sink node [5].*

See Figure 1 in [5] for an example of this equivalence. Similar to a suffix tree, each edge in a CDAWG has a *label* representing a non-empty string, and each path from the root node, or *source*, to an internal node has a label that is the concatenation of the labels of the edges in the path. Every such path represents a repeat that prefixes a suffix recognized by the CDAWG.

Lemma 3. *The longest path label from the source to each internal node of a CDAWG represents a maximal repeat [9, Lemma 3].*

Corollary 3.1. *There is a bijection between the internal nodes of a CDAWG and the maximal repeats of the string it encodes [9, Theorem 1]*

Corollary 3.2. *Excluding the sink node, the paths of the maximal repeats in a CDAWG form a spanning tree rooted at the source node (proof omitted).*

Context-Free Grammars

A *context-free grammar* (CFG) is a set of recursive rules that describe how to form strings from a language's alphabet. Formally, a CFG is defined as $G = \langle V, \Sigma, R, S\rangle$, where V is a finite set of non-terminal characters, Σ is a finite set of terminal characters disjoint from V, R is a finite relation in $V \times (V \cup \Sigma)^*$, and S is the symbol in V that should be used as the *start rule* when using G to parse or generate a string. R defines the *rules* of the grammar and each rule is written as $V \rightarrow (V \cup \Sigma)^*$. A rule may be referred to by the non-terminal on its left side, and the right side of a rule is called the rule's *production.* A *straight-line grammar* (SLG) is a CFG that unambiguously produces exactly one string.

The CDAWG CFG

We use MR-CFG $= \langle V, \Sigma, R, S\rangle$ to denote the CFG derived from a CDAWG. A formal algorithm for constructing an MR-CFG from a CDAWG is not given in [5] so we sketch one in Algorithm 1 based on their description of the derivation. Note that in [5] the authors add a rule to MR-CFG for each character in Σ and the end character \$. Our MR-CFG construction algorithm naturally omits such rules and we do not add them. Any MR-CFG built using Algorithm 1 has the following properties.

Theorem 4. *An MR-CFG produces the same string that the CDAWG it was derived from recognizes the suffixes of [5].*

Algorithm 1: Construct MR-CFG from CDAWG

```
   Input: string $T$
   Output: MR-CFG
1  Construct CDAWG from $T$
2  Construct $\overline{\text{CDAWG}}$ by reversing the direction of all the edges in the CDAWG
3  begin Compact $\overline{\text{CDAWG}}$
4      Remove every node $v$ with out-degree 1
5      Redirect $v$'s in-neighbors to $v$'s out-neighbor
6      Concatenate the label of $v$'s out-edge to the label of each in-edge
7  end
8  begin Build MR-CFG from the compacted $\overline{\text{CDAWG}}$
9      All nodes except the sink are rules in the grammar
10     The out-neighbors of each node are the characters in its rule's production,
         ordered by their occurrence in the string
11     Out-edges to the sink node add the first character of the edge's label to
         the rule's production
12     The source node is the start rule
13 end
```

Corollary 4.1. *An MR-CFG has size $\mathcal{O}(e)$, where e is the number of edges in the CDAWG it was derived from. Equivalently, an MR-CFG has size $\mathcal{O}(n)$. [5]*

Theorem 5. *The set of non-start rules in R correspond to a subset of the maximal repeats in the string the MR-CFG of R produces.*

Proof. Algorithm 1 derives the set of rules R from the non-sink nodes of the compacted $\overline{\text{CDAWG}}$ (lines 8–13). $\overline{\text{CDAWG}}$ has the same nodes as CDAWG (line 2), which, by Corollary 3.1, means the internal nodes correspond to the maximal repeats of the string encoded by CDAWG. The compaction of $\overline{\text{CDAWG}}$ removes internal nodes while preserving all path labels (lines 3–7), meaning nodes that are not removed still correspond to maximal repeats. Therefore, the internal nodes of the compacted $\overline{\text{CDAWG}}$ from which the set of non-start rules R are derived correspond to a subset of the maximal repeats of the string the MR-CFG of R produces. □

As such, the "MR" of "MR-CFG" stands for "Maximal Repeat." Although there is a correspondence between the non-start rules of an MR-CFG and the maximal repeats of the string it produces, the rules do not necessarily produce the maximal repeats they correspond to.

Corollary 5.1. *The production of every non-start rule in R is left-maximal.*

Proof. By Lemma 3, the longest path label from the source to each internal node of a CDAWG represents a maximal repeat. This means that the edge label of the last edge in an internal node's maximal repeat path must have the leftmost occurrence of any of the node's in-edge labels. By Theorem 5, every non-start rule in R corresponds to a maximal repeat, specifically, the maximal repeat defined by the longest path label of the rule's CDAWG node before the CDAWG was reversed (line 2) and compacted (lines 3–7). Since the reversal and compaction steps preserve the path labels of the CDAWG (lines 2 and 6), it follows that the out-edge label of each internal node in the compacted $\overline{\text{CDAWG}}$ with the leftmost occurrence is still in the node's (reversed) maximal repeat path. The production of a rule in R is made by adding characters for the rule's compacted $\overline{\text{CDAWG}}$ node in leftmost-first order (line 10). Therefore, the leftmost character of the production of every non-start rule in R is either the leftmost character in the rule's corresponding maximal repeat or the non-terminal of a rule in R whose production prefixes the maximal repeat, thus preserving the production's left-maximality. □

Corollary 5.2. *The size of V is $\mathcal{O}(m)$, where m is the number of maximal repeats in the string the MR-CFG produces (proof omitted).*

Algorithms and Data Structures

In this section, we introduce novel algorithms and data-structures for computing the MR-CFG. Complexity analyses do not consider the data-structure used to compute LCP-intervals because these structures may vary asymptotically.

Interval Stabbing

Our MR-CFG construction algorithm requires the ability to perform stabbing queries on the LCP-intervals of a string. To perform these queries efficiently we augment the LCP-interval tree with a bit vector and a lookup table. Specifically, the bit vector B has length n and for every LCP-interval l-$[i..j]$ bits i and $j+1$ are set to 1; all other bits are set to 0. Similarly, for every LCP-interval l-$[i..j]$ lookups for i and $j+1$ are added to L. Lookup i points to the LCP-interval tree node with the largest LCP-value that starts at i. And lookup $j+1$ points to the parent of the LCP-interval tree node with the largest LCP-value that ends at j. If $j_1+1=i_2$ for two LCP-intervals l_1-$[i_1..j_1]$ and l_2-$[i_2..j_2]$ then the lookup points to the node that starts at i_2. If there is no parent for a $j+1$ lookup to point to then the pointer is set to the null value $\emptyset$. A stabbing query can then be performed with Algorithm 2. An example of this data structure is shown in Figure 1b.

Since the length of B is n and a string has $\mathcal{O}(n)$ LCP-intervals, B and L can be computed in $\mathcal{O}(n)$ time. B can be preprocessed in $\mathcal{O}(n)$ time to support $\mathcal{O}(1)$ time *rank* and *select* queries [10], thus, lines 1–6 of Algorithm 2 take $\mathcal{O}(1)$ time. And lines 7–10 take $\mathcal{O}(k)$ time, where k is the number of LCP-intervals stabbed, making the time complexity of Algorithm 2 $\mathcal{O}(1+k)$ after $\mathcal{O}(n)$ preprocessing time. We observe that the preprocessing step of Algorithm 2 is not strictly necessary and that the interval stabbing data-structure can be used while it is being constructed, i.e. *online*. This can be done by omitting B and using a sorted lookup table for L. Insertions and lookups would then be performed in $\mathcal{O}(\log n)$ time using binary search, meaning it would take $\mathcal{O}(n \log n)$ total time to build L and the time complexity of Algorithm 2 would be $\mathcal{O}(\log n + k)$.

Algorithm 2: LCP-interval stabbing query

```
   Input: suffix array index $i$
   Output: LCP-interval iterator
   Data: bit vector $B$
   Data: lookup table $L$
1  $r = rank(B, i)$
2  $p = select(B, r)$
3  node = $\emptyset$
4  if $L.hasEntry(p)$ then
5      node = $L[p]$
6  end
7  while $node \mathrel{!}= \emptyset$ do
8      $yield\ node$
9      $node = node.parent$
10 end
```

MR-CFG Construction

Internal nodes of a CDAWG represent isomorphic subtrees in a suffix tree (Lemma 2). Since the longest path label from the CDAWG source node to each internal node represents a maximal repeat (Lemma 3) and these paths form a spanning tree of the CDAWG (Corollary 3.2), the isomorphic subtrees from which all other edges of the CDAWG are derived must correspond to suffixes of the maximal repeats, excluding the sink node. In other words, all in-edges of an internal CDAWG node can be computed from the node's maximal repeat and its non-maximal suffixes by identifying the longest maximal repeats that prefix them. From this insight, we observe that because there is a bijection between the LCP-interval tree and the suffix tree (Theorem 1), a CDAWG can be computed from the LCP-intervals of a string. Furthermore, if the LCP-intervals are iterated in shortest-LCP-value-first order then the CDAWG will be computed in a breadth-first manner, allowing the compaction step of Algorithm 1 (lines 3–7) to be performed as the CDAWG is computed. This enables the MR-CFG to be computed in place of the CDAWG while skipping the edge reversal step of Algorithm 1 (line 2). Our MR-CFG construction algorithm is given in Algorithm 3.

Concisely, Algorithm 3 works by iterating LCP-intervals in shortest-LCP-value-first order (line 4). Whenever an LCP-interval is identified as a maximal repeat (line 9), a rule is added to the MR-CFG (line 10). The rule's production is computed by iterating the suffixes of its repeat (line 12). If a suffix is prefixed by a rule that has already been computed, then the rule's non-terminal character is added to the production (lines 13–17). Otherwise, the terminal character the suffix starts with is added instead (lines 18–20). If the rule's production has more than one character (line 23) then its non-terminal character can be added to future rules (line 24), otherwise, it is removed from the MR-CFG (line 26).

Explicitly, Algorithm 3 exploits the fact that every MR-CFG rule corresponds to a maximal repeat (Theorem 5) and is left-maximal (Corollary 5.1). This means an MR-

Algorithm 3: Construct MR-CFG from LCP-intervals

```
Input: LCP-interval iterator $I$
Output: MR-CFG
Data: maximal repeat LCP-interval stabber $M$
1  $R = []$ // rules; an empty map
2  $R_{lengths} = []$ // rule production lengths; an empty map
3  $S = \emptyset$ // the start rule identifier
4  for $l$-$[i..j]$, $id \in I$ do
5      if $!R_{lengths}.hasEntry(id)$ then
6          $R_{lengths}[id] = 0$
7      end
8      $R_{lengths}[id] = R_{lengths}[id] + 1$
9      if $isMaximal(l\text{-}[i..j])$ then
10         $R[id] = ()$ // a rule production; an empty ordered list
11         $k = 0$
12         while $k < R_{lengths}[id]$ do
13             $p = getRelativeIndex(i, k)$
14             $id_p = M.stabDeepest(p)$
15             if $id_p \mathrel{!=} \emptyset$ then
16                 $R[id] \leftarrow id_p$ // append a non-terminal character
17                 $k = k + R_{lengths}[id_p]$
18             else
19                 $R[id] \leftarrow getCharAt(p)$ // append a terminal character
20                 $k = k + 1$
21             end
22         end
23         if $|R[id]| > 1$ then
24             $M.update(l\text{-}[i..j], id)$
25         else
26             $R = R \setminus id$ // CDAWG edge compaction; omit useless rules
27             $R_{lengths} = R_{lengths} \setminus id$
28         end
29         if $i == j$ then
30             $S = id$
31         end
32     end
33 end
34 return $\langle R, S \rangle$
```

CFG rule and its maximal repeat will always have the same LCP-interval $[i..j]$, even if the string produced by the rule is shorter than its maximal repeat's LCP-value. This allows the in-edges of an internal CDAWG node, and therefore the characters of its MR-CFG rule's production, to be computed via LCP-interval stabbing queries using our interval stabbing data-structure ($M.stabDeepest(.)$ on line 14). Furthermore, since each CDAWG in-edge only has one source node, only one LCP-interval needs to be computed for each edge, namely, the LCP-interval stabbed with the largest LCP-value, which is the first interval returned by Algorithm 2.

For clarity, Algorithm 3 uses identifiers for maximal repeats and their corresponding MR-CFG non-terminal characters. Specifically, each LCP-interval has an identifier id that gets passed to its left extensions until a maximal repeat LCP-interval. Then a new id is used for the left extensions of the maximal repeat, and so on. For simplicity, Algorithm 3 assumes that the LCP-interval iterator I includes an interval for every suffix in the string, starting with interval 1-[0..0] for the end character \$. By treating the LCP-interval with LCP-value n as maximal, this allows the start rule to be computed using the same code as the other rules. The start rule identifier can be easily distinguished from the other rules' identifiers because no other maximal LCP-interval will begin and end on the same index (lines 29–31).

Note that Algorithm 3 does not return a set of non-terminal characters V or a set of terminal characters Σ because they can be computed by iterating R. Additionally, operations that depend on how LCP-intervals are computed are performed using helper functions: $isMaximal(.)$ (line 9) determines if an LCP-interval represents a maximal repeat; $getRelativeIndex(i, k)$ (line 13) computes the index of the suffix that starts k characters after the suffix at index i; and $getCharAt(p)$ gets the alphabet character at index p. Finally, note that the method $M.update(.)$ (line 24) updates the interval stabbing data-structure M so that an LCP-interval may be returned by a stabbing query. In the case that M was built and preprocessed before Algorithm 3, the bit vector B and lookup table L will already be populated so care will need to be taken to only return intervals added by $M.update(.)$. For example, maximal repeat identifiers could be treated as bit vectors that reflect the structure of the LCP-interval tree, as depicted in Figure 1b. This would allow a stabbed node's nearest added ancestor to be identified using bitwise operations, which can be computed efficiently on modern processors.

Since there are $\mathcal{O}(n)$ LCP-intervals, the main loop (lines 4–33) will iterate $\mathcal{O}(n)$ times. The production construction loop (lines 12–22) will iterate as many times total as there are edges in the CDAWG — $\mathcal{O}(n)$. $M.stabDeepest(.)$ (line 14) will take $\mathcal{O}(1)$ time if M is preprocessed and $\mathcal{O}(\log m)$ time if M is computed online since only maximal LCP-intervals are being stabbed. And $M.update(.)$ (line 24) will take $\mathcal{O}(1)$ time if M is preprocessed and $\mathcal{O}(\log m)$ time if M is computed online. This makes the overall time-complexity of Algorithm 3 $\mathcal{O}(n)$ if M is preprocessed and $\mathcal{O}(n \log m)$ if M is computed online.

Lastly, since Algorithm 3 only requires the deepest stabbed LCP-interval, M need not explicitly store the LCP-interval tree. This means M only requires $2\mathcal{O}(m)$ words for L, and n bits for B if M is preprocessed. R requires $\mathcal{O}(m)$ words for non-terminal characters and $\mathcal{O}(n)$ words total for their productions. And $R_{lengths}$ requires $2\mathcal{O}(m)$ words. In total, Algorithm 3 requires $\mathcal{O}(n) + 5\mathcal{O}(m)$ words of space if M is computed online, including the size of the computed MR-CFG. If M is preprocessed, then n additional bits are required for B and $\mathcal{O}(m)$ words are required for identifiers.

Conclusion

We provided a generic algorithm for computing the MR-CFG from a string's LCP-intervals, connecting the MR-CFG to properties of the string it produces and relating it to other string data-structures. This allows the MR-CFG to be studied independently of the CDAWG and provides opportunity for innovation of grammar-based compression algorithms. We observe that the MR-CFG is reducible, leaving room for further improvement. A reference implementation of our MR-CFG construction algorithm is available at `https://github.com/alancleary/mr-cfg`.

Acknowledgements

This work was supported by NSF Award Number 2105391.

References

[1] Katrin Casel, Henning Fernau, Serge Gaspers, Benjamin Gras, and Markus L. Schmid, "On the complexity of the smallest grammar problem over fixed alphabets," *Theory of Computing Systems*, vol. 65, no. 2, pp. 344–409, Nov. 2020.

[2] N Jesper Larsson and Alistair Moffat, "Off-line dictionary-based compression," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[3] Dominik Kempa and Nicola Prezza, "At the roots of dictionary compression: String attractors," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, New York, NY, USA, 2018, STOC 2018, p. 827–840, Association for Computing Machinery.

[4] Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida, "MR-RePair: Grammar compression based on maximal repeats," in *2019 Data Compression Conference (DCC)*. Mar. 2019, pp. 508–517, IEEE.

[5] Djamal Belazzougui and Fabio Cunial, "Representing the Suffix Tree with the CDAWG," in *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, Eds., Dagstuhl, Germany, 2017, vol. 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 7:1–7:13, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004, The 9th International Symposium on String Processing and Information Retrieval.

[7] Timo Beller, Katharina Berger, and Enno Ohlebusch, "Space-efficient computation of maximal and supermaximal repeats in genome sequences," in *String Processing and Information Retrieval*, Liliana Calderón-Benavides, Cristina González-Caro, Edgar Chávez, and Nivio Ziviani, Eds., Berlin, Heidelberg, 2012, pp. 99–110, Springer Berlin Heidelberg.

[8] Takaaki Nishimoto and Yasuo Tabei, "R-enum: Enumeration of characteristic substrings in bwt-runs bounded space," 2020.

[9] Mathieu Raffinot, "On maximal repeats in strings," *Information Processing Letters*, vol. 80, no. 3, pp. 165–169, 2001.

[10] Francisco Claude and Gonzalo Navarro, "Practical rank/select queries over arbitrary sequences," in *String Processing and Information Retrieval*, Amihood Amir, Andrew Turpin, and Alistair Moffat, Eds., Berlin, Heidelberg, 2009, pp. 176–187, Springer Berlin Heidelberg.