SEA2025

Bit Packed Encodings for Grammar-Compressed Strings Supporting Fast Random Access

Alan M. Cleary, Joseph Winjum, Jordan Dood
Hiroki Shibata, and Shunsuke Inenaga

Grammar-Based Compression

- **Compress a string by an admissible context-free grammar that produces only that string**
- $G = \langle X, \Sigma, R, S \rangle$ **: a grammar**
  - X : nonterminals, Σ : terminals, $S \in X$: start symbol
  - R defines the set of production rules $\{X_i \rightarrow expr_i \mid 1 \leq i \leq m\}$

Σ : {a, b}
X : {A, B, W, X, Y, Z}
S : Z

R

A → a	B → b
W → AB	X → AA
Y → WB	Z → XY

Three Types of Grammars

We consider three types of grammars:

- **Straight-Line Programs (SLP)**

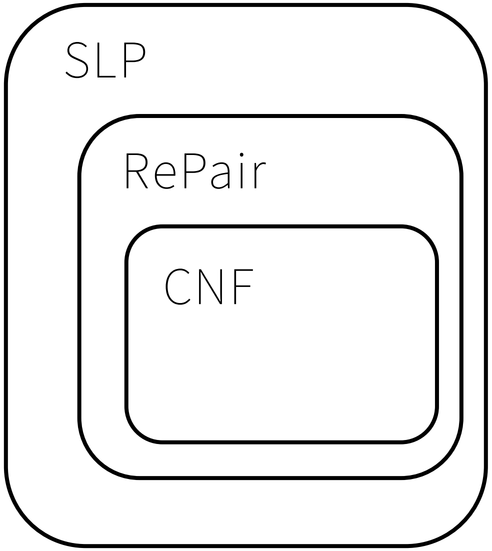
  An admissible grammar that generates exactly one string

- **Chomsky Normal Form (CNF)**

  Each rule (including the start rule) is of the form $X_i \to c$ or $X_i \to X_l X_r$

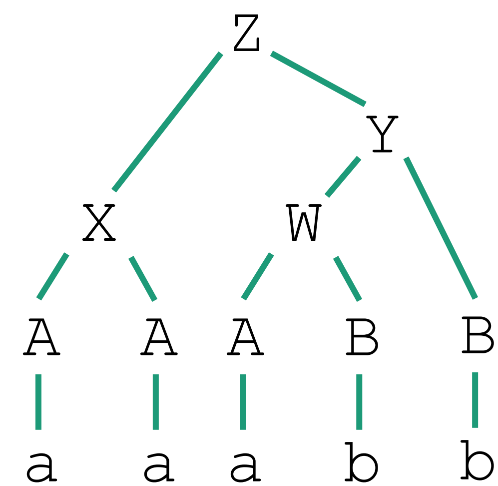
- **RePair Grammar**

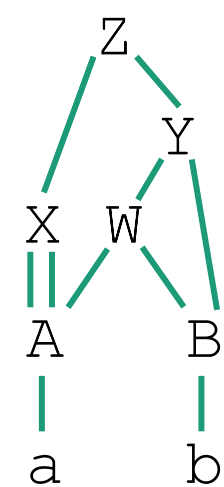
  A CNF grammar where the start rule may contain an arbitrary number of symbols on its right-hand side



Derivation Tree and Derivation DAG

- **Production rules naturally define the tree derivates the string (derivation tree)**
  - The root corresponds to the start symbol
  - Leaves represent the string
- **derivation DAG : A DAG obtained by merging all isomorphic subtrees of derivation tree**





Σ : {a, b}
X : {A, B, W, X, Y, Z}
S : Z

R

A → a	B → b
W → AB	X → AA
Y → WB	Z → XY

Random Access Problem

- **Preprocessing: Given a text T, construct an index supporting the following query**
- **Query:**
  - Input: A position p $(1 \leq p \leq |T|)$
  - Output: the p-th character $T[p]$
- **Evaluation Criteria:**
  - The space of preprocessed index
  - The query time

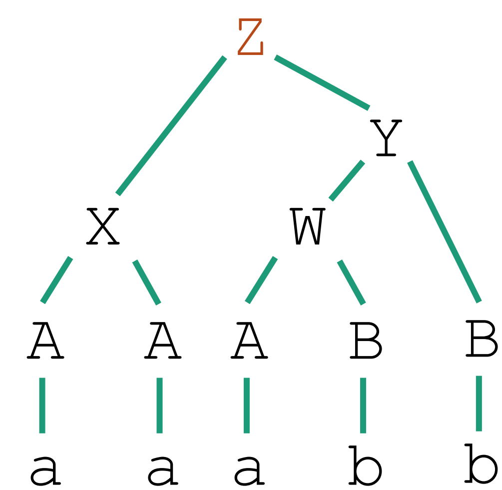
Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
   1. Compute the subtree size s of the left child of v
   2. If $s < p$, set $p \leftarrow p - s$ and move v to the right child

      Otherwise, move v to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



$v = \mathrm{Z}$

$p = 3$

$\Sigma : \{\mathrm{a}, \mathrm{b}\}$

$X : \{\mathrm{A}, \mathrm{B}, \mathrm{W}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$

$S : \mathrm{Z}$

R

$\mathrm{A} \rightarrow \mathrm{a}$	$\mathrm{B} \rightarrow \mathrm{b}$
$\mathrm{W} \rightarrow \mathrm{AB}$	$\mathrm{X} \rightarrow \mathrm{AA}$
$\mathrm{Y} \rightarrow \mathrm{WB}$	$\mathrm{Z} \rightarrow \mathrm{XY}$

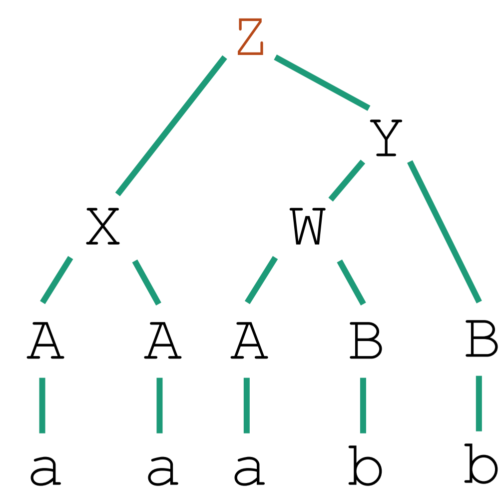
Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
   1. Compute the subtree size $\boldsymbol{s}$ of the left child of $\boldsymbol{v}$
   2. If $\boldsymbol{s} < \boldsymbol{p}$, set $\boldsymbol{p} \leftarrow \boldsymbol{p} - \boldsymbol{s}$ and move $\boldsymbol{v}$ to the right child

      Otherwise, move $\boldsymbol{v}$ to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



$\Sigma : \{\mathtt{a}, \mathtt{b}\}$

$X : \{\mathtt{A}, \mathtt{B}, \mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$

$S : \mathtt{Z}$

R

$\mathtt{A} \rightarrow \mathtt{a}$	$\mathtt{B} \rightarrow \mathtt{b}$
$\mathtt{W} \rightarrow \mathtt{AB}$	$\mathtt{X} \rightarrow \mathtt{AA}$
$\mathtt{Y} \rightarrow \mathtt{WB}$	$\mathtt{Z} \rightarrow \mathtt{XY}$

$v = \mathtt{Z}$

$p = 3$

$s = 2$

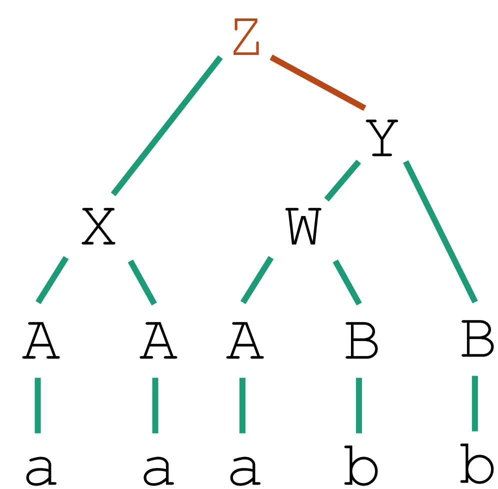
Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
    1. Compute the subtree size $\boldsymbol{s}$ of the left child of $\boldsymbol{v}$
    2. If $\boldsymbol{s} < \boldsymbol{p}$, set $\boldsymbol{p} \leftarrow \boldsymbol{p} - \boldsymbol{s}$ and move $\boldsymbol{v}$ to the right child

       Otherwise, move $\boldsymbol{v}$ to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



Σ : {a, b}

X : {A, B, W, X, Y, Z}

S : Z

R

A → a   B → b

W → AB   X → AA

Y → WB   Z → XY

v = Z

$p = 3$

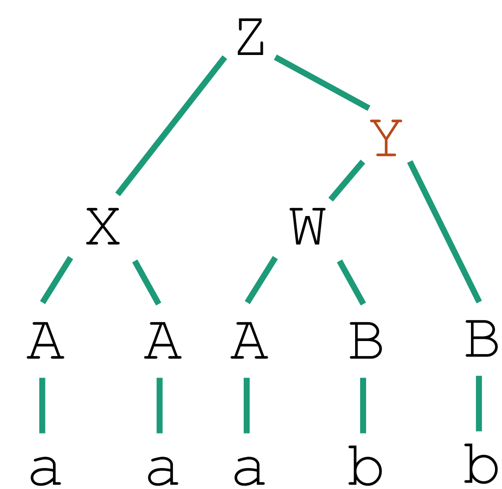
$s = 2$

Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
   1. Compute the subtree size s of the left child of v
   2. If $s < p$, set $p \leftarrow p - s$ and move v to the right child
      Otherwise, move v to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



$v = \mathrm{Y}$

$p = 1$

Σ : {a, b}

X : {A, B, W, X, Y, Z}

S : Z

R

A → a　B → b

W → AB　X → AA

Y → WB　Z → XY

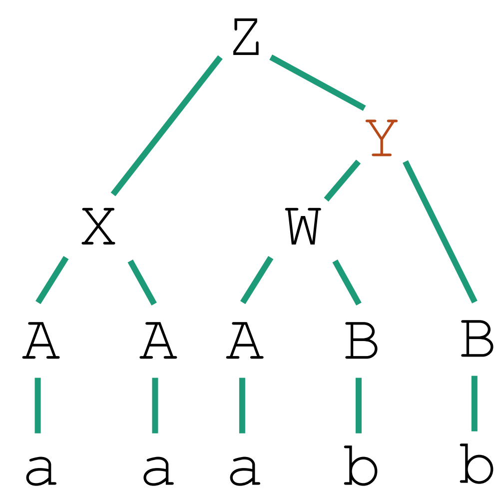
Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
    1. Compute the subtree size s of the left child of v
    2. If $s < p$, set $p \leftarrow p - s$ and move v to the right child

       Otherwise, move v to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



Σ : {a, b}

X : {A, B, W, X, Y, Z}

S : Z

R

A → a   B → b

W → AB   X → AA

Y → WB   Z → XY

$v =$ Y

$p = 1$

$s = 2$

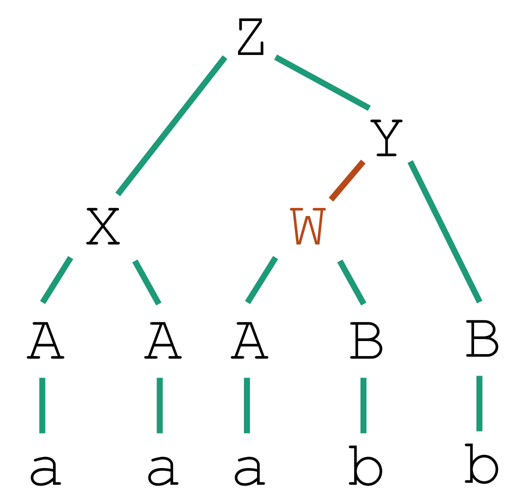
Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
    1. Compute the subtree size $\boldsymbol{s}$ of the left child of $\boldsymbol{v}$
    2. If $\boldsymbol{s} < \boldsymbol{p}$, set $\boldsymbol{p} \leftarrow \boldsymbol{p} - \boldsymbol{s}$ and move $\boldsymbol{v}$ to the right child

       Otherwise, move $\boldsymbol{v}$ to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



$v = \mathtt{W}$

$p = 1$

$\Sigma : \{\mathtt{a}, \mathtt{b}\}$

$X : \{\mathtt{A}, \mathtt{B}, \mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$

$S : \mathtt{Z}$

R

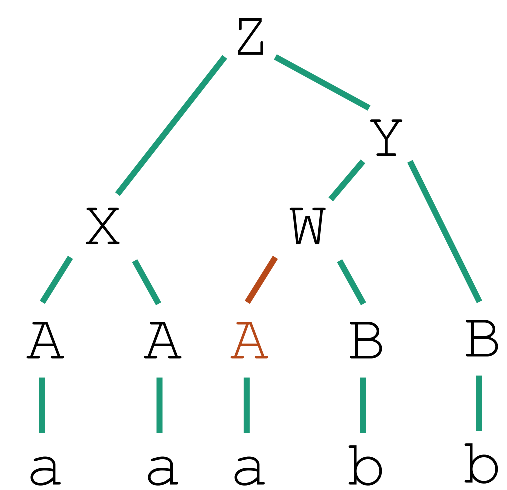
$\mathtt{A} \rightarrow \mathtt{a}$	$\mathtt{B} \rightarrow \mathtt{b}$
$\mathtt{W} \rightarrow \mathtt{AB}$	$\mathtt{X} \rightarrow \mathtt{AA}$
$\mathtt{Y} \rightarrow \mathtt{WB}$	$\mathtt{Z} \rightarrow \mathtt{XY}$

Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
    1. Compute the subtree size s of the left child of v
    2. If $s < p$, set $p \leftarrow p - s$ and move v to the right child
       Otherwise, move v to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



$v = \mathrm{A}$

$p = 1$

$\Sigma : \{\mathrm{a}, \mathrm{b}\}$

$X : \{\mathrm{A}, \mathrm{B}, \mathrm{W}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$

$S : \mathrm{Z}$

R

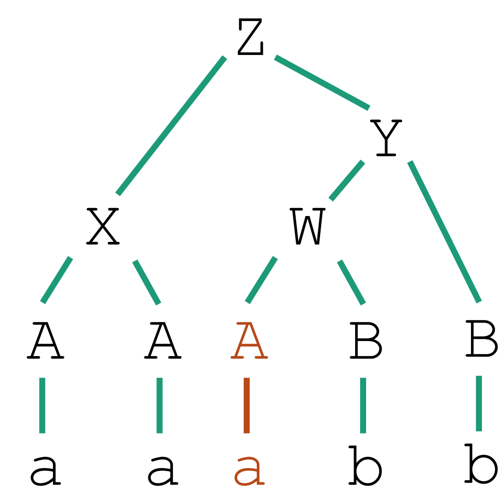
$\mathrm{A} \rightarrow \mathrm{a}$	$\mathrm{B} \rightarrow \mathrm{b}$
$\mathrm{W} \rightarrow \mathrm{AB}$	$\mathrm{X} \rightarrow \mathrm{AA}$
$\mathrm{Y} \rightarrow \mathrm{WB}$	$\mathrm{Z} \rightarrow \mathrm{XY}$

Random Access to Grammar-Compressed Strings

The algorithm for random access to $T[p]$ for a CFG:

1. **Set $v \leftarrow S$ (S : the start symbol)**
2. **Repeat the following procedure while the production rule corresponding to v is of the form $v \rightarrow X_l X_r$**
    1. Compute the subtree size s of the left child of v
    2. If $s < p$, set $p \leftarrow p - s$ and move v to the right child
       Otherwise, move v to the left child
3. **Return the character corresponding to v**

(This algorithm can be modified for general SLPs)



$v = \mathtt{a}$

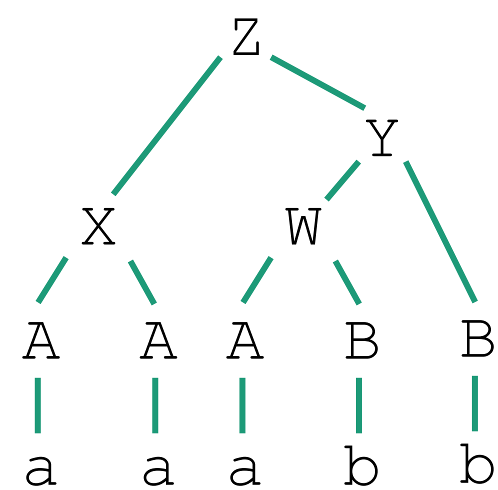
$\Sigma : \{\mathtt{a}, \mathtt{b}\}$
$X : \{\mathtt{A}, \mathtt{B}, \mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$
$S : \mathtt{Z}$

R

$\mathtt{A} \rightarrow \mathtt{a}$	$\mathtt{B} \rightarrow \mathtt{b}$
$\mathtt{W} \rightarrow \mathtt{AB}$	$\mathtt{X} \rightarrow \mathtt{AA}$
$\mathtt{Y} \rightarrow \mathtt{WB}$	$\mathtt{Z} \rightarrow \mathtt{XY}$

Random Access to Grammar-Compressed Strings

- **Time complexity:** $O(h)$
  - h : the height of the grammar
- **Space complexity:** $O(|G|)$
  - $|G|$: the total number of symbols on the right-hand side of the rules
- **Required operations:**
  - Computing the size of string generated by each nonterminal
    - we assume that these values are precomputed
  - Efficient access to the right-hand side symbols for each rule



Σ : {a, b}

X : {A, B, W, X, Y, Z}

S : Z

R

A → a B → b

W → AB X → AA

Y → WB Z → XY

Representing Grammars

- **We assume that the input is a CNF grammar with σ rules of the form $X_i \to c$ and m rules of the form $X_i \to X_l X_r$**
  - Our method also works for RePair grammar
- **We encode each nonterminal as an integer**
  - We number all nonterminals so that the number of each right-hand side symbol is smaller than that of left-hand side symbol

R

A → a	B → b
W → AB	X → AA
Y → WB	Z → XY

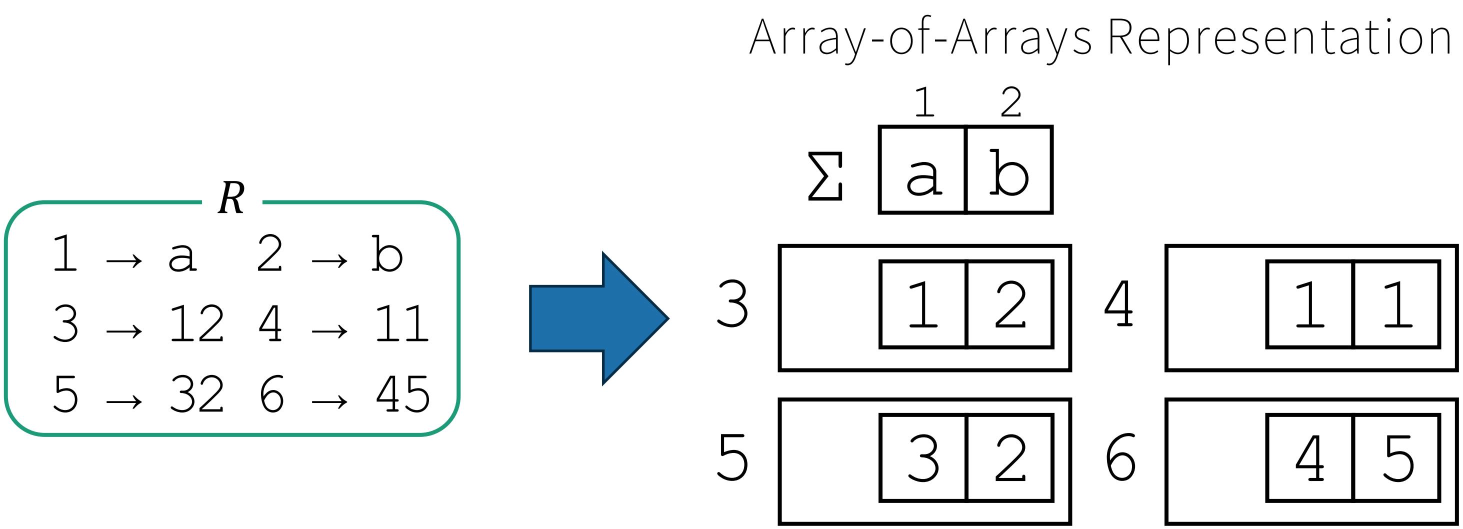
R

1 → a	2 → b
3 → 12	4 → 11
5 → 32	6 → 45

The Basic Array-of-Arrays Representation

Canonical representation: array-of-arrays

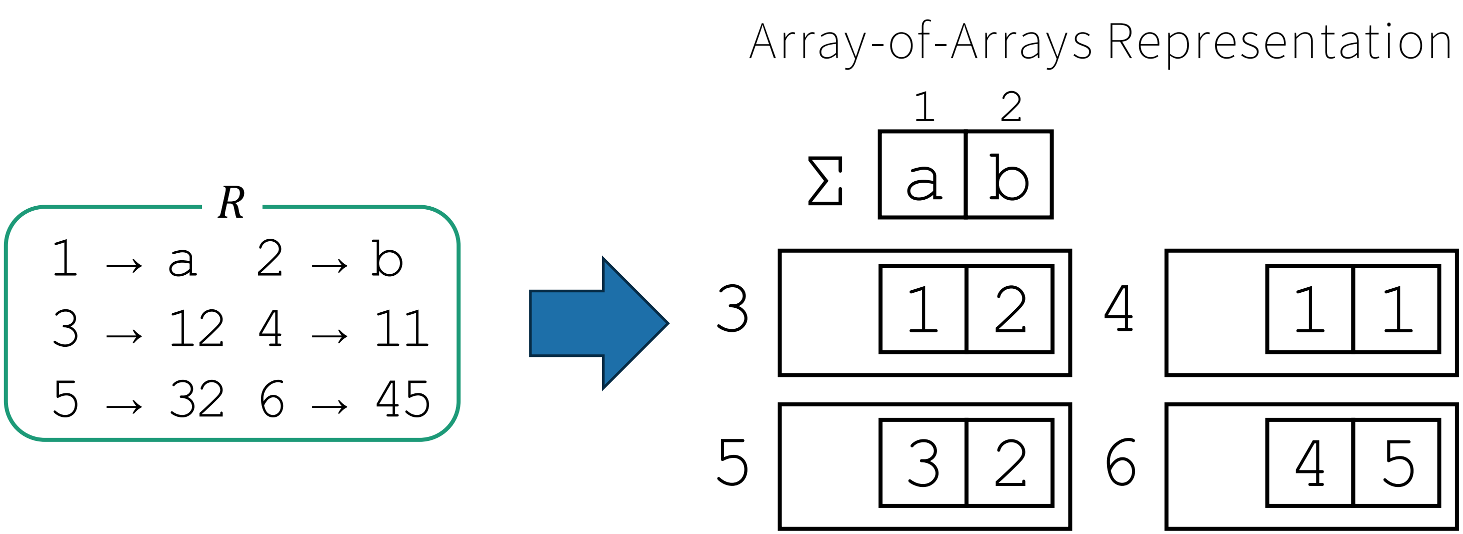
- **The right-hand side symbols of each rule of the form $X_i \rightarrow X_l X_r$ are represented by an array**



The Basic Array-of-Arrays Representation

The basic array-of-arrays representation:

- **Pro: Fast look-up operation**
  - Supports constant time access to the right-hand symbols of the rule
- **Con: Space consumption**
  - It uses at least $\sum_{i=1}^{m} |expr_i|$ words
  - Additional space is required to store pointers to each of the m arrays



Bit Packing Technique

A technique for encoding data in small space by packing the raw bits into contiguous blocks of memory

- **Each value in the packed array can be accessed in constant time**

`(1,4,4,2,4)` as a byte array with bit width 3

$$\underbrace{001}_{1}\underbrace{100}_{4}\underbrace{10\ \ 0}_{4}\underbrace{010}_{2}\underbrace{100}_{4}0$$

Bit Packing Grammar Compressed Strings

- **We encode each right-hand side symbols in a rule $X_i \rightarrow expr_i$ in the same bit width w_i**

- **Three strategies to determine the bit width w_i :**
  - Bit Pack Left (BPL)
  - Bit Pack Right (BPR)
  - Bit Pack Right Monotonic (BPRM)

Bit Pack Left (BPL) Strategy

- **We set w_i as $w_i = MSB(X_i)$**
  - This bit width is sufficient to represent the right-hand side symbols because $j < i$ holds for each X_i and $X_j \in expr_i$

R

1 → a	2 → b
3 → 12	4 → 11
5 → 32	6 → 45

$$w_3 = MSB(3) = MSB(011_2) = 2$$
$$w_4 = MSB(4) = MSB(100_2) = 3$$
$$w_5 = MSB(5) = MSB(101_2) = 3$$
$$w_6 = MSB(6) = MSB(110_2) = 3$$

Rule 3–6 as a BPL byte array

$$\overbrace{01}^{1}\overbrace{10}^{2}\overbrace{001}^{1}\overbrace{0\;01}^{1}\overbrace{011}^{3}\overbrace{010}^{2}$$
$$\underbrace{100}_{4}\underbrace{101}_{5}00\;\;00000000$$

Bit Pack Left (BPL) Strategy

- **We can access right-hand side symbols in constant time without storing offset values**
  - The bit position $s(i)$ where a given rule $X_i \rightarrow expr_i$ starts is

$$s(i) = 1 + \sum_{k=1}^{\sigma} \lfloor \log_2 k \rfloor + 2 \sum_{j=\sigma+1}^{i-1} \lfloor \log_2 j \rfloor$$

This position can be computed in $O(1)$ time using MSB operation

Bit Pack Right (BPR) Strategy

- **We set w_i as $w_i = MSB\left(\max\{X_j \mid X_j \in expr_i\}\right)$**
- **A cumulative sum array of w_i is maintained**
  - It enables constant-time access for $\boldsymbol{expr_i}$
  - This array also allows restoring each $\boldsymbol{w_i}$

R

1 → a 2 → b
3 → 12 4 → 11
5 → 32 6 → 45

$$w_3 = MSB(\max\{1,2\}) = MSB(010_2) = 2$$
$$w_4 = MSB(\max\{1,1\}) = MSB(001_2) = 1$$
$$w_5 = MSB(\max\{3,2\}) = MSB(011_2) = 2$$
$$w_6 = MSB(\max\{4,5\}) = MSB(101_2) = 3$$

Rule 3–6 as a BPR byte array

$$\overbrace{0}^{1}\overbrace{11}^{2}\overbrace{0}^{1}\overbrace{1}^{1}\overbrace{111}^{3}\quad\overbrace{10}^{2}\overbrace{100}^{4}\overbrace{101}^{5}$$

Bit Pack Right Monotonic (BPRM) Strategy

- **We set** w_i **as** $w_i = \max\left\{MSB\left(\max\left\{X_j \mid X_j \in expr_i\right\}\right), w_{i-1}\right\}$
  - This strategy ensures monotonic bit widths (i.e., $\boldsymbol{w_{i-1} \leq w_i}$)
  - The number of unique values of $\boldsymbol{w_i}$ is at most $\lceil \log_2(m+\sigma) \rceil$

R

1 → a	2 → b
3 → 12	4 → 11
5 → 32	6 → 45

$$w_3 = MSB(010_2) = 2$$
$$w_4 = \max\{MSB(001_2), 2\} = 2$$
$$w_5 = \max\{MSB(011_2), 2\} = 2$$
$$w_6 = \max\{MSB(101_2), 2\} = 3$$

Rule 3–6 as a BPRM byte array

1 2 1 1 3 2 4 5

10011101 00110010

01000000 00000000

Bit Pack Right Monotonic (BPRM) Strategy

- **We store the following data structures:**
  - A bitvector BV such that $BV[i] = 1$ holds iff $w_i < w_{i+1}$, supporting rank/select queries
  - A unique bit-width array w'

- **Using the above data structures, $expr_i$ can be accessed in constant time**

Analysis of Space Complexity

We analyze the bit count required by the BPL strategy

- Given a CNF grammar with $n = m + \sigma$ total symbols, the total number of bits is

$$1 + \sum_{k=1}^{\sigma} \lfloor \log_2 k \rfloor + 2 \sum_{j=\sigma+1}^{n} \lfloor \log_2 j \rfloor = 1 + 2 \sum_{j=1}^{n} \lfloor \log_2 j \rfloor - \sum_{k=1}^{\sigma} \lfloor \log_2 k \rfloor$$

Analysis of Space Complexity

We analyze the bit count required by the BPL strategy

- **Using $\sum_{i=1}^{n} \log_2 i = \log_2(n!)$, we can obtain the upper bound :**

$$1 + 2\sum_{j=1}^{n} \lfloor \log_2 j \rfloor - \sum_{k=1}^{\sigma} \lfloor \log_2 k \rfloor < 2\log_2 n! - \log_2 \sigma!$$

  - Similarly, we can obtain the bound of a RePair grammar
  - The bound for BPR/BPRM can also be obtained by simply adding the additional term for effi cient random access

Optimality

- **The total bit usage for CNF can be approximated by :**

$$1 + 2\log_2 n! - \log_2 \sigma! \approx 2n\log_2 n - O(m) + O(\log n)$$

  - Analysis uses Stirling's approximation

- **The information-theoretic lower bound is :**

$$2n + \log_2 n! + o(n) \approx n\log_2 n + (2 - \log_2 e)n + o(n)$$

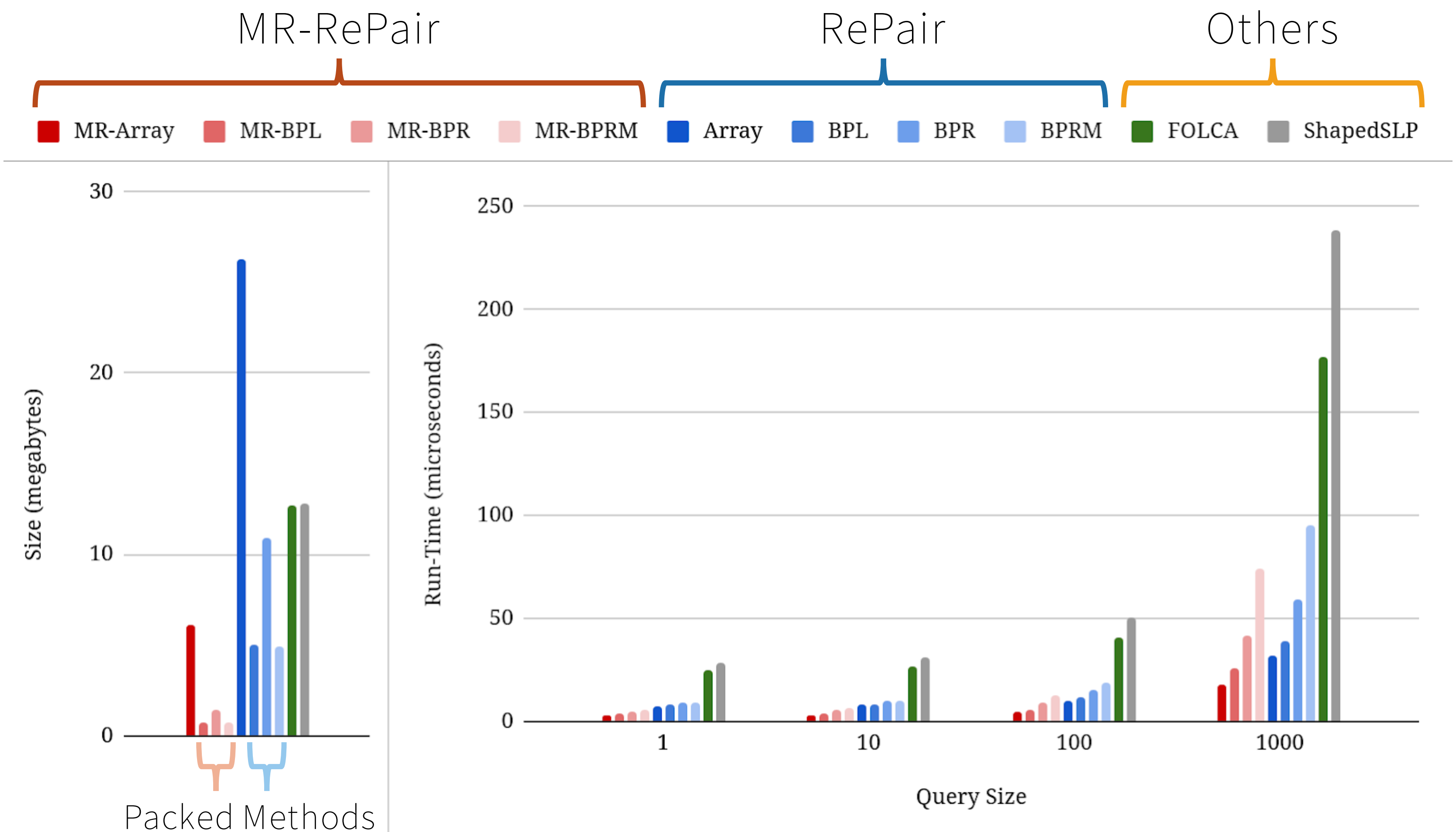
- **Ignoring lower-order terms, our bound is approximately**

  $n\log_2 n$ **bits above the information-theoretic lower bound**

Experiments

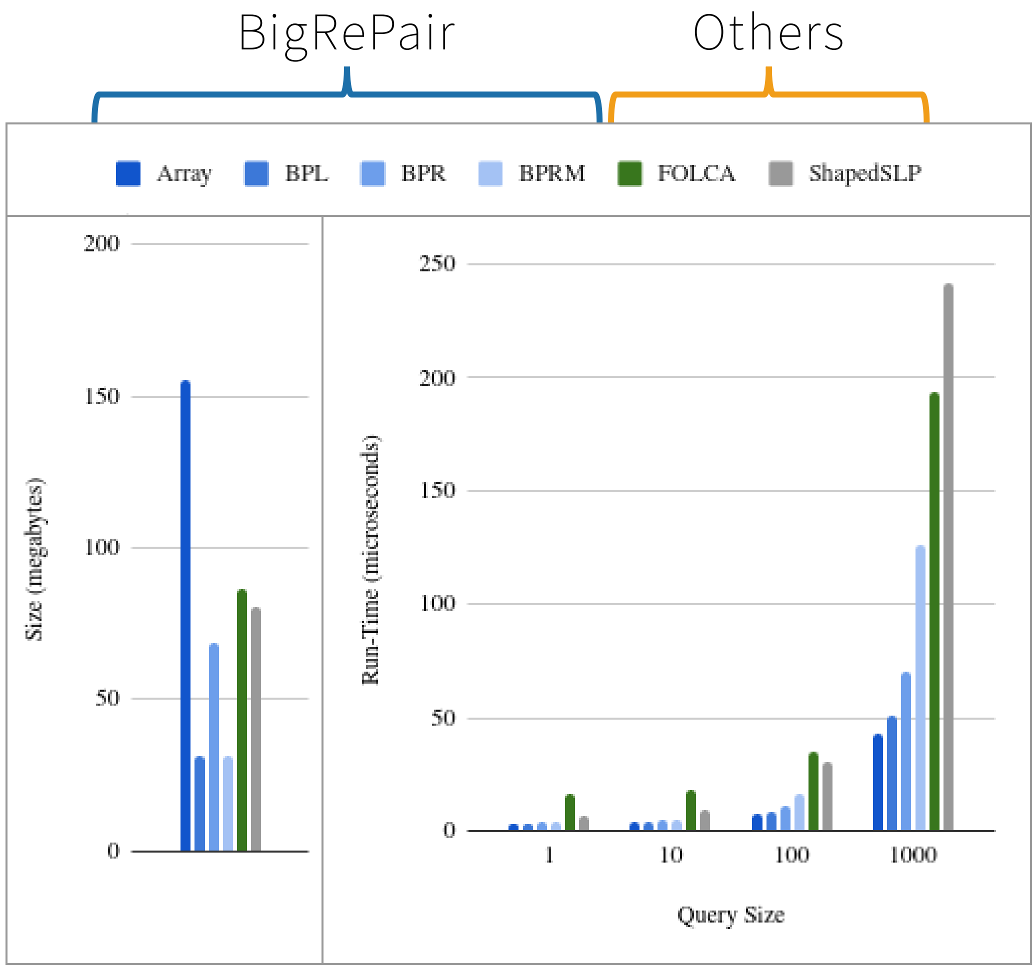
- **Extended Folklore Random Access for SLPs (FRAS) [Cleary et al., 2024] to support bit packing strategies**
- **Compared encoded grammar sizes and random access run-times against FOLCA [Maruyama et al., 2013] / ShapedSLP [Gagie et al., 2020]**
  - The length of extracted substring length: 1, 10, 100, 1000
- **Built grammars using 3 different algorithm on 2 corpora of data:**
  - Pizza & Chili
    - Re-Pair [Larsson and Moffat, 2000] and MR-RePair [Furuya et al., 2019]
  - Pangenomes (we show only the result of c1000 dataset)
    - We used BigRePair [Gagie et al., 2019] algorithm because the dataset is large

Results (Pizza&Chilli)



- Packed methods achieved significant space saving (particularly BPL and BPRM)
- Non-packed methods was the fastest, but packed methods were competitive with the non-packed method
- Packed methods were both smaller and faster than FOLCA / Shaped SLP

Results (c1000 dataset, BigRePair)



- Packed methods achieved significant space saving (particularly BPL and BPRM)
- Non-packed methods was the fastest, but packed methods were competitive with the non-packed method
- Packed methods were both smaller and faster than FOLCA / Shaped SLP

Conclusions

- **Bit-packed encodings are more space-efficient in practice**
  - Our methods also have theoretical analysis compared to the information-theoretic lower bound
- **Bit packed encodings support fast random access**
  - The encodings preserve the array-of-arrays good memory locality
- **Bit packed encodings preserve the benefits of array-of-arrays representations**