SPIRE2024

LZ78 Substring Compression in CDAWG-compressed space

Hiroki Shibata[1], Dominik Köppl[2]

[1]. Kyushu University

[2]. University of Yamanashi

Substring Compression Problem [Cormode et al., 2005]

- **The compression method C is specified.**
- **The text T of length n is given <u>in advance</u>.**
  - We can construct an index of T before the queries.
- **Queries:**
  - Input: two integers l, r $(1 \leq l \leq r \leq n)$
  - Output: The compressed representation $C(T[l..r])$.

$$T = \text{acbaababaacaab}, (l, r) = (6, 10)$$

→ Output is $C(\text{baaba})$.

Substring Compression Problem [Cormode et al., 2005]

- **The text T of length n is given in advance.**
  - We can construct an index of T before the queries.
- **Queries:**
  - Input: two integers l, r $(1 \leq l \leq r \leq n)$
  - Output: The compressed representation $C(T[l..r])$.

- **Natural solution: use compression method for each query substring.**
  - It takes at least $O(r - l + 1)$ time.
- **Efficient solution: create an index for substring compression.**

The goal is to output $C(T[l..r])$ in time close to optimal $O(|C(T[l..r])|)$.

Substring Compression Problem [Cormode et al., 2005]

- **Natural solution**: use compression method for each query substring.
  - It takes at least $O(r - l + 1)$ time.
- **Efficient solution**: create an index for substring compression.

The goal is to output $C(T[l..r])$ in time close to optimal $O(C(T[l..r])|)$.

	Space for Index	Query Time
LZ78 [Köppl, '21]	$O(n)$	$O(c)$
LZ77 [Keyller et al., '14]	$O(n)$	$O(c \log^{\varepsilon} n)$
LZ77 [Keyller et al., '14]	$O(n^{1+\varepsilon})$	$O(c)$

($c = |C(T[l..r])|$ is the length of output)

LZ78 Factorization [Ziv & Lempel, '78]

The LZ78 factorization of a string T is a factorization

$$T = F_0 F_1 \dots F_f$$

T' is the unfactorized part of T.

where F_i $(i \geq 1)$ is the longest prefix of $T' = F_i \dots F_f$ that can be represented by $F_i = F_j c$ using some $j < i$ and $c \in \Sigma$.

A concatenation of an already computed factor and a character.

$$T = \text{a b aa ba bac}$$

F_0 F_1 F_2 F_3 F_4 F_5

F_0a F_0b F_1a F_2a F_4c

($F_0 = \varepsilon$ is the empty string)

Previous Work

We focus on LZ78 substring compression problem.

Recently, [Köppl, '21] proposed a method for LZ78 substring compression.

- **This method is time optimal but consumes $O(n)$ words of space.**

Time/Space Complexity

	Space for Index	Working Space	Preprocessing Time	Query Time
[Köppl, '21]	$O(n)$	$O(c)$	$O(n)$	$O(c)$

($c = |C(T[l..r])|$ is the length of the output.)

Our Work

We propose a method for LZ78 substring compression in compressed space.

Time/Space Complexity

	Space for Index	Working Space	Preprocessing Time	Query Time
[Köppl, '21]	$O(n)$	$O(c)$	$O(n)$	$O(c)$
Ours	$O(e)$	$O(c)$	$O(n)$	$O(c \log n)$

e is the number of the edges of the CDAWG of T.

- $e \in O(n)$ holds for all strings.
- $e \in \Theta(\log n)$ for Fibonacci strings.

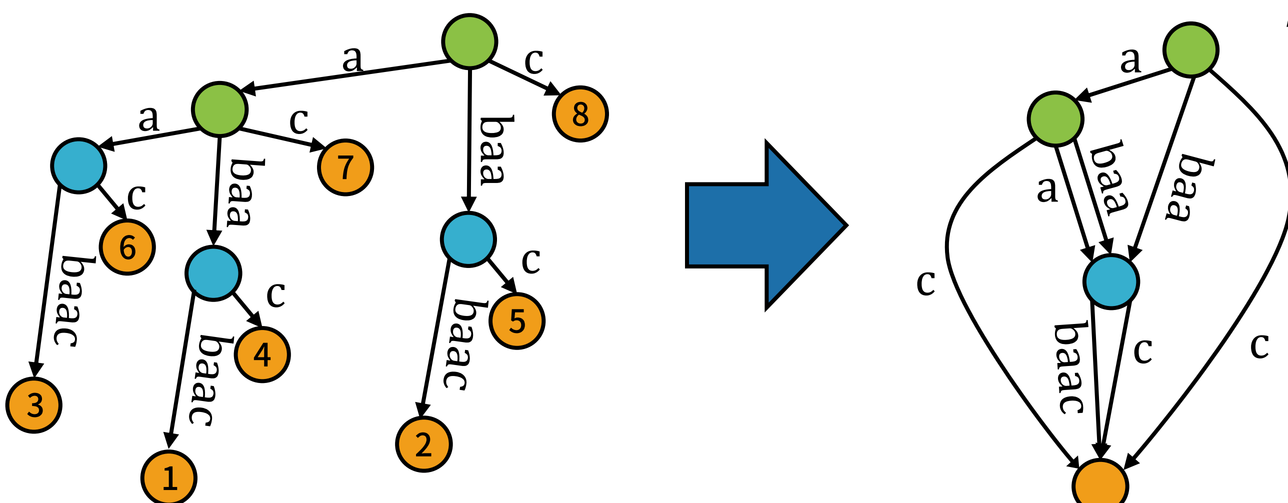
($c = |C(T[l..r])|$ is the length of the output.)

Our Work

	Space for Index	Working Space	Preprocessing Time	Query Time
[Köppl, '21]	$O(n)$	$O(c)$	$O(n)$	$O(c)$
Ours	$O(e)$	$O(c)$	$O(n)$	$O(c \log n)$

Previous method uses a suffix tree for an index.

Our method uses a CDAWG instead of a suffix tree.

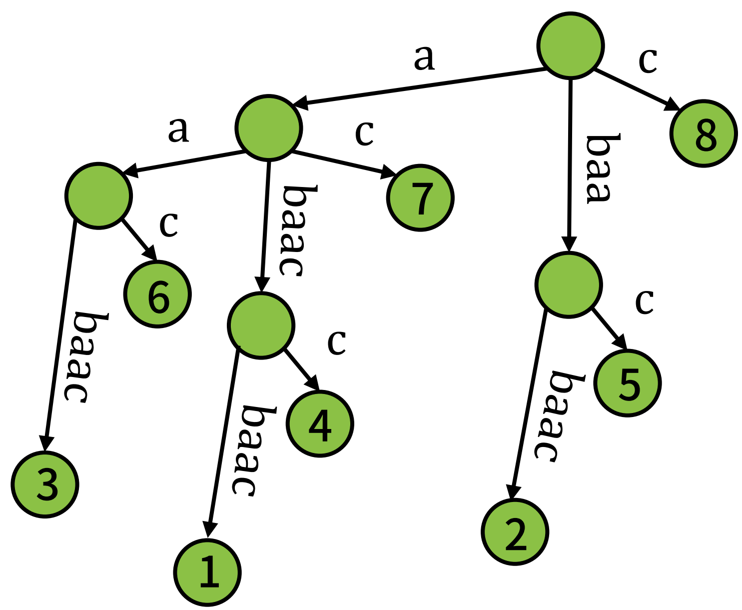


$T = \text{abaabaac}$

Computing LZ78 by suffix trees (previous method)

Suffix Trees (ST)

Suffix Tree: The compact trie representing all suffixes of T.



$$T = \text{abaabaac}$$

LZ78 Tries

LZ78 Trie: The trie consisting of all LZ78 factors of T.

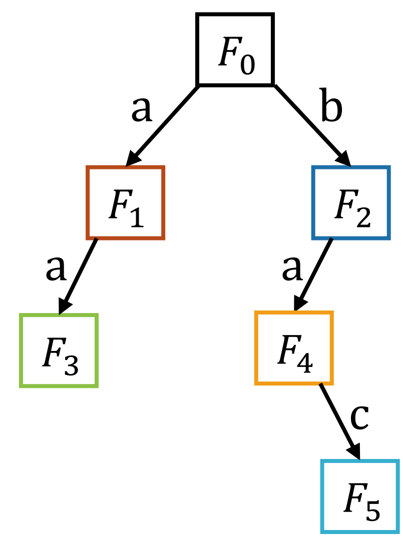
- **The nodes have one-to-one correspondence to the LZ78 factors.**
- **Computing LZ78 can be regarded as constructing a LZ78 trie.**

T = a b aa ba bac

F_0 F_1 F_2 F_3 F_4 F_5

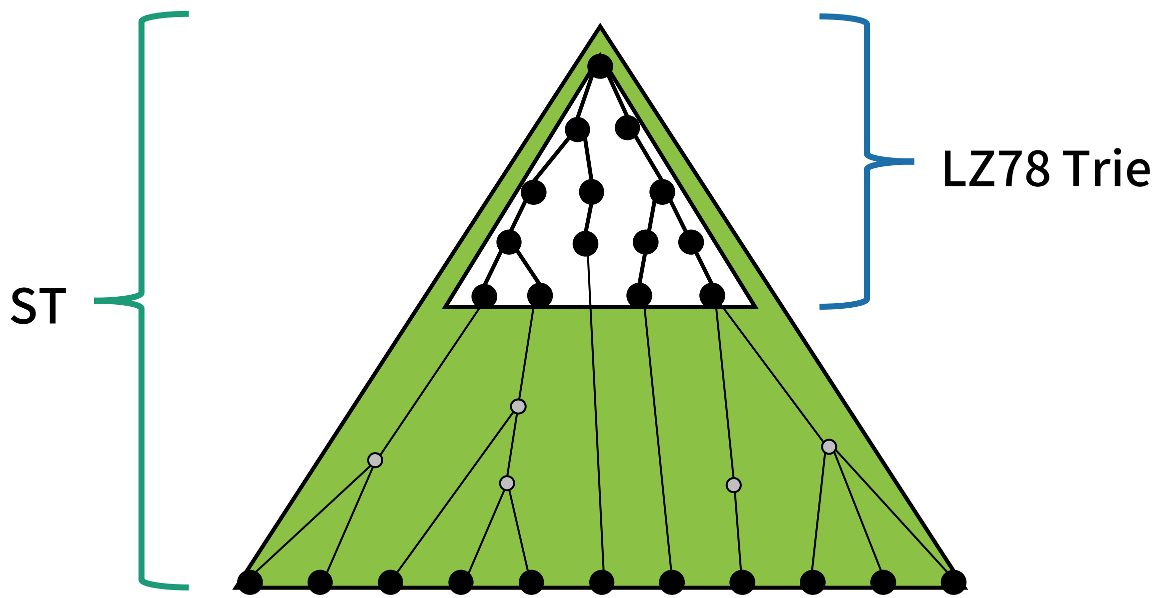
F_0a F_1a F_4c

F_0b F_2a



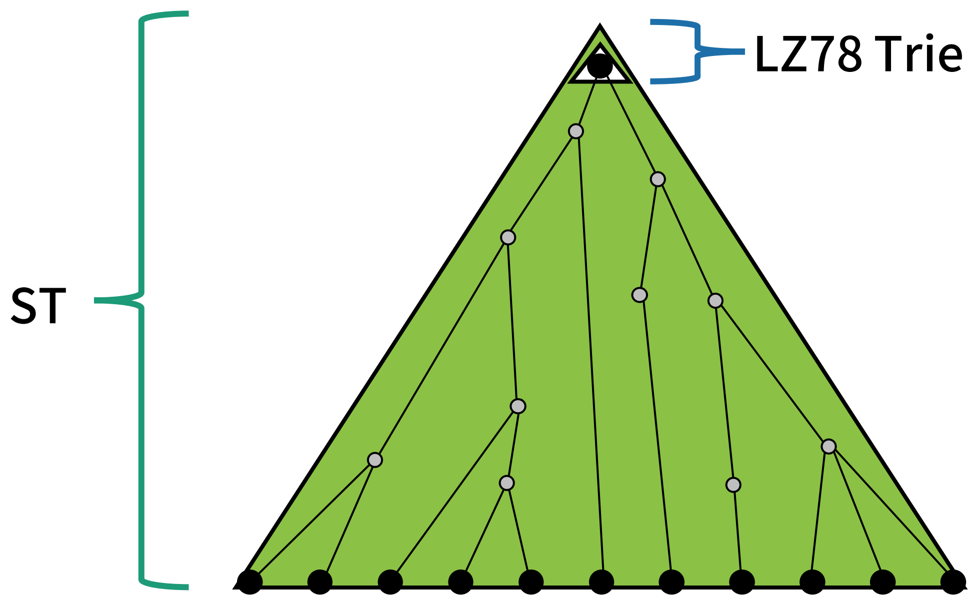
Superimposing LZ78 trie onto ST

- **We can superimpose the LZ78 trie onto the ST.**
  - The LZ78 trie is an induced subgraph over the ST consisting only of the LZ78 factors.



Overview of LZ78 Substring Compression

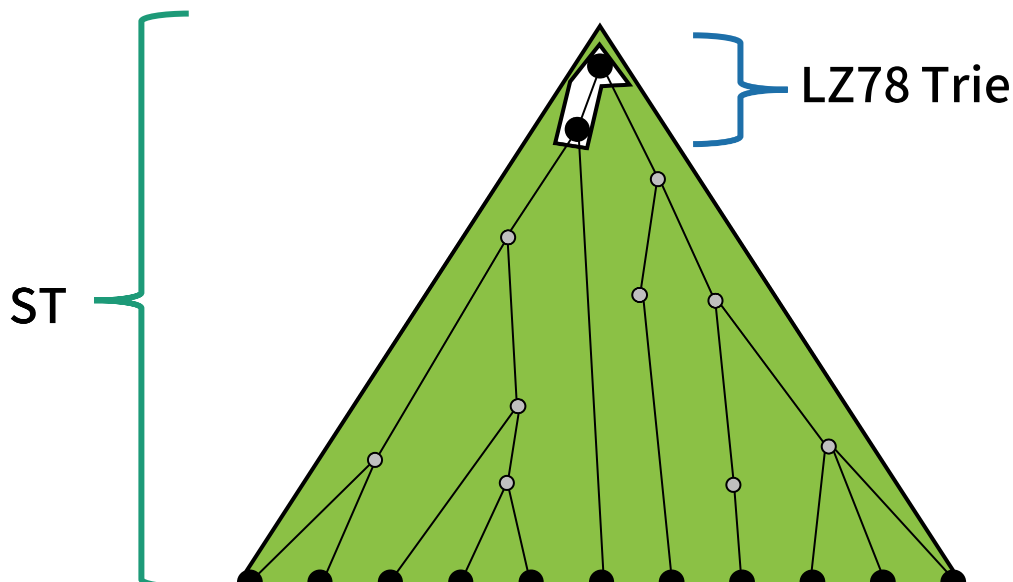
- **We can compute LZ78 compression of $T[l..r]$ by following procedure:**
  1. Initialize a LZ78 trie with one node and set $i \leftarrow l, j \leftarrow 1$.
  2. Repeat the following procedure until $i \geq r$.
     1. Compute a new factor F_j and insert F_j into LZ78 trie.
     2. Set $i \leftarrow i + |F_j|, j \leftarrow j + 1$.



Overview of LZ78 Substring Compression

- **We can compute LZ78 compression of $T[l..r]$ by following procedure:**
  1. Initialize a LZ78 trie with one node and set $i \leftarrow l, j \leftarrow 1$.
  2. Repeat the following procedure until $i \geq r$.
     1. Compute a new factor F_j and insert F_j into LZ78 trie.
     2. Set $i \leftarrow i + |F_j|, j \leftarrow j + 1$.

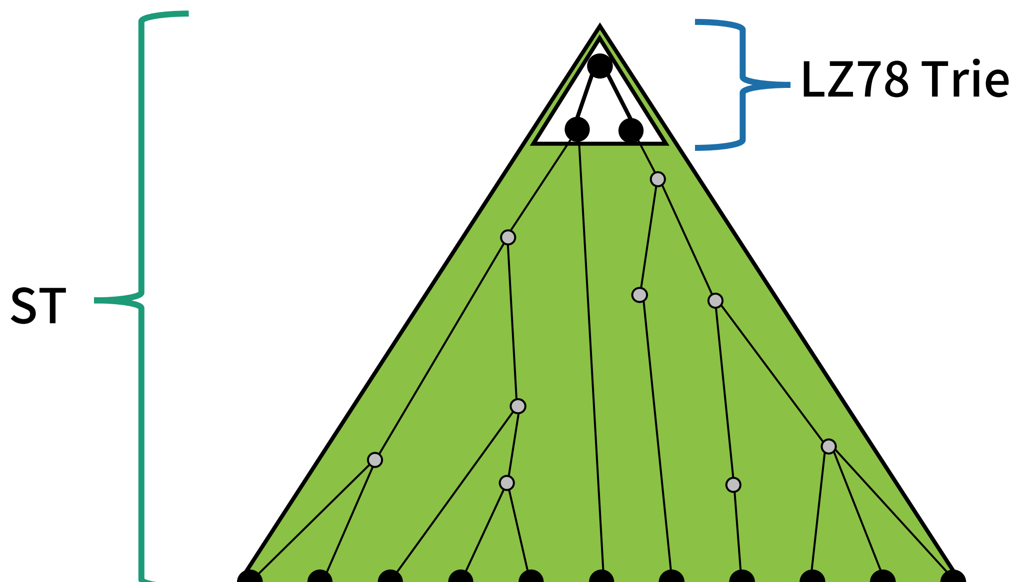
The details are described later.



Overview of LZ78 Substring Compression

- **We can compute LZ78 compression of $T[l..r]$ by following procedure:**
  1. Initialize a LZ78 trie with one node and set $i \leftarrow l, j \leftarrow 1$.
  2. Repeat the following procedure until $i \geq r$.
     1. Compute a new factor F_j and insert F_j into LZ78 trie.
     2. Set $i \leftarrow i + |F_j|, j \leftarrow j + 1$.

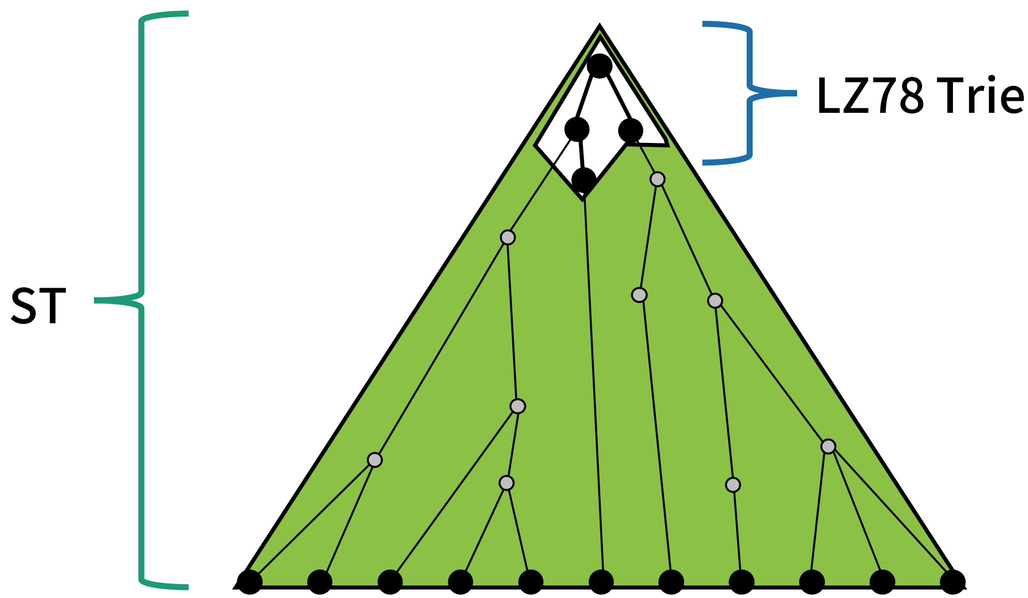
The details are described later.



Overview of LZ78 Substring Compression

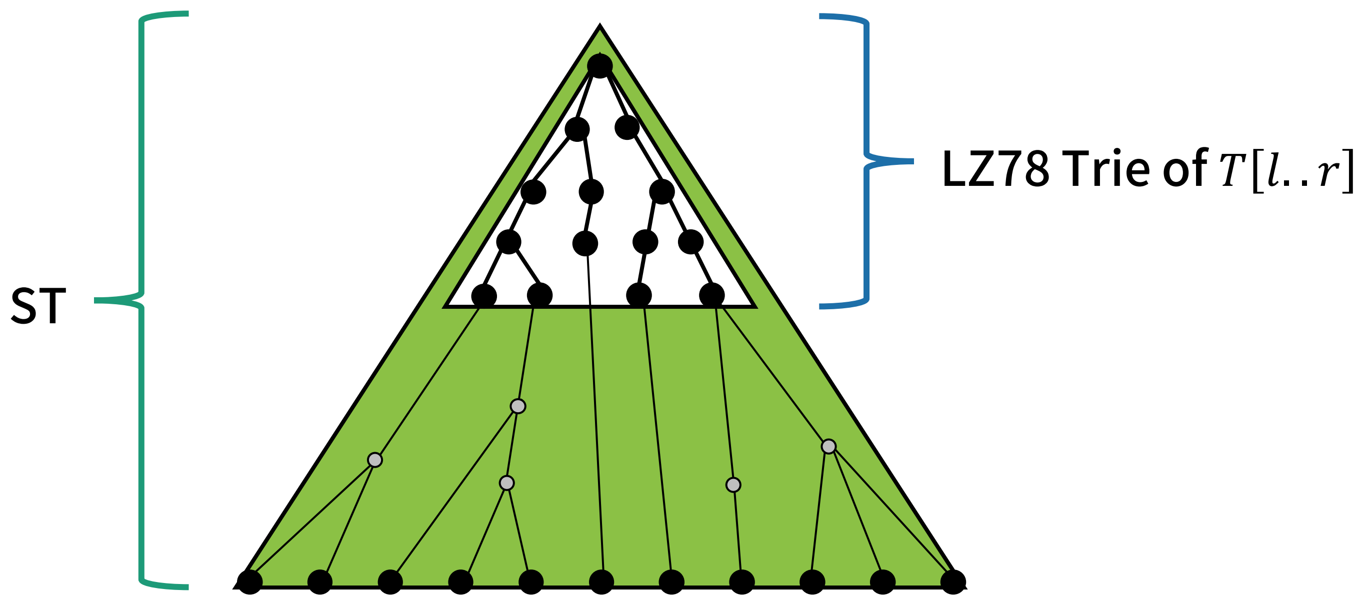
- **We can compute LZ78 compression of $T[l..r]$ by following procedure:**
  1. Initialize a LZ78 trie with one node and set $i \leftarrow l, j \leftarrow 1$.
  2. Repeat the following procedure until $i \geq r$.
     1. Compute a new factor F_j and insert F_j into LZ78 trie.
     2. Set $i \leftarrow i + |F_j|, j \leftarrow j + 1$.

The details are described later.



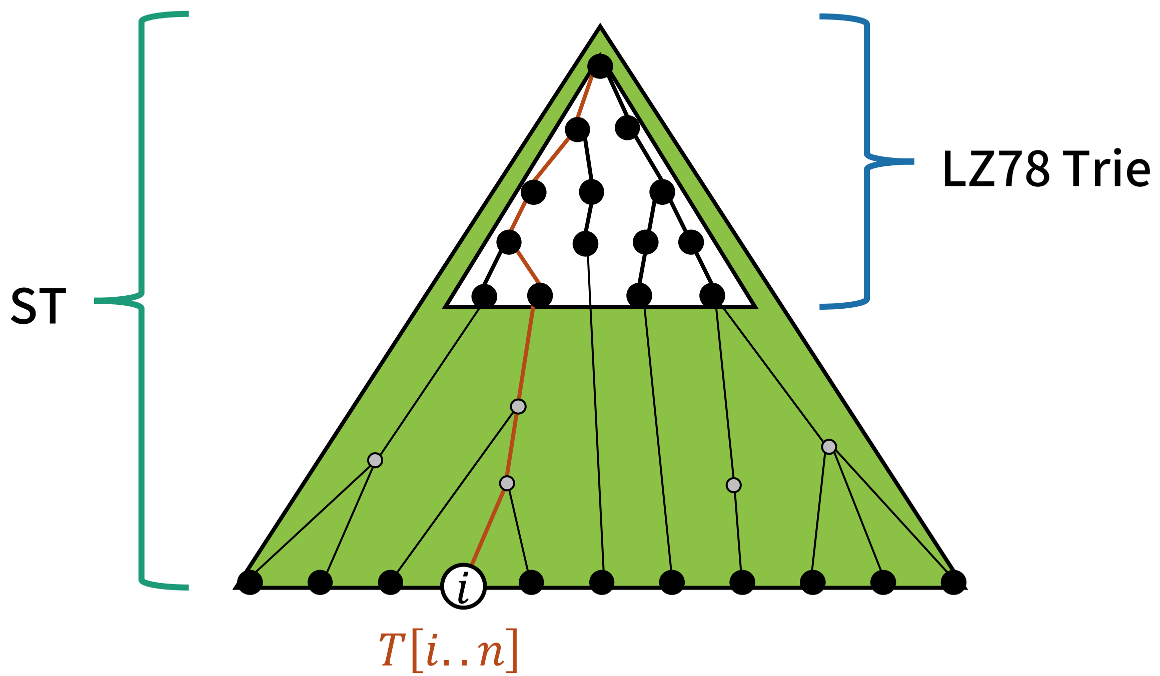
Overview of LZ78 Substring Compression

- **We can compute LZ78 compression of $T[l..r]$ by following procedure:**
  1. Initialize a LZ78 trie with one node and set $i \leftarrow l, j \leftarrow 1$.
  2. Repeat the following procedure until $i \geq r$.
     1. Compute a new factor F_j and insert F_j into LZ78 trie.
     2. Set $i \leftarrow i + |F_j|, j \leftarrow j + 1$.



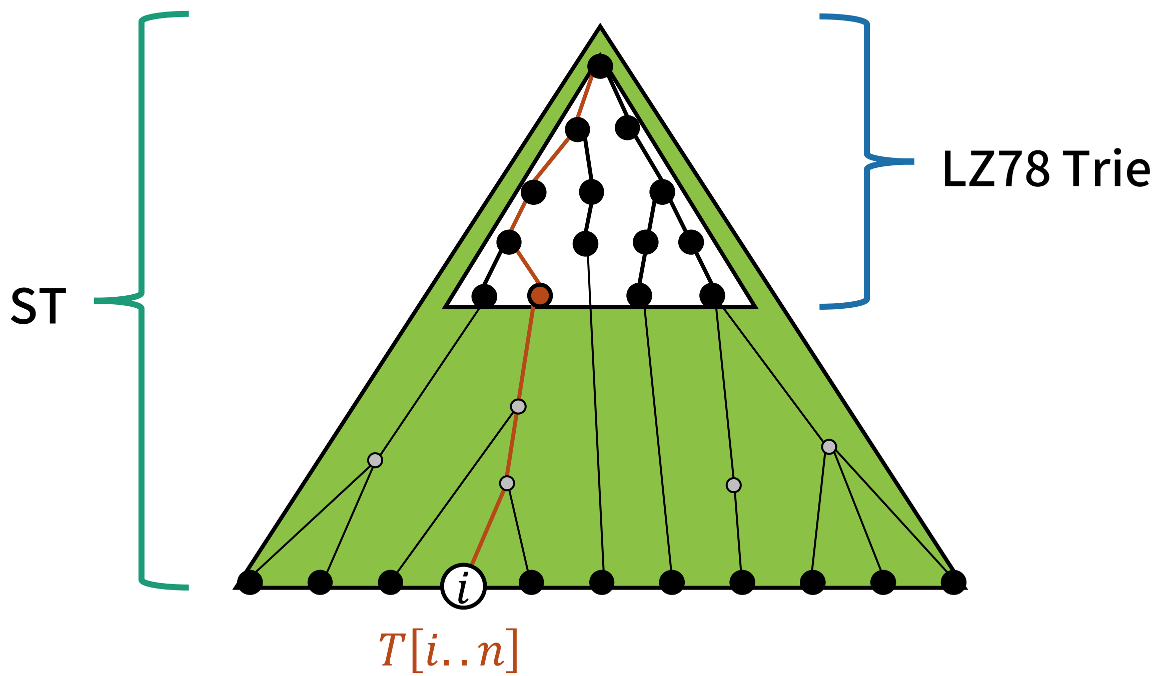
Computing the new factor by ST

- The new factor F_j starting from the position i is computed as follows:
  1. Find the leaf corresponding to the position i.
  2. Find the lowest LZ78 node on the path.
  3. Add an LZ78 node F_j immediately below the lowest LZ78 node.



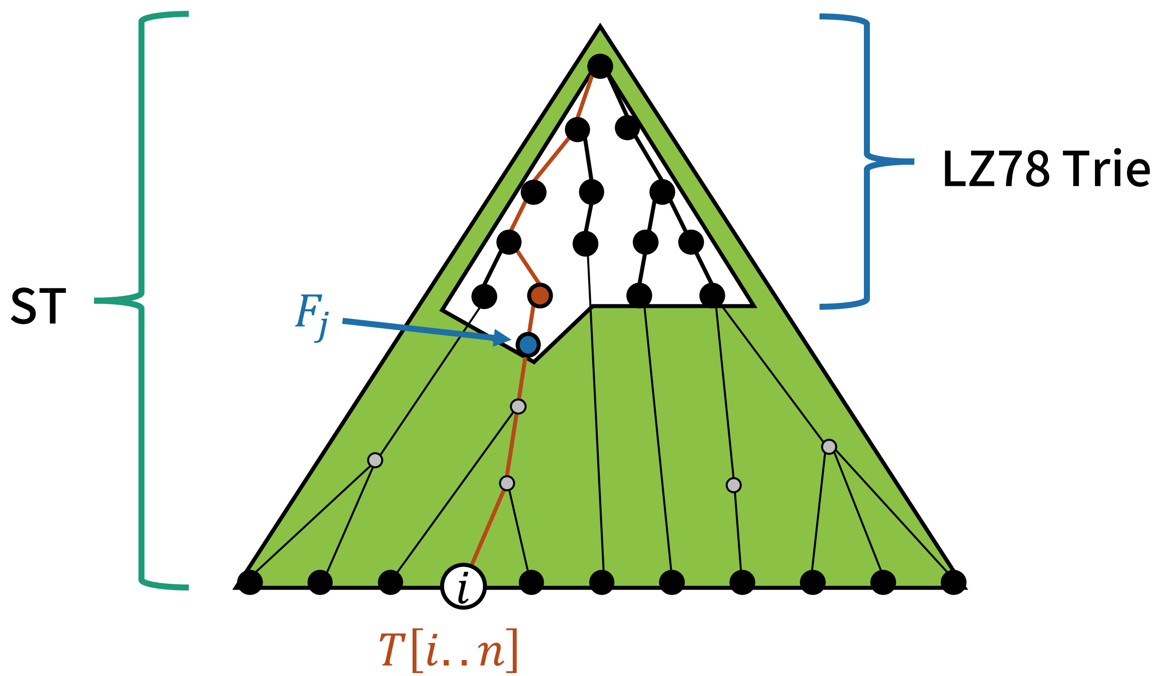
Computing the new factor by ST

- **The new factor F_j starting from the position i is computed as follows:**
  1. Find the leaf corresponding to the position i.
  2. Find the lowest LZ78 node on the path.
  3. Add an LZ78 node F_j immediately below the lowest LZ78 node.



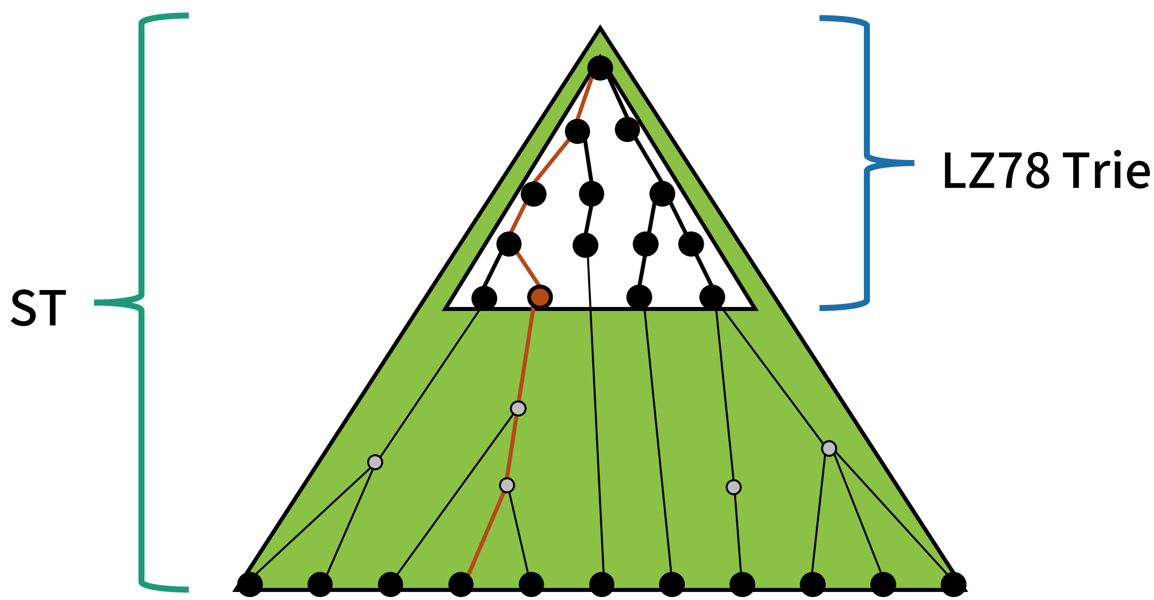
Computing the new factor by ST

- **The new factor F_j starting from the position i is computed as follows:**
  1. Find the leaf corresponding to the position i.
  2. Find the lowest LZ78 node on the path.
  3. Add an LZ78 node F_j immediately below the lowest LZ78 node.



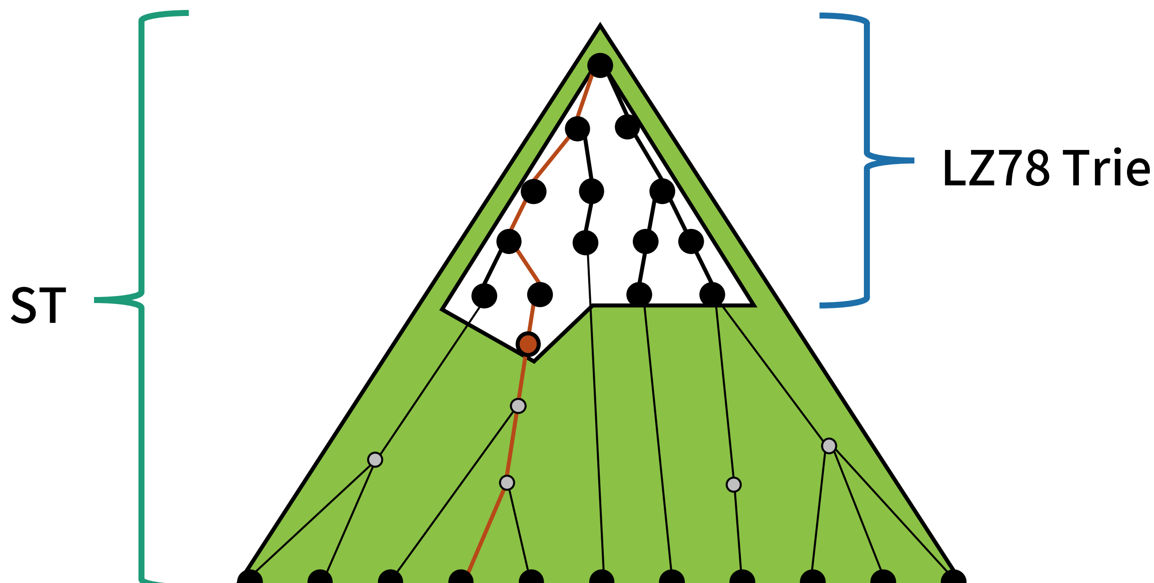
Required Data Structures

- These operations requires some data structure:
  - Nearest Marked Ancestor: To find the lowest LZ78 node
  - Weighted Level Ancestor: To find the position on the path immediately below the lowest LZ78 node



Required Data Structures

- These operations requires some data structure:
  - Nearest Marked Ancestor: To find the lowest LZ78 node
  - Weighted Level Ancestor: To find the position on the path immediately below the lowest LZ78 node



Implementation and Complexity

Required data structures:

- Lowest Marked Ancestor [Westbrook, 1992]
  - amortized O(1) time / query
- Weighted Level Ancestor [Gawrychowski et al., 2014]
  - O(1) time / query

Time / Space complexity

- Each data structure can be stored in $O(n)$ words.
- We can find the lowest factor and add a factor in constant time per factor.

■ **Overall Complexity ⇒ $O(n)$ words • $O(c)$ time / query**

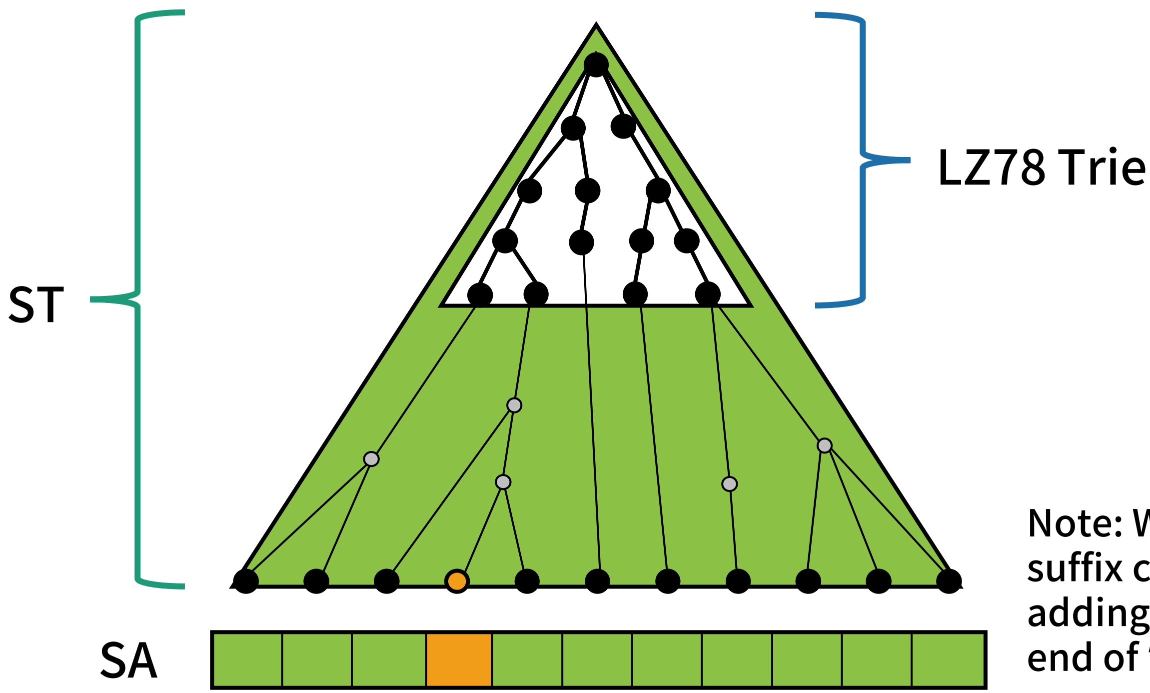
(c is the number of LZ78 factors of $T[l..r]$)

Computing LZ78 using a stabbing-max structure (Intermediate method)

Suffix Arrays (SA)

The suffix array stores the lexicographic order of all suffixes.

- Since the leaves have a one-to-one correspondence to all suffixes, the SA represents the order of the leaves.
- One leaf of the ST corresponds to one position on the SA.

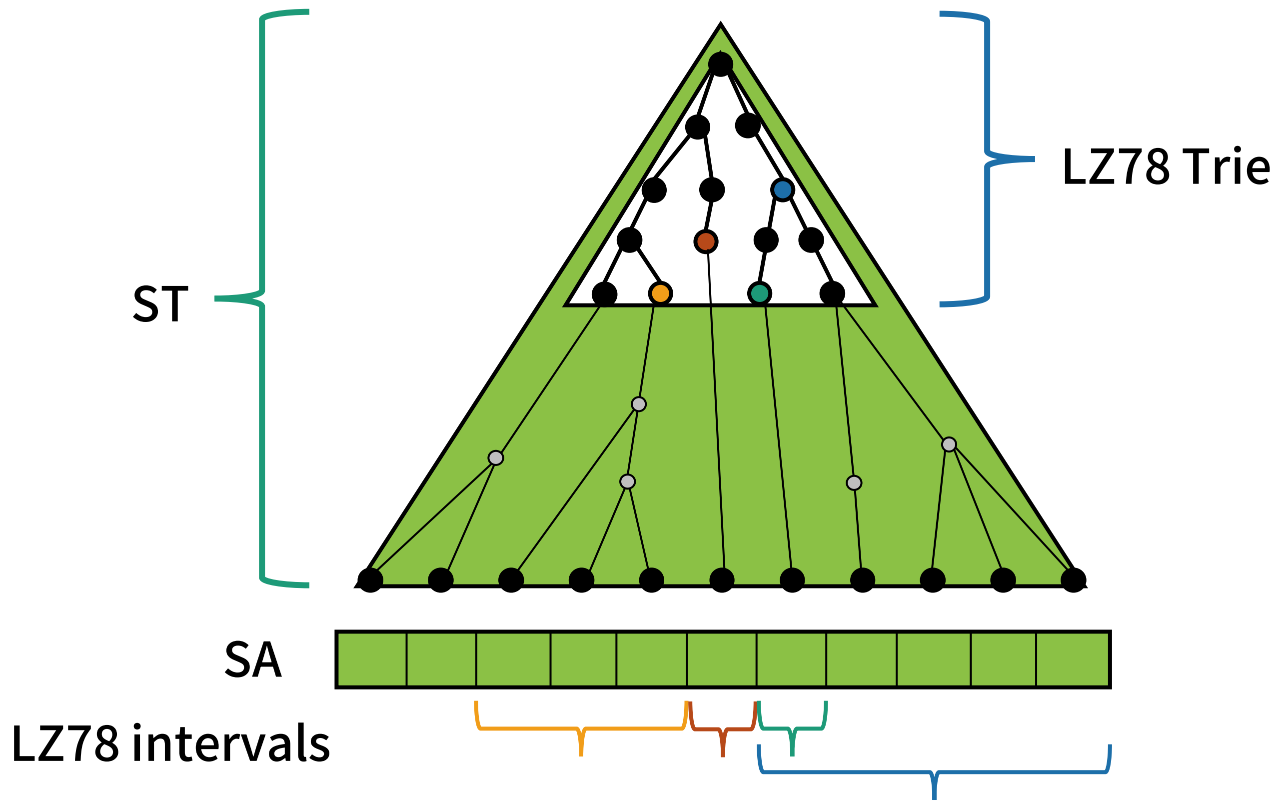


Note: We can guarantee that each suffix corresponds to a leaf node by adding a unique character at the end of T.

Converting LZ78 Nodes to Intervals

Each LZ78 node corresponds to an interval on SA.

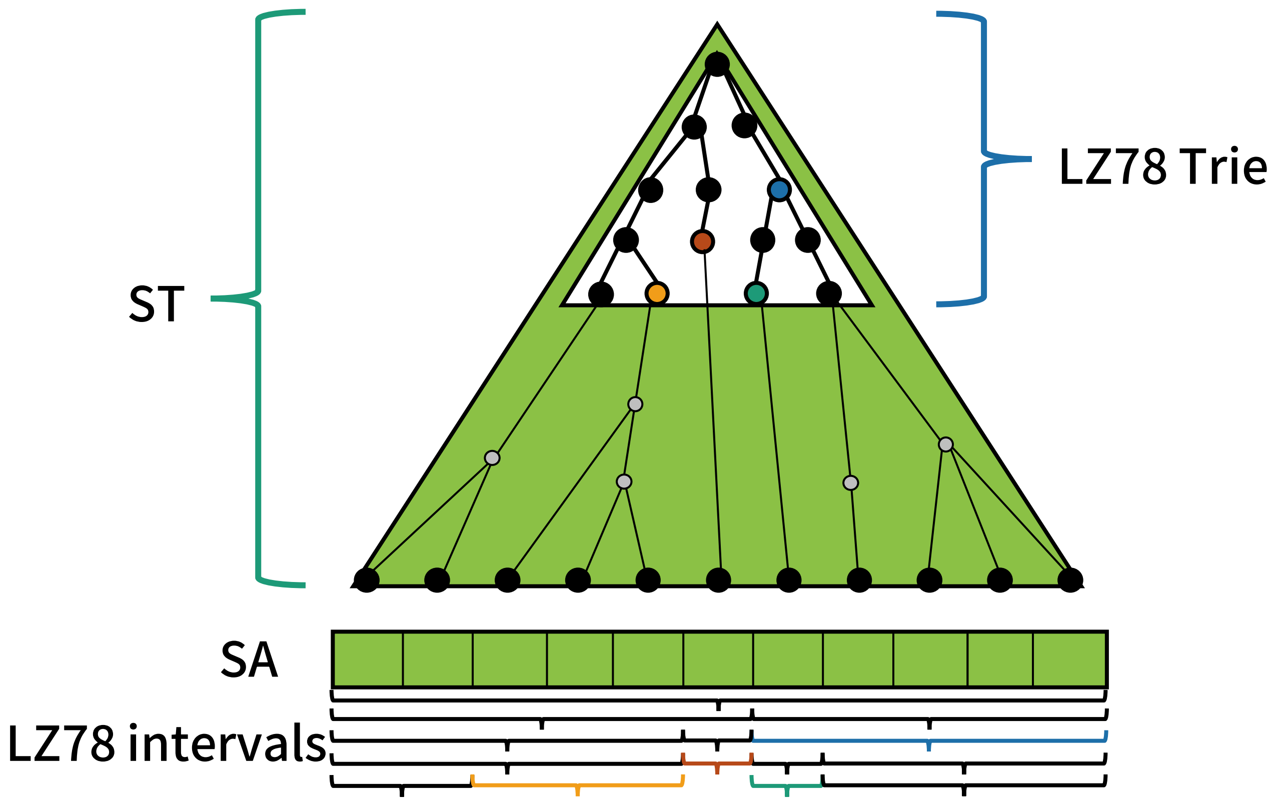
→ The set of all LZ78 nodes can be represented by the set of intervals.



Converting LZ78 Nodes to Intervals

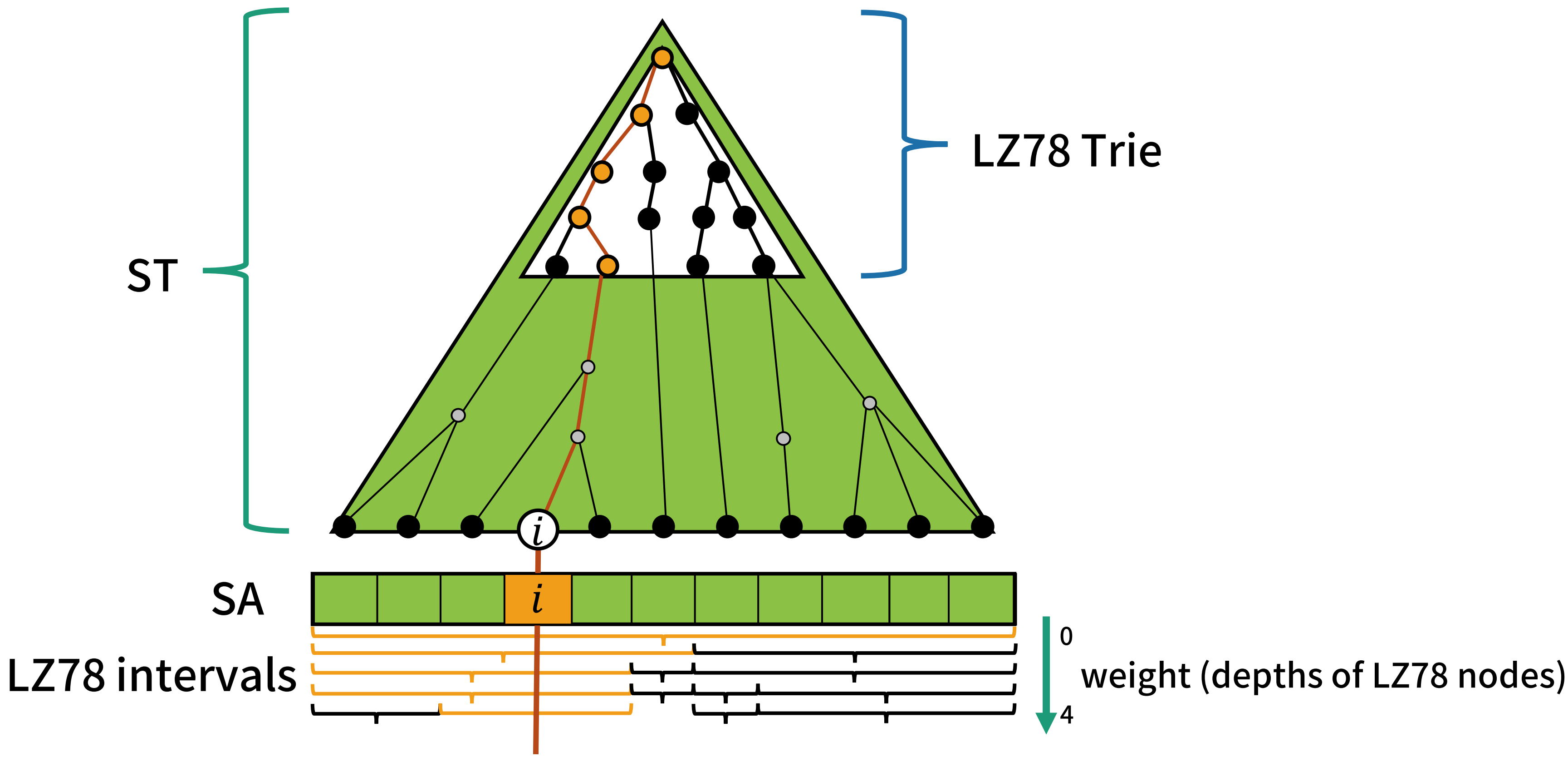
Each LZ78 node corresponds to an interval on SA.

→ The set of all LZ78 nodes can be represented by the set of intervals.



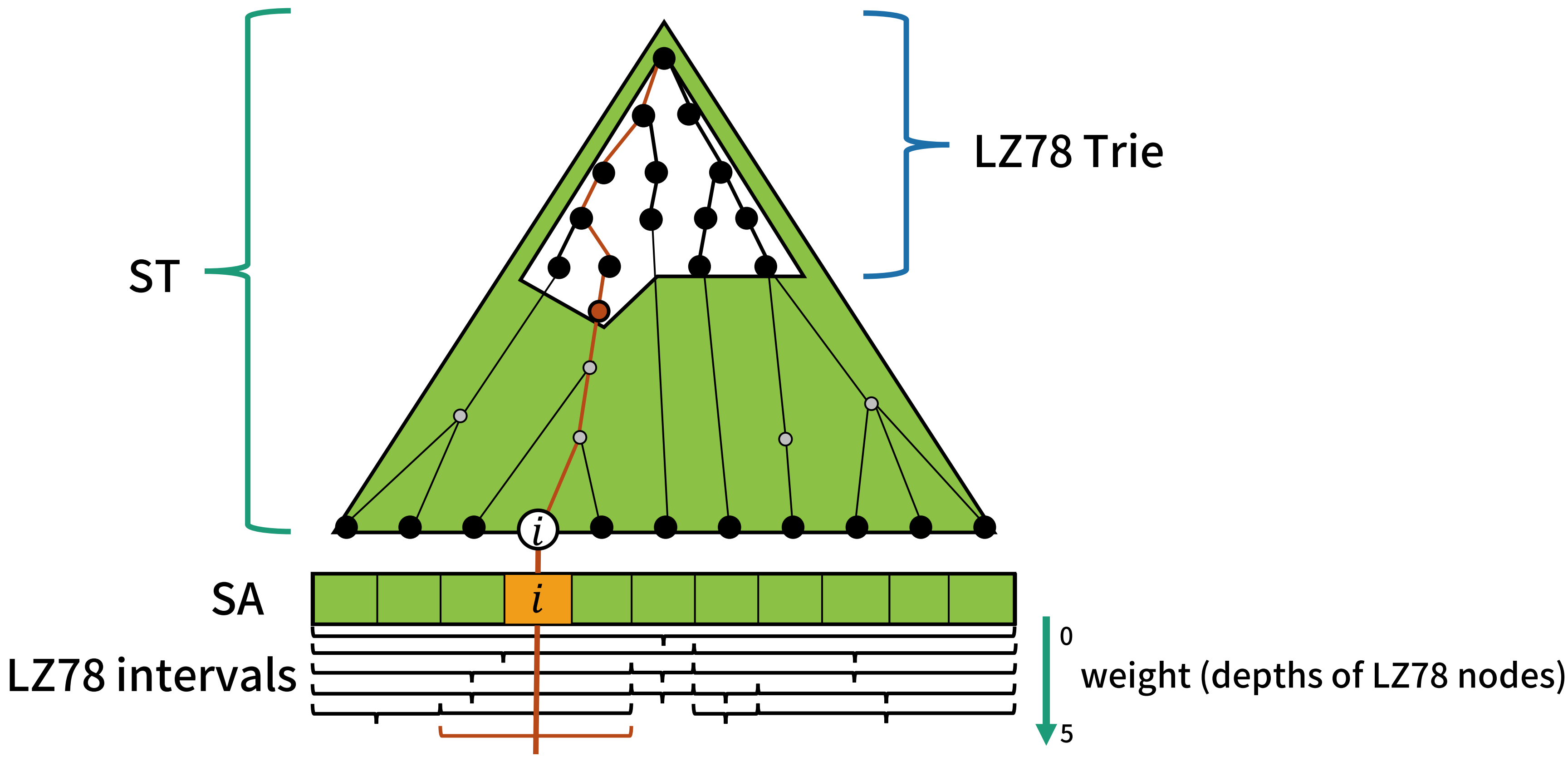
Converting LZ78 Nodes to Intervals

Finding lowest LZ78 node on the path corresponds to find the interval containing the specific position whose weight is maximum.



Converting LZ78 Nodes to Intervals

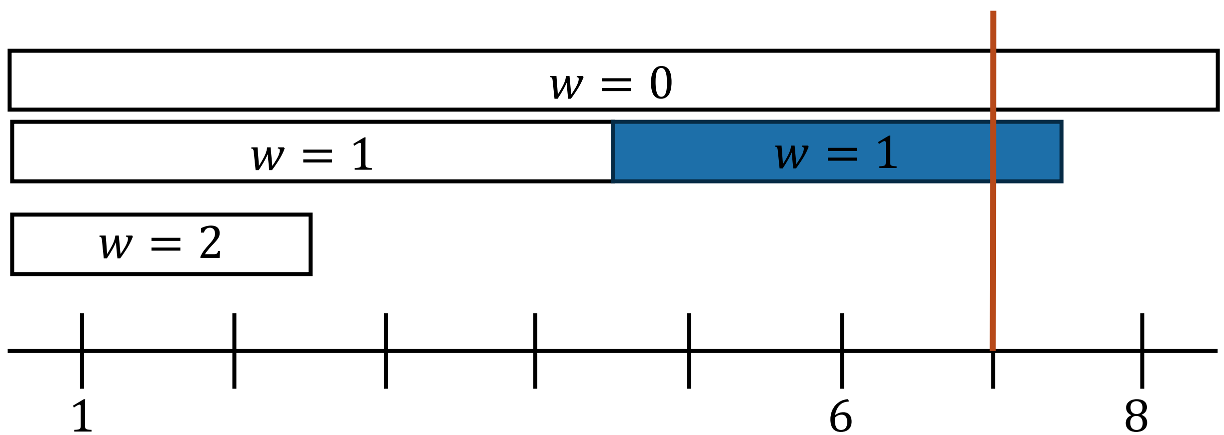
Adding an LZ78 node corresponds to add an interval representing a new LZ78 node.



Stabbing-Max Problem

Computing a LZ78 factor can be reduced to the following queries and operations:

1. **Find the interval $[a, b] \in S$ containing k whose weight is maximum.**
   - Corresponding to find the lowest LZ78 node.
2. **Add an interval $[a, b]$ with weight w to a set S.**
   - Corresponding to add an LZ78 node.



Stabbing-Max Problem

Computing a LZ78 factor can be reduced to the following queries and operations:

1. Find the interval $[a, b] \in S$ containing k whose weight is maximum.
2. Add an interval $[a, b]$ with weight w to a set S.

This problem is known as the stabbing-max problem, and there is a data structure that performs any query with the following complexity [Tarjan, 1979]:

- Add/Find an interval: $O(\log m)$ time / query
- Space complexity: $O(m)$ words

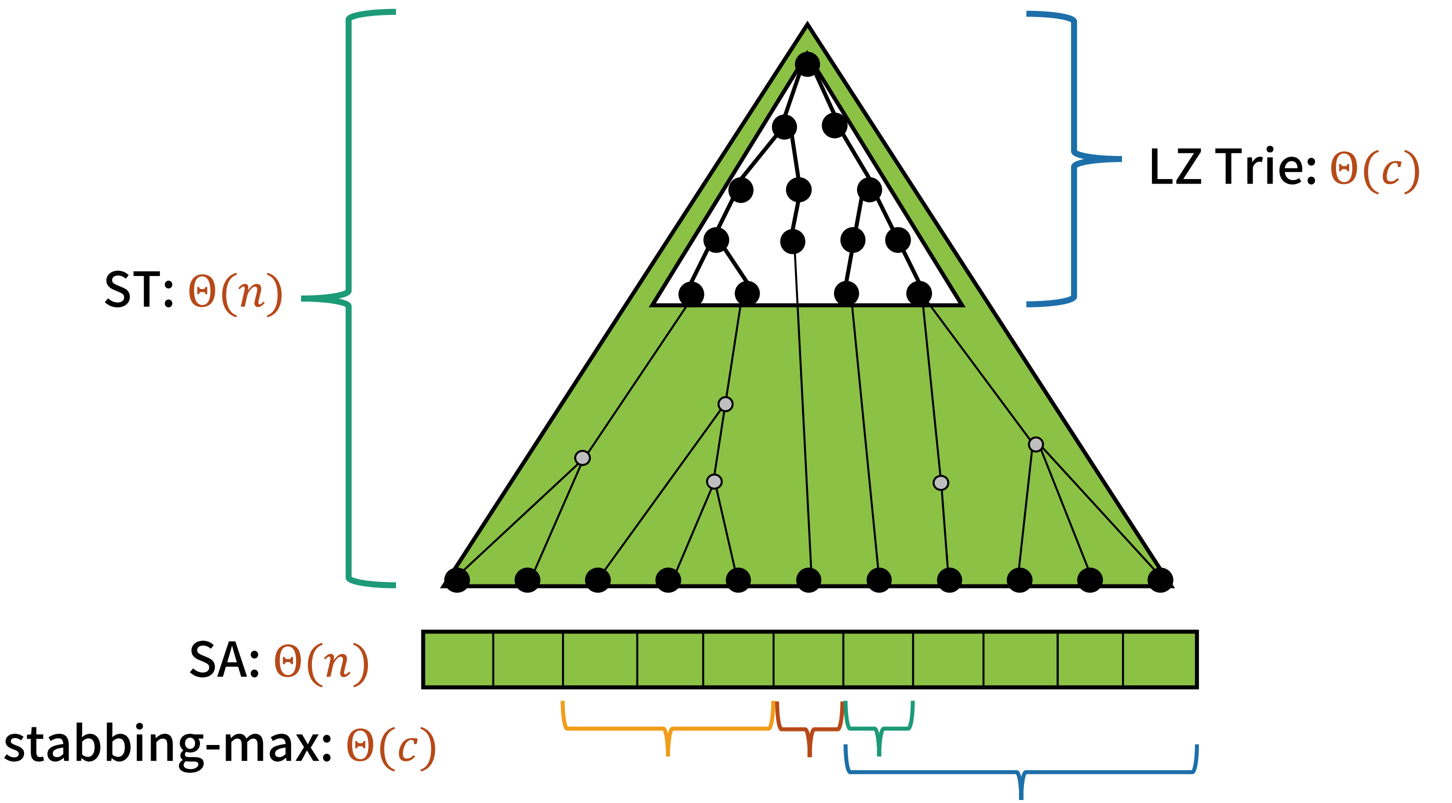
(m is the number of intervals)

Computing LZ78 in compressed space (proposed method)

Bottleneck of Space Complexity

Now, our data structure uses $\Theta(n)$ space because of the ST and the SA.

To reduce the space, we need to compute LZ78 factors without them.



Note: $c \in O(n)$

Required operations

Our method depends on some operations:

- Obtaining $T[i]$.
- Computing the leaf corresponding to $T[i..n]$.
- Computing the interval corresponding to $T[l..r]$.

Without ST/SA and the text, we cannot perform these queries.

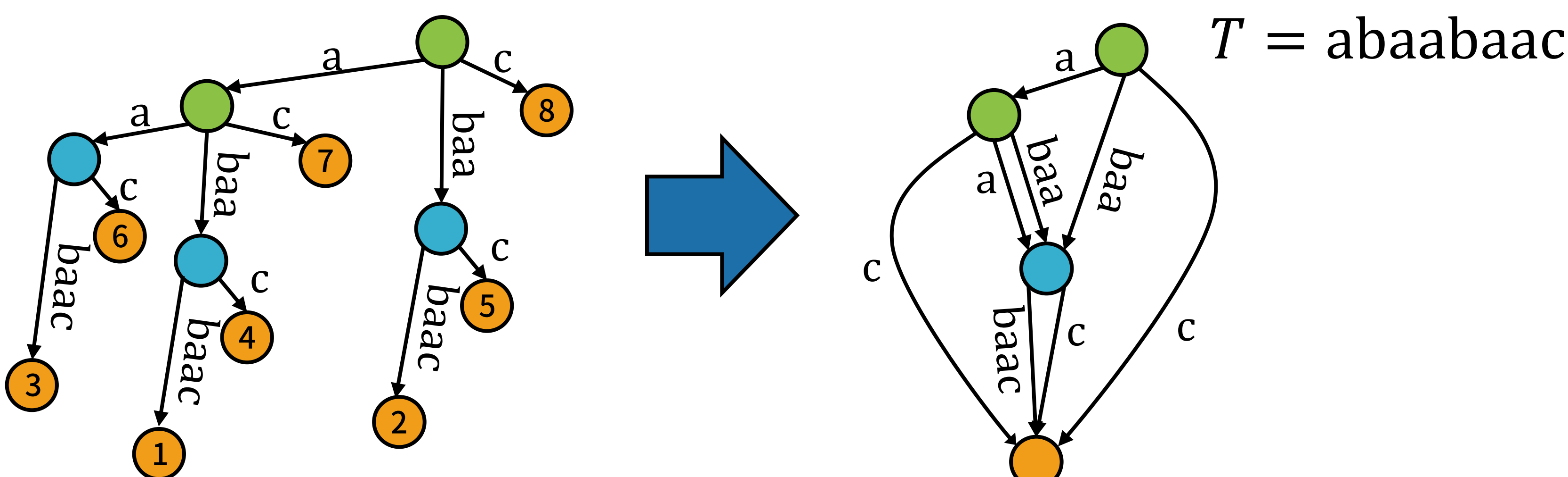
→ We use a data structure that simulates ST/SA.

CDAWG (Compact Directed Acyclic Word Graph)

CDAWG: The edge-labelled DAG which obtained by merging isomorphic subtrees of the corresponding ST.

- **Property: The number e of edges of the CDAWG is small for some highly repetitive strings.**

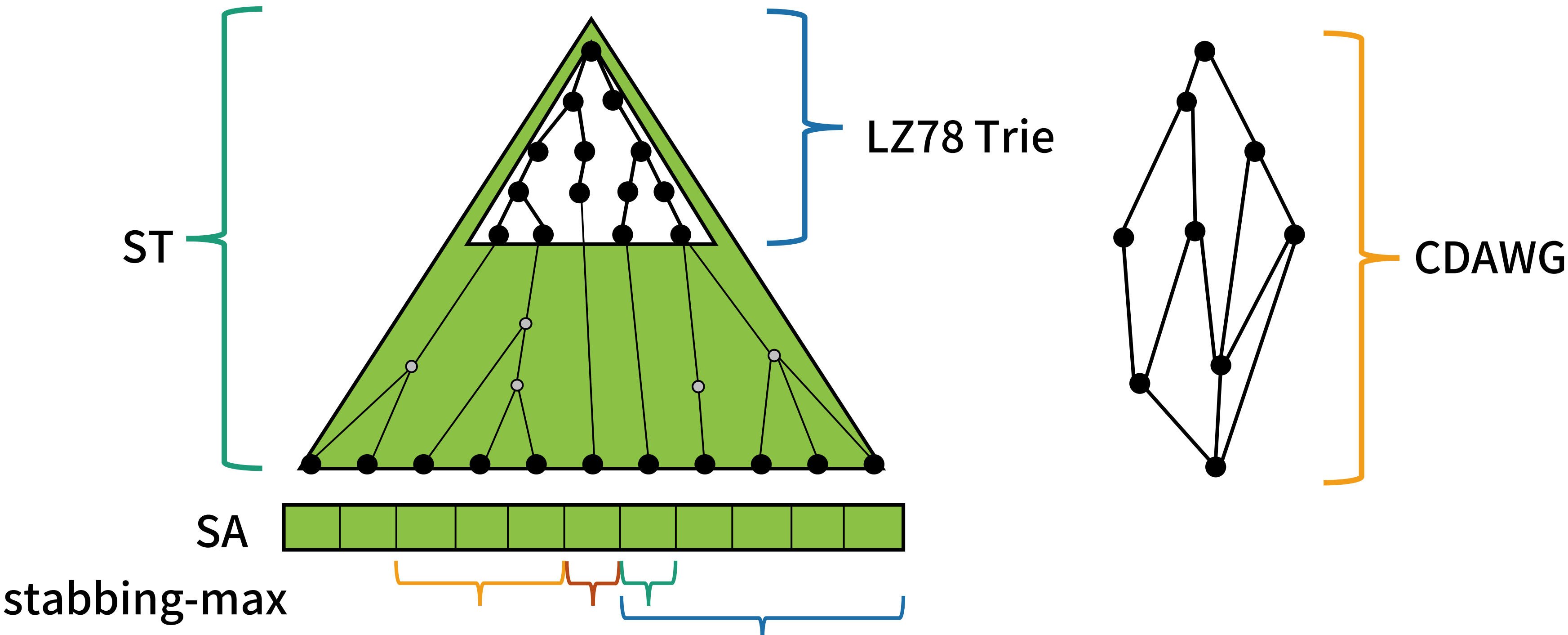
  → CDAWG can be regarded as a compressed representation of STs.



Replacing ST/SA with CDAWG

Since CDAWG is a compacted variant of ST, some CDAWG-based indexes efficiently performs ST/SA operation.

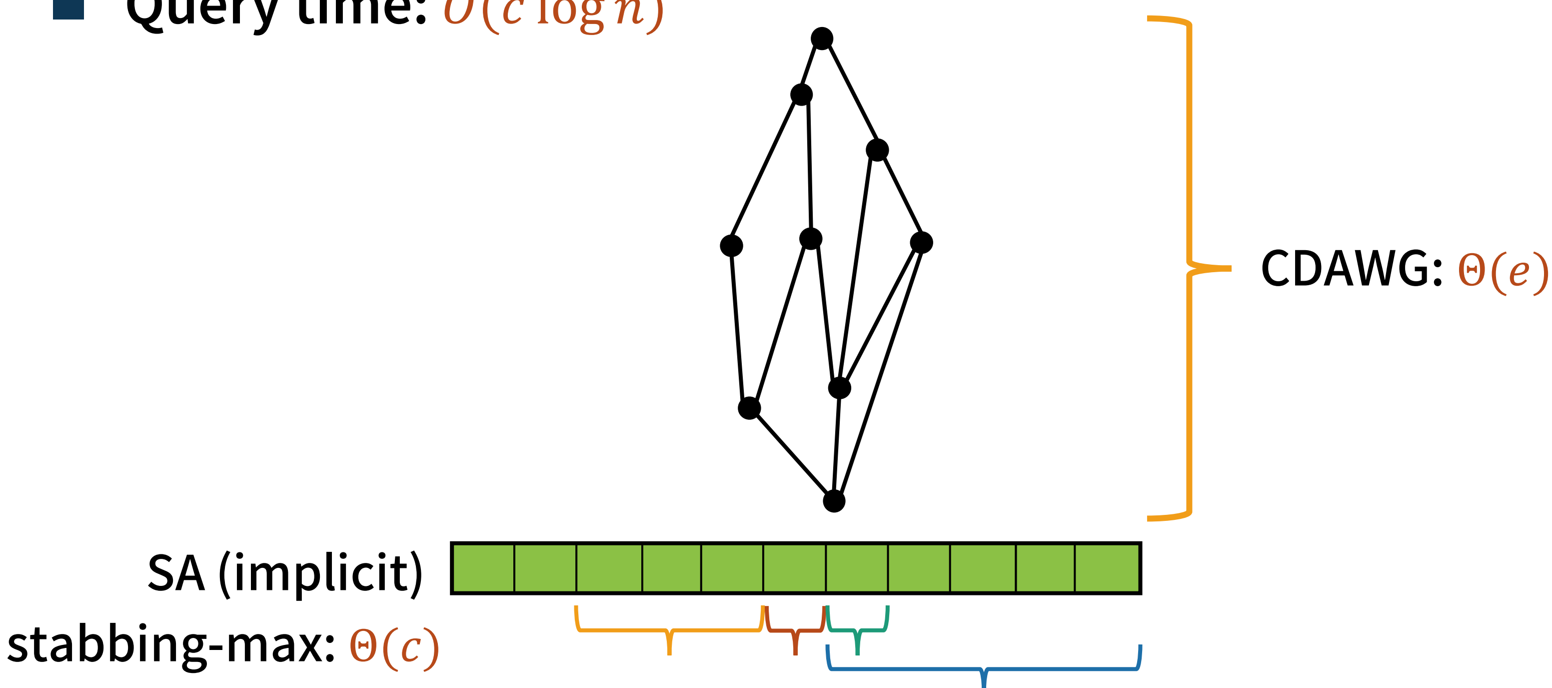
There is a CDAWG-based index that simulate ST in $O(\log n)$ time and is stored in $O(e)$ space. [Bealazzougui and Cunial, CPM 2017]



Overall Structure

Our method only uses the CDAWG and the stabbing-max structure.

- **Space for index: $\Theta(e)$ words**
- **Working space for queries: $\Theta(c)$ words**
- **Query time: $O(c \log n)$**



Conclusion

- We proposed a method for LZ78 substring compression in compressed space.
- Some applications:
  - We can replace the CDAWG by some other compressed index.
  - Applying this method for other LZ78-like compressions. (LZD, LZMW)

Time/Space Complexity

	Space for Index	Working Space	Preprocessing Time	Query Time
[Köppl, '21]	$O(n)$	$O(c)$	$O(n)$	$O(c)$
Ours	$O(e)$	$O(c)$	$O(n)$	$O(c \log n)$