SOFSEM2025

Packed Acyclic Deterministic Finite Automata

Hiroki Shibata[1], Masakazu Ishihata[2], Shunsuke Inenaga[1]

[1.] Kyushu University

[2.] NTT Communication Science Laboratories

Overview

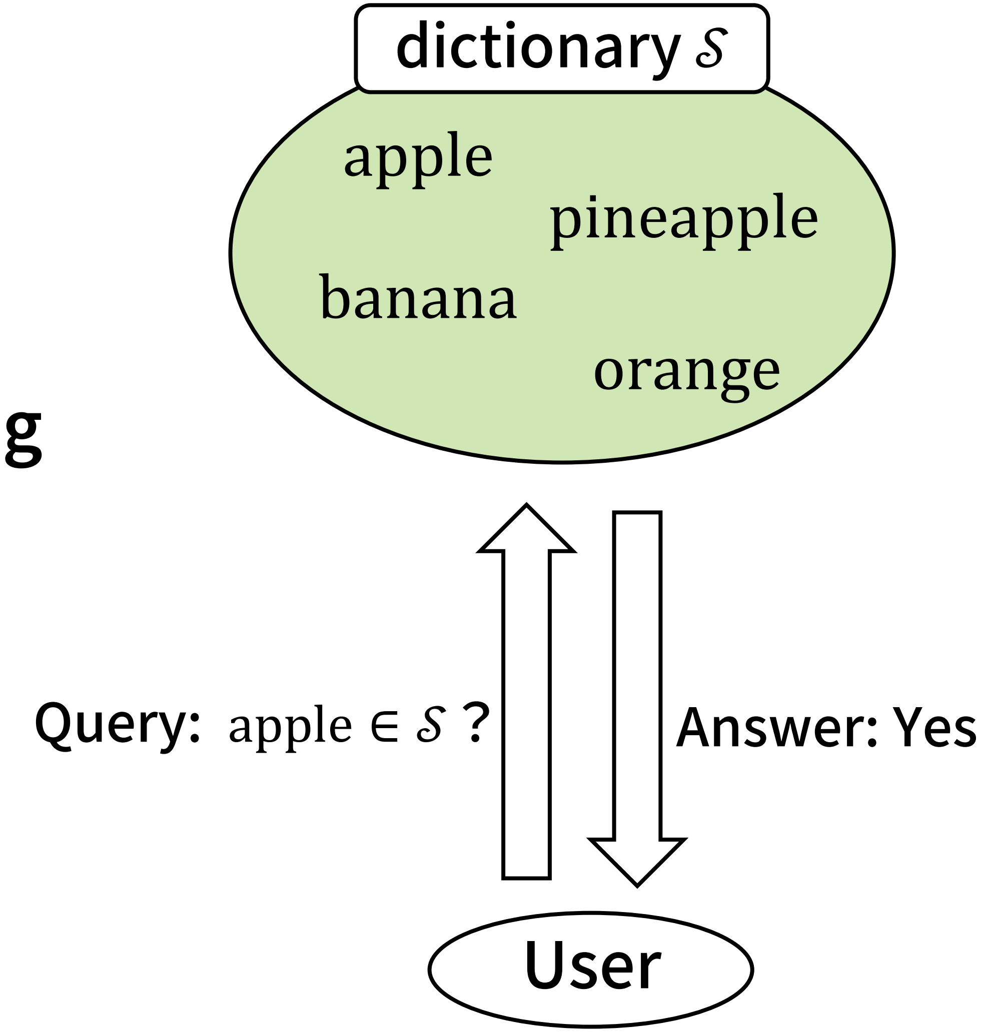
- Pattern-searching: the problem of deciding whether the input pattern is in a dictionary (a set of strings)
  - Pattern-searching index: an index structure supporting efficient pattern-searching queries.
- ADFA: a compact pattern-searching index
- Packed string: storing multiple characters in one machine word

Our method: Packed ADFA

- Performing pattern-searching in $O\left(\frac{|P|}{\alpha} + \lg k\right)$ time
- More compact than trie-based indexes

Pattern-searching

- **Dictionary**: a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$
- **Pattern-searching**: the problem of deciding whether $P \in \mathcal{S}$ (P : pattern string)



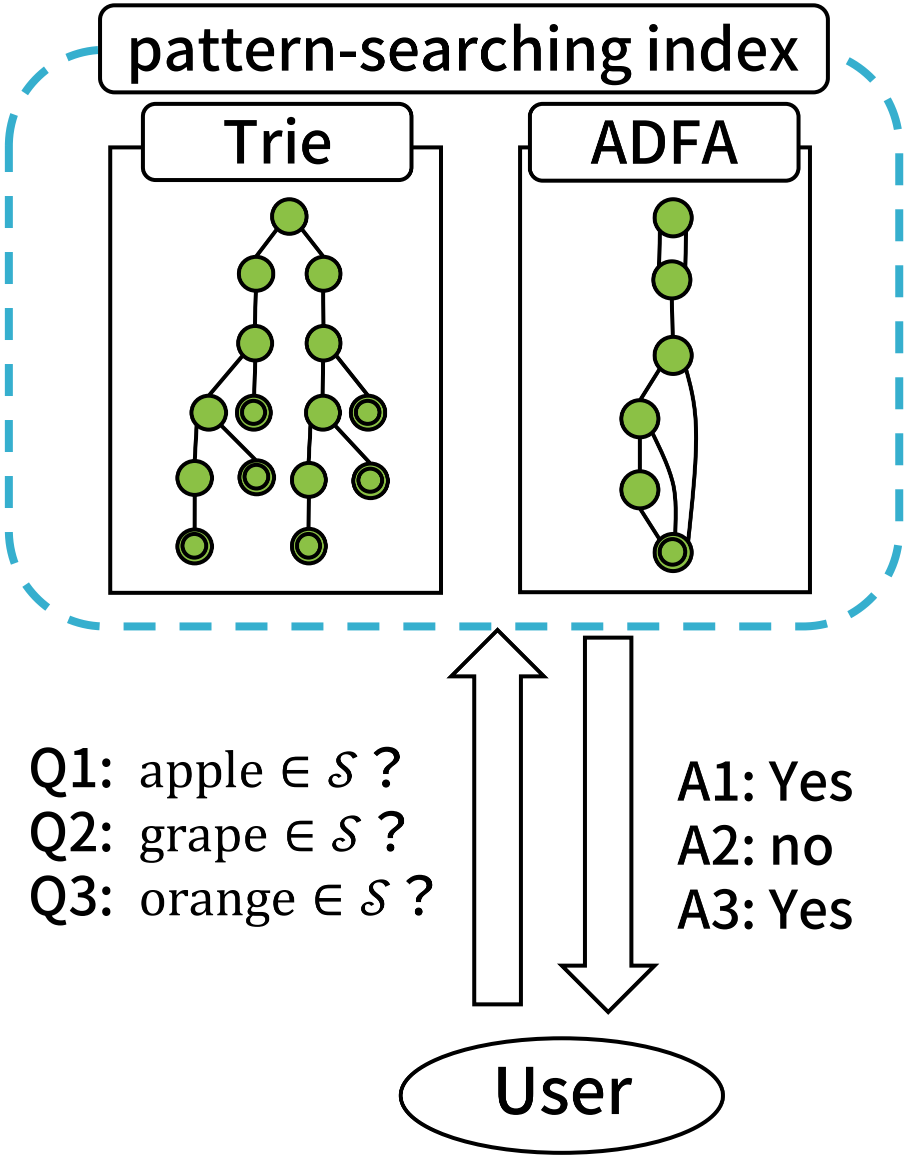
Pattern-searching index

Pattern-searching index: an index structure storing a dictionary $\mathcal{S}$ and can answer pattern-searching queries efficiently for $\mathcal{S}$.

- **It is effective if there are many queries with the same dictionary and different patterns.**

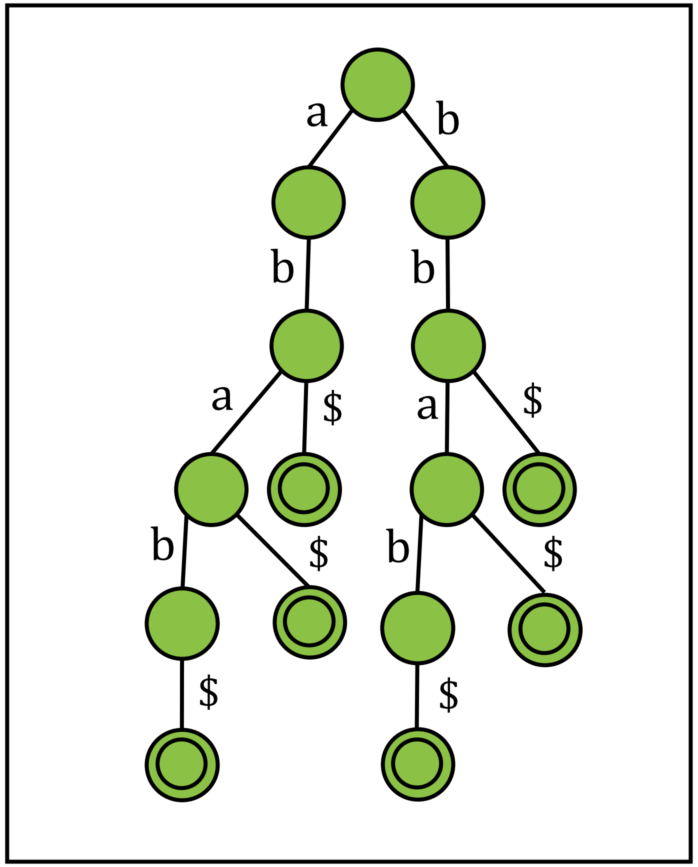
Famous pattern-searching indexes:

- Trie: a tree-formed index
- ADFA: a DAG-formed index



Trie: a tree-formed pattern-searching index

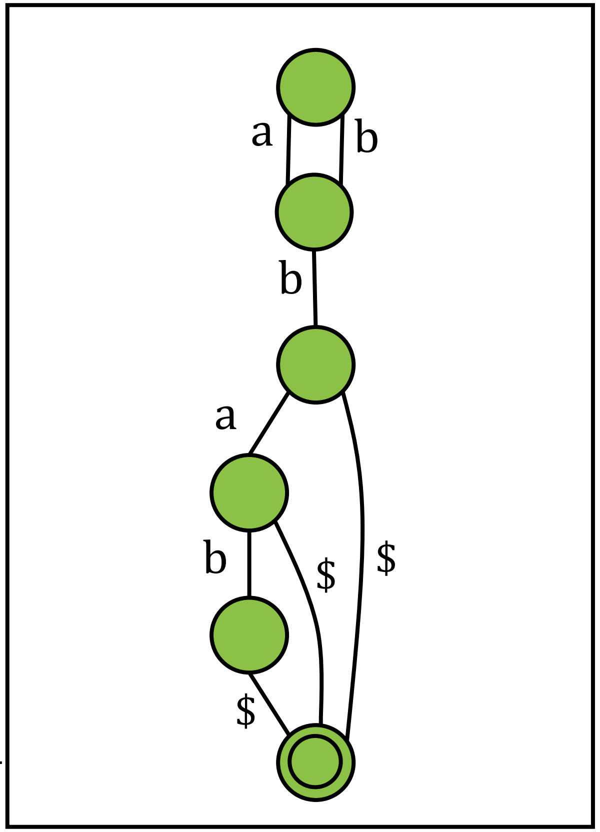
- Each path from the root to an accepting state represents a string
- Supports efficient pattern searching query
- Easy to handle and allows for the use of various techniques due to its tree structure
  - e.g. centroid path decomposition, Euler tour



a trie of
{abab$, aba$, ab$, bbab$, bba$, bb$}

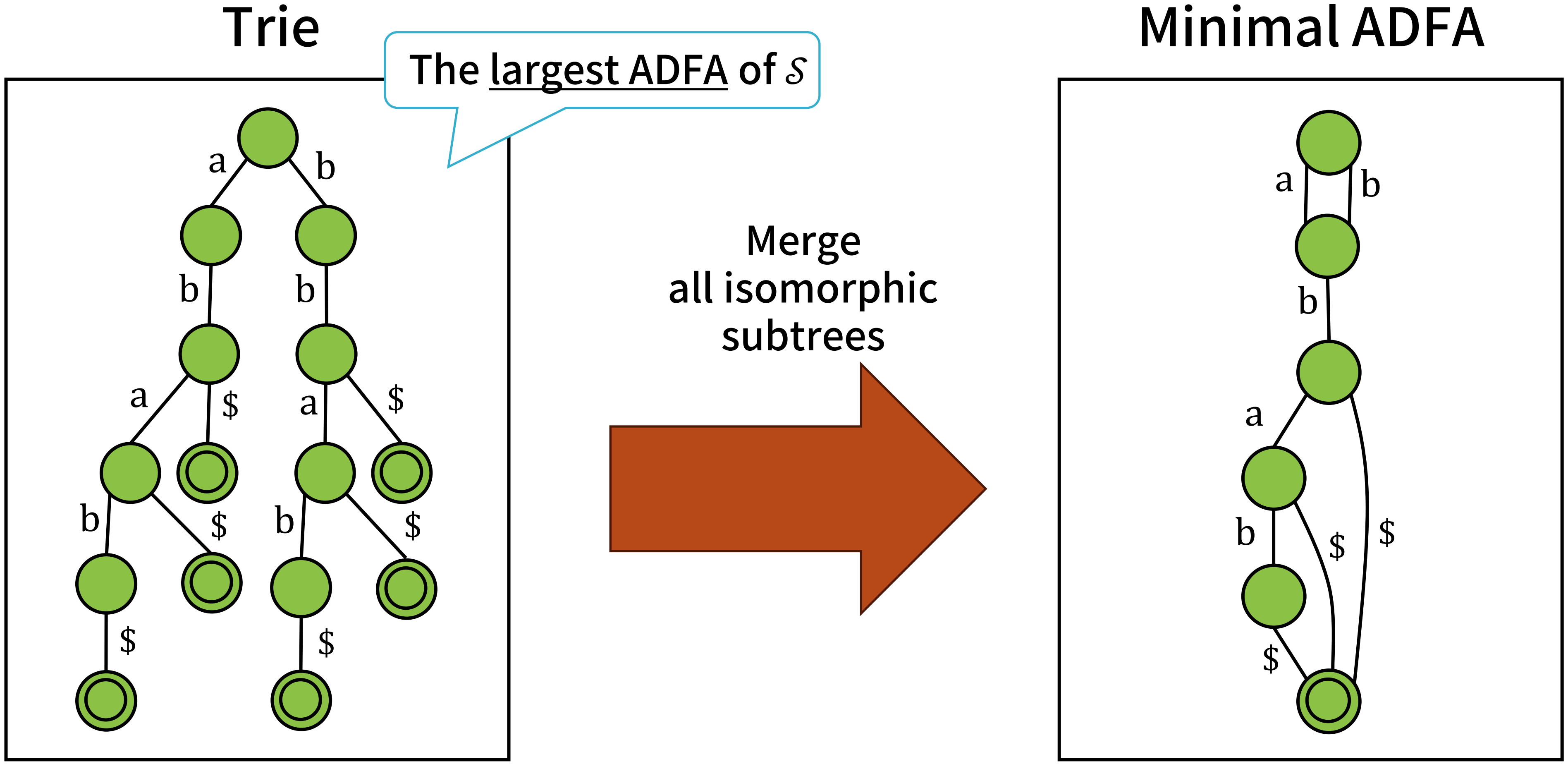
ADFA: a DAG-formed pattern-searching index

- Each path from root to an accepting state represents a string
- Supports efficient pattern searching query
- Is a generalization of trie
  - Advantage of generalization: it can be more compact than a trie
  - Disadvantage: efficient tree algorithms are not directly applicable due to its DAG structure



a ADFA of
{abab$, aba$, ab$, bbab$, bba$, bb$}

A relationship between trie and minimal ADFA

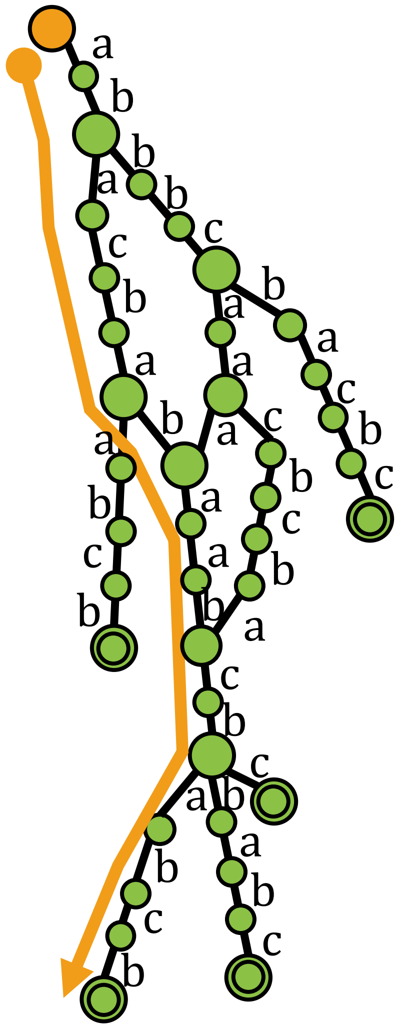


$$\mathcal{S} = \{\text{abab\$}, \text{aba\$}, \text{ab\$}, \text{bbab\$}, \text{bba\$}, \text{bb\$}\}$$

Pattern-searching query of a ADFA

- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \le |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return “No”.
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return “Yes” iff v is an acceptable state.

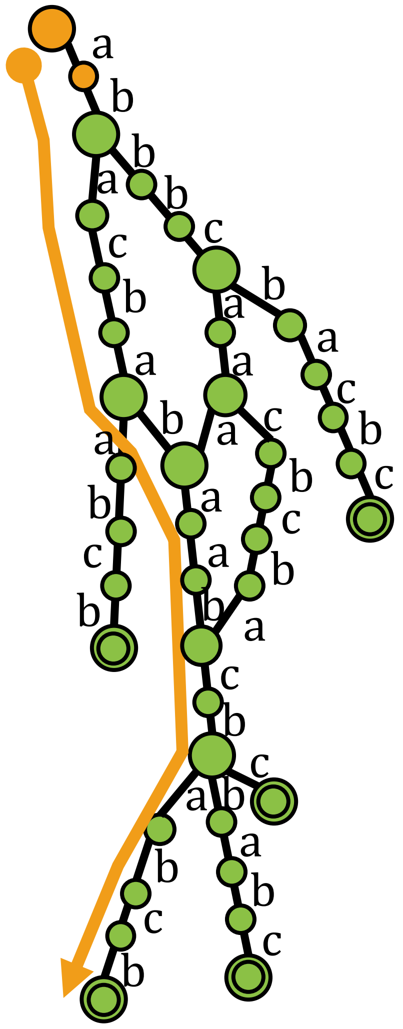
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



Pattern-searching query of a ADFA

- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \leq |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return "No".
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return "Yes" iff v is an acceptable state.

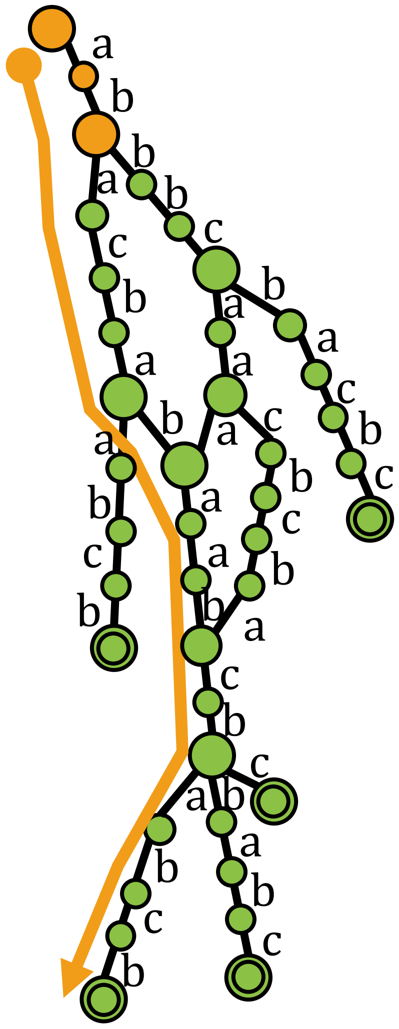
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



Pattern-searching query of a ADFA

- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \leq |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return “No”.
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return “Yes” iff v is an acceptable state.

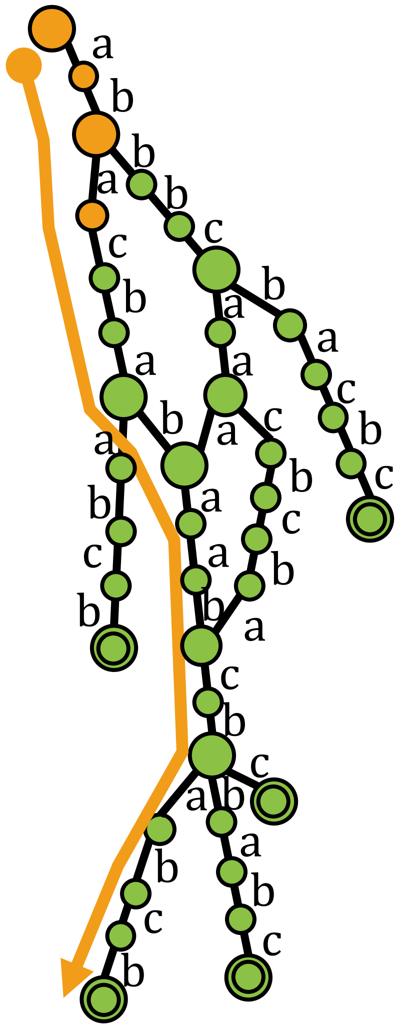
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



Pattern-searching query of a ADFA

- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \leq |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return "No".
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return "Yes" iff v is an acceptable state.

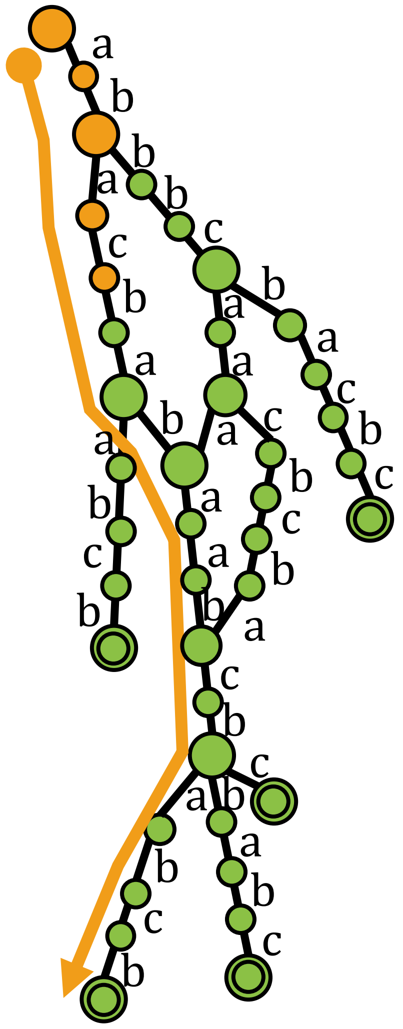
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



Pattern-searching query of a ADFA

- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \leq |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return “No”.
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return “Yes” iff v is an acceptable state.

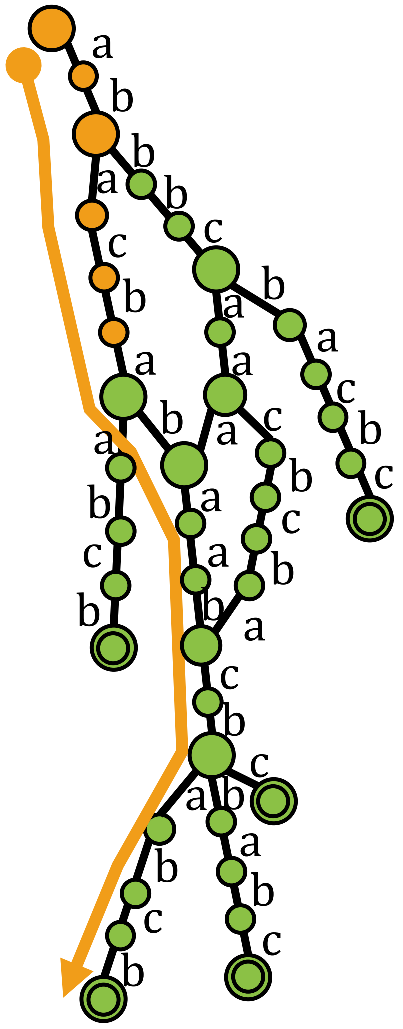
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



Pattern-searching query of a ADFA

- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \leq |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return “No”.
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return “Yes” iff v is an acceptable state.

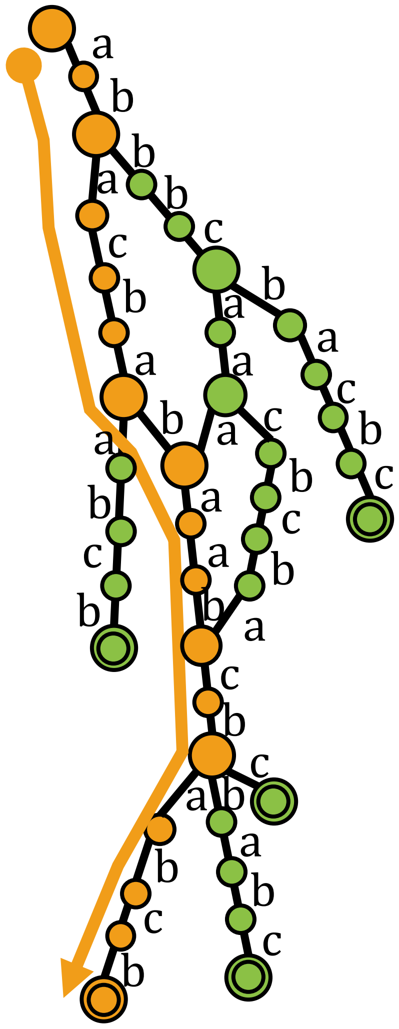
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



Pattern-searching query of a ADFA

- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \leq |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return "No".
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return "Yes" iff v is an acceptable state.

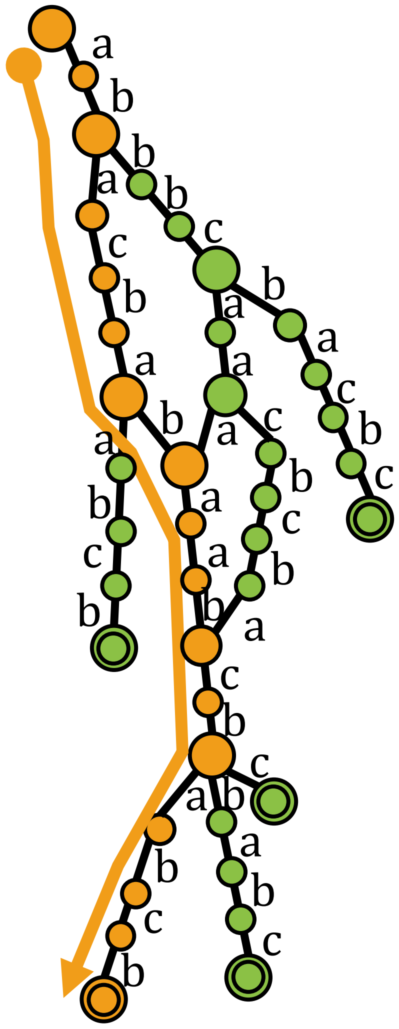
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



Pattern-searching query of a ADFA

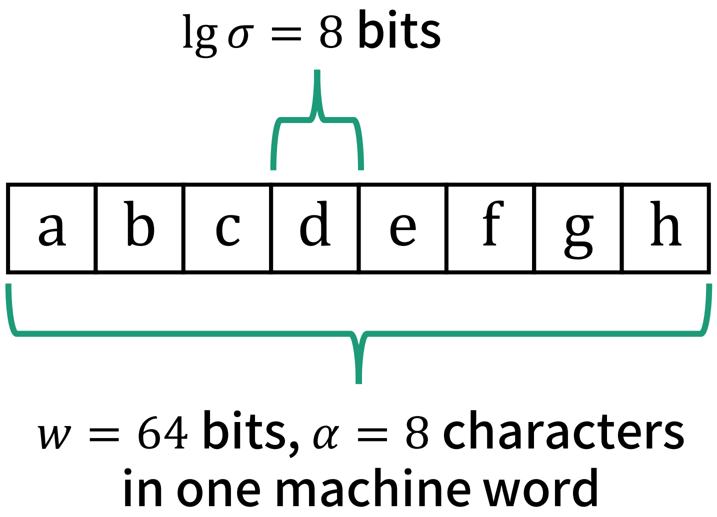
- **Pattern searching algorithm:**
  1. Let $v \leftarrow \mathrm{root}$ and $i \leftarrow 1$.
  2. While $i \leq |P|$, repeat the following procedure:
     1. Find the edge starting from v labeled by $P[i]$.
        - If there is no such edge, return "No".
     2. Move v along the edge and set $i \leftarrow i + 1$.
  3. Return "Yes" iff v is an acceptable state.

→ Overall time complexity: $\Omega(|P|)$



Packed string: storing multiple characters into a single machine word

- Modern computers have <u>64~512-bit</u> registers.
- The bit-width for one character is <u>typically less than the bit-width of registers.</u>
  - e.g. 8bits for ASCII, 2bits for nucleotides.

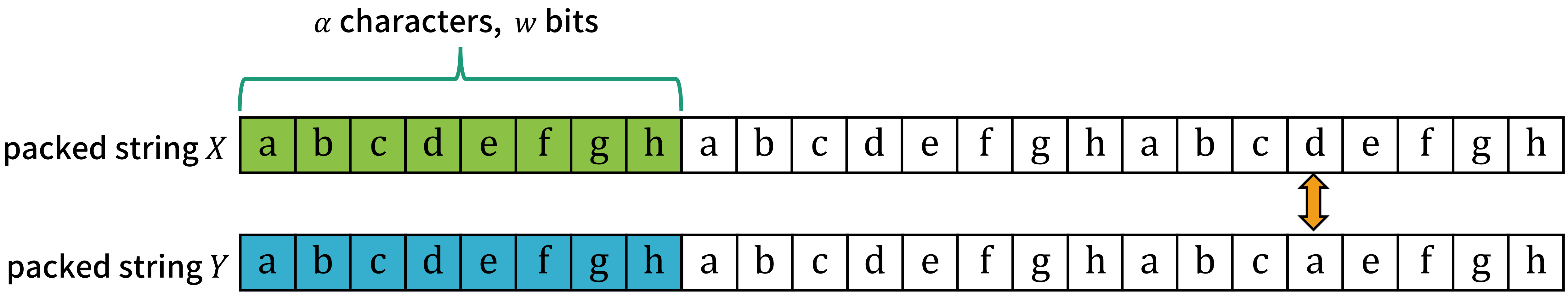


We can pack $\alpha = \dfrac{w}{\lg \sigma}$ characters into a single machine word (packed string).

Efficient comparison using packed strings

Packed string allows comparing consecutive α characters at once.

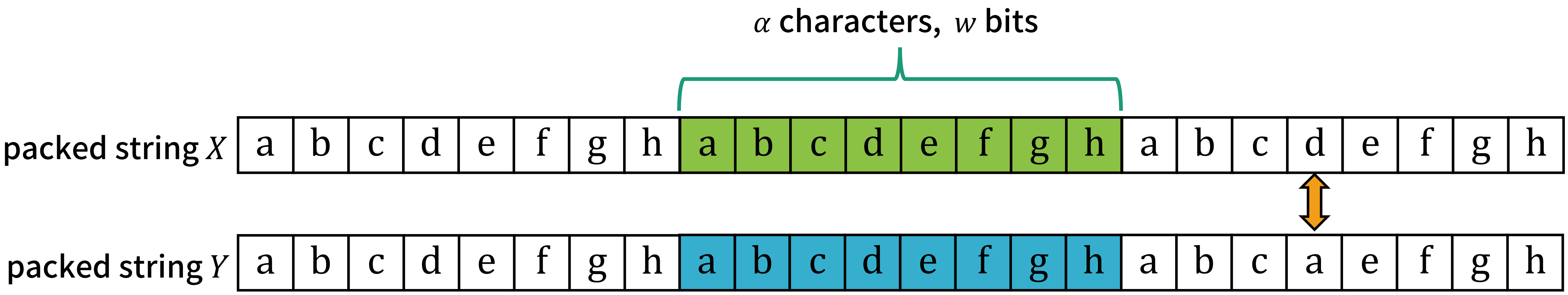
We can find the first mismatched index p of two packed strings in $O\left(\frac{p}{\alpha}\right)$ time.



Efficient comparison using packed strings

Packed string allows comparing consecutive α characters at once.

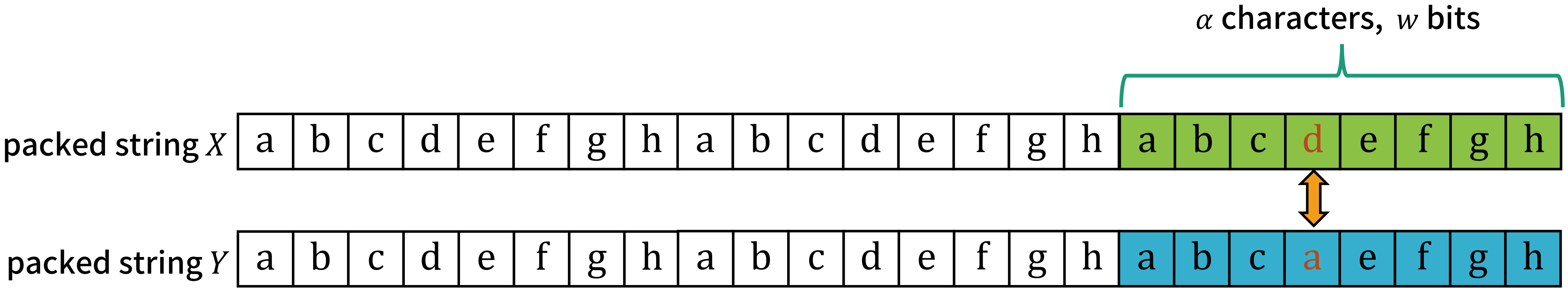
We can find the first mismatched index p of two packed strings in $O\left(\frac{p}{\alpha}\right)$ time.



Efficient comparison using packed strings

Packed string allows comparing consecutive α characters at once.

We can find the first mismatched index p of two packed strings in $O\left(\frac{p}{\alpha}\right)$ time.



Combining pattern-searching indexes and packed string

- **Trie: several methods for applying packed string have been proposed**
  - applying centroid path decomposition for speeding up in cache-oblivious model [Ferragina et al., 2008]
  - decomposition into micro trees of $O(w)$ size [Takagi et al., 2017]

- **ADFA: there is no method for combining packed strings and ADFAs**
  - Because all packed string methods for tries rely on its tree structure.

Our work: Packed ADFA

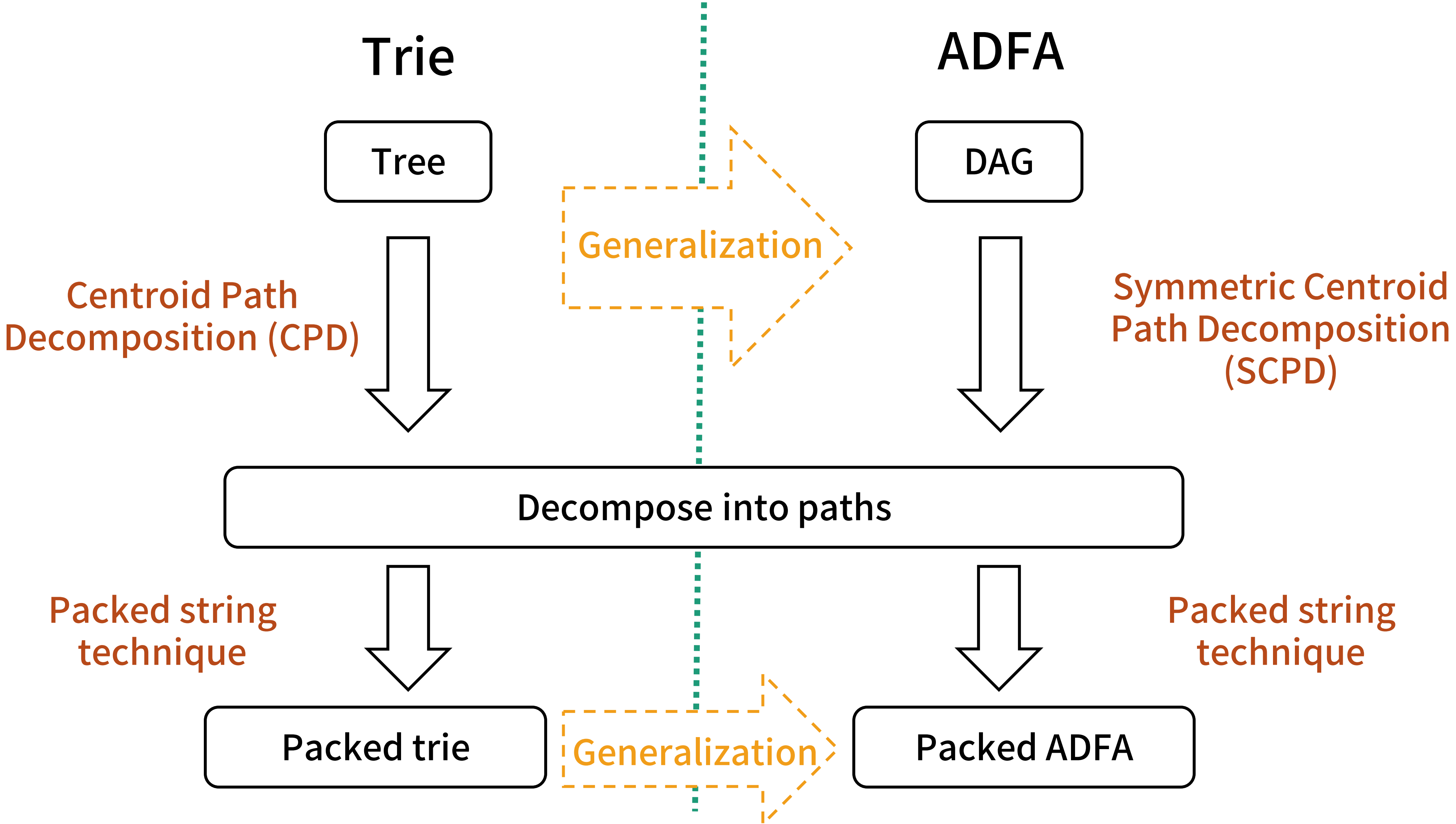
We propose packed ADFA: ADFA-based pattern-searching index with packed string.

Main advantages of Packed ADFA:

It is optimal when P is sufficiently long.

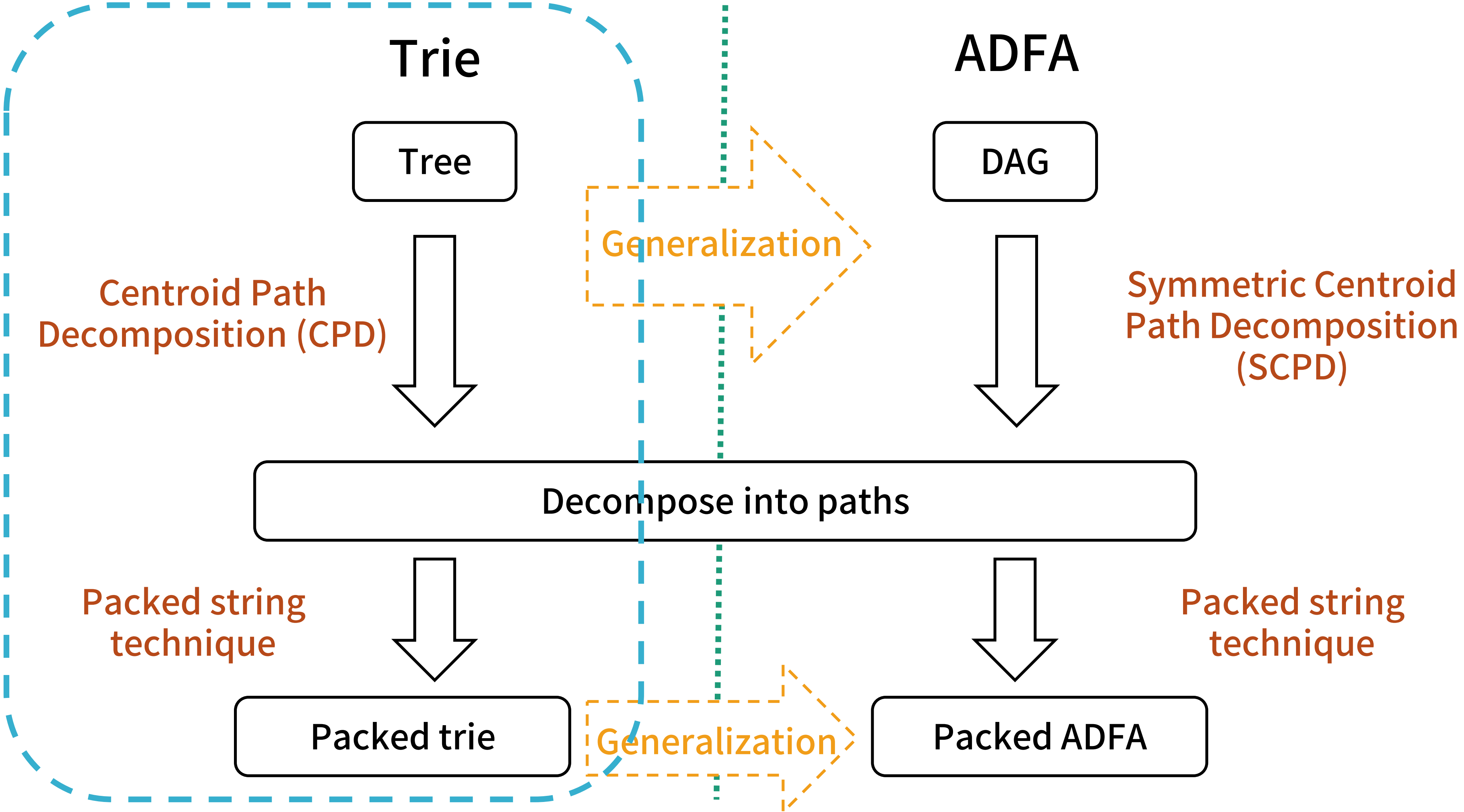
- performing pattern-searching in $O\left(\frac{|P|}{\alpha} + \lg k\right)$ time
- more compact than trie-based indexes

Overview of our methods



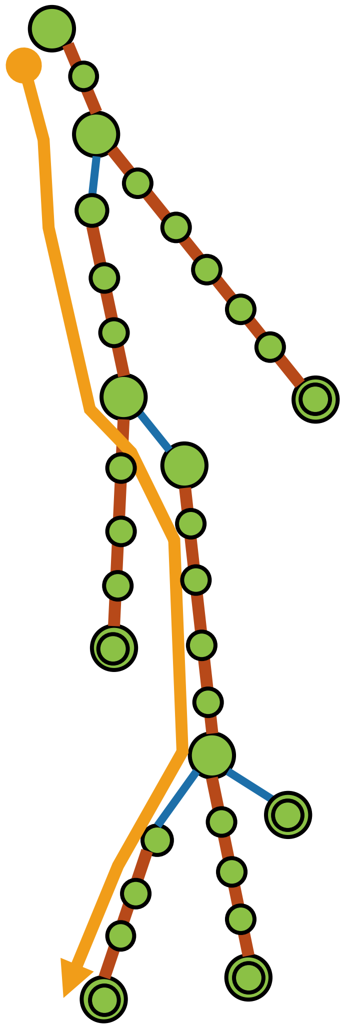
Packed pattern-searching for tries

Overview of our methods



Centroid-path decomposition (CPD) [Ferragina et al., 2008]: Decomposing a tree into paths

Main idea: Split the edge set into heavy edges and light edges to accelerate searching on the heavy path.



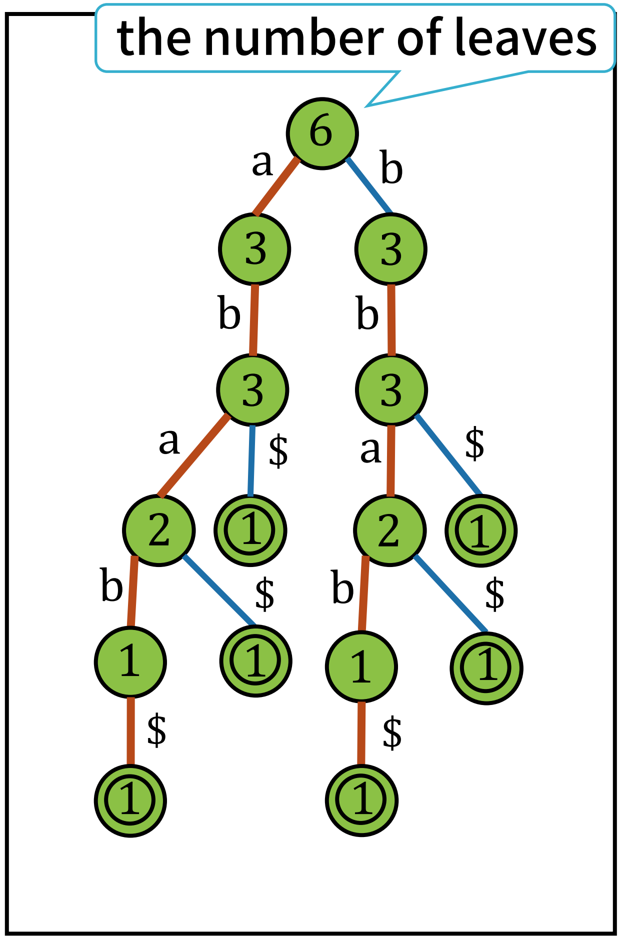
Properties of CPD:

- Heavy edges forms disjoint paths.
- Any path contains a small number of light edges.

Centroid-path decomposition (CPD) [Ferragina et al., 2008]: Decomposing a tree into paths

CPD classifies the edges of a tree into heavy edge and light edge to satisfy the following:

1. Each internal node u has exact one heavy edge starting with u.
   - Because of this rule, heavy edges does not have any branching and form disjoint paths.
2. The heavy edge starting from u connects to the vertex with the most leaves of u.

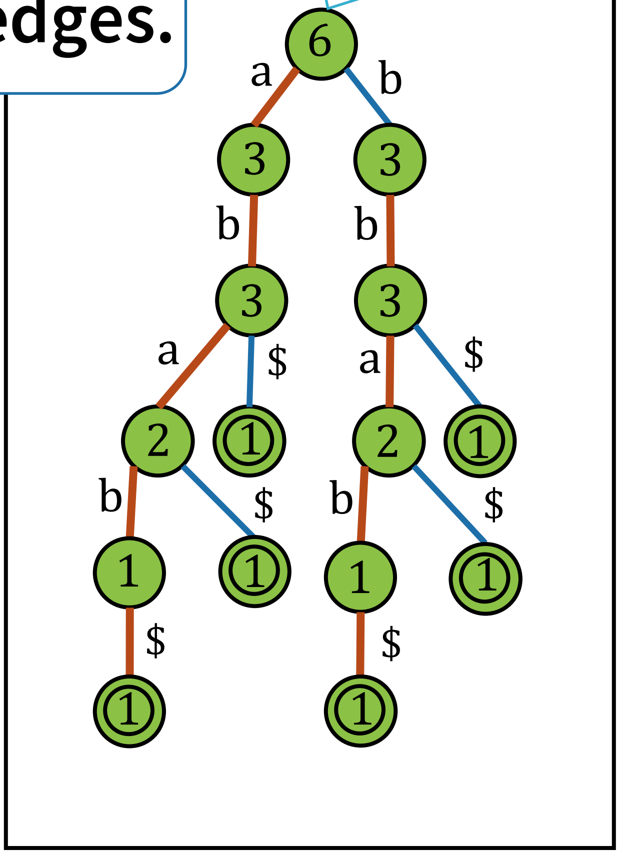


Centroid-path decomposition (CPD) [Ferragina et al., 2008]: Decomposing a tree into paths

Key Property

Each path in a tree of k leaves contains at most $\lfloor \lg k \rfloor$ light edges.

the number of leaves $s(v)$



Let $s(v)$ be the number of leaves of the subtree of v.

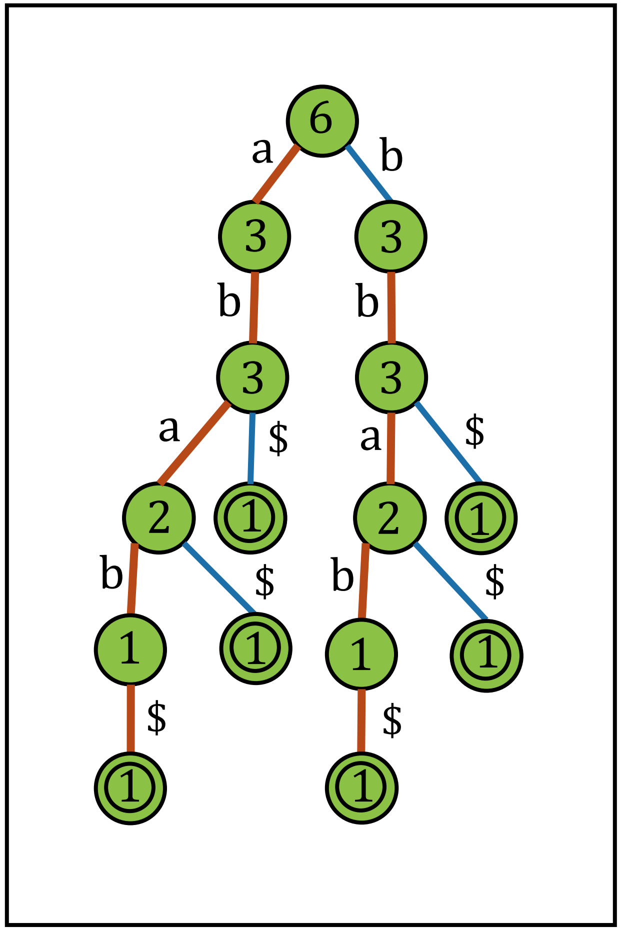
We can observe the key property by the following facts.

1. For any node v, $0 \le \lfloor \lg s(v) \rfloor \le \lfloor \lg k \rfloor$.
2. For any heavy edge (u, v), $\lfloor \lg s(u) \rfloor \ge \lfloor \lg s(v) \rfloor$.
3. For any light edge (u, v), $\lfloor \lg s(u) \rfloor > \lfloor \lg s(v) \rfloor$.

Packed Trie: packed pattern-searching index for trie

Packed trie stores the edge labels of the edges as follows:

- **Heavy paths**: by a packed string
  - Enables α times faster comparison between the pattern and the edge labels.
- **Light edges**: by biased search trees [Bent et al., 1985]
  - Enables finding edge (u, v) labeled by c in $O\left(\frac{\lg s(u)}{\lg s(v)}\right)$ time for given u and c.



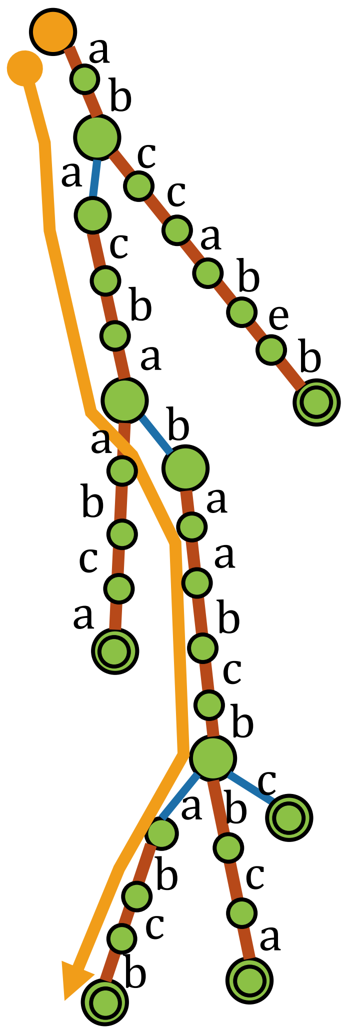
The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. **Compare the pattern string and its heavy path string and jump to the first mismatched point.**
   - We find mismatched point by comparison of packed strings.
2. **Jump along one light edge.**
   - We find the edge by biased search trees.

Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$

Heavy Path: $H = \mathrm{abccabeb}$

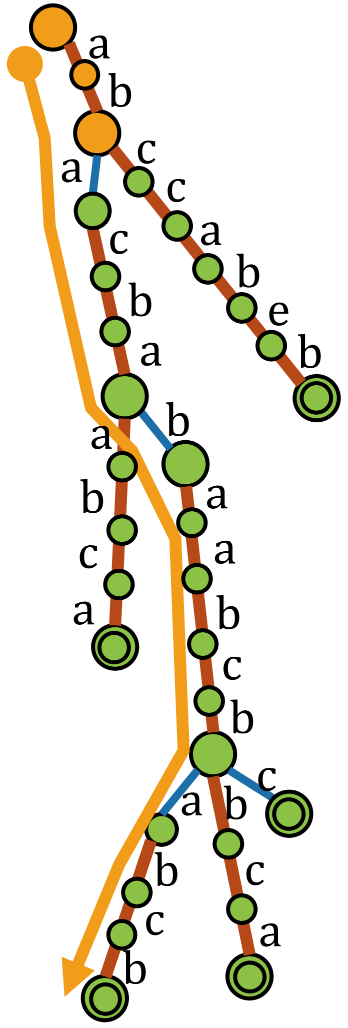


The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. **Compare the pattern string and its heavy path string and jump to the first mismatched point.**
   - We find mismatched point by comparison of packed strings.
2. **Jump along one light edge.**
   - We find the edge by biased search trees.

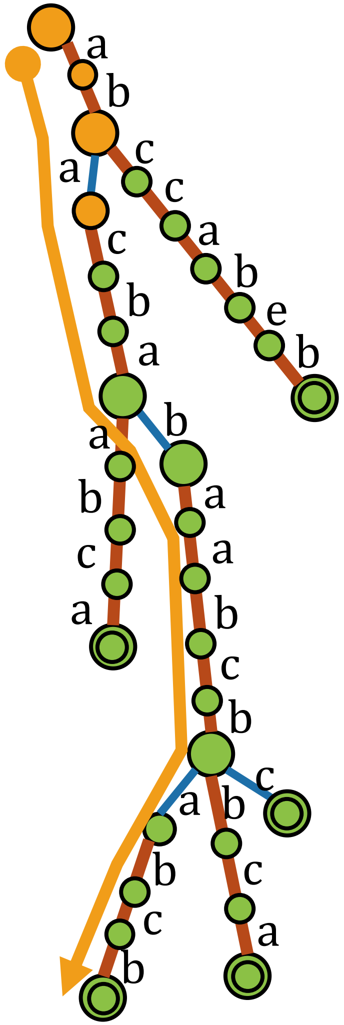
Ex. Pattern: $P =$ abacbabaabcbabcb
Heavy Path: $H =$ abccabeb



The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. **Compare the pattern string and its heavy path string and jump to the first mismatched point.**
   - We find mismatched point by comparison of packed strings.
2. **Jump along one light edge.**
   - We find the edge by biased search trees.



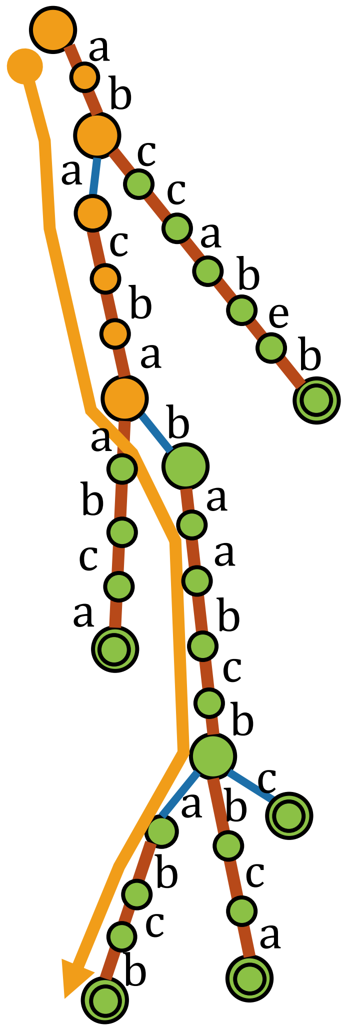
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$

Heavy Path: $H = \mathrm{cbaabca}$

The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. **Compare the pattern string and its heavy path string and jump to the first mismatched point.**
   - We find mismatched point by comparison of packed strings.
2. **Jump along one light edge.**
   - We find the edge by biased search trees.



Ex. Pattern: $P =$ abacbabaabcbabcb
Heavy Path: $H =$ cbaabca

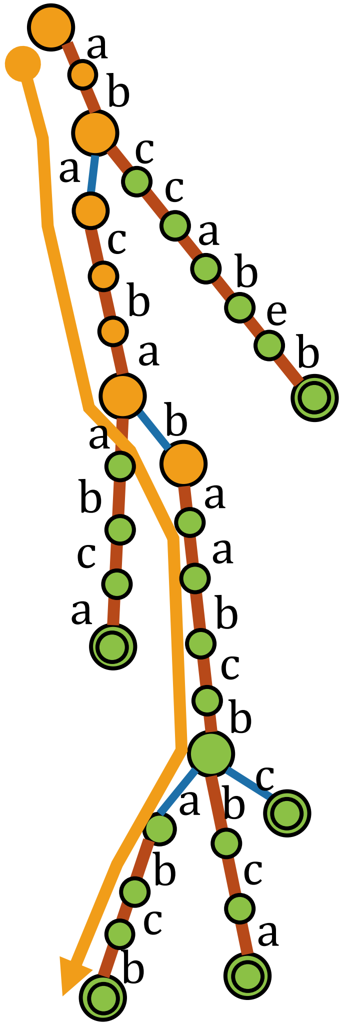
The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. Compare the pattern string and its heavy path string and jump to the first mismatched point.
   - We find mismatched point by comparison of packed strings.
2. Jump along one light edge.
   - We find the edge by biased search trees.

Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$

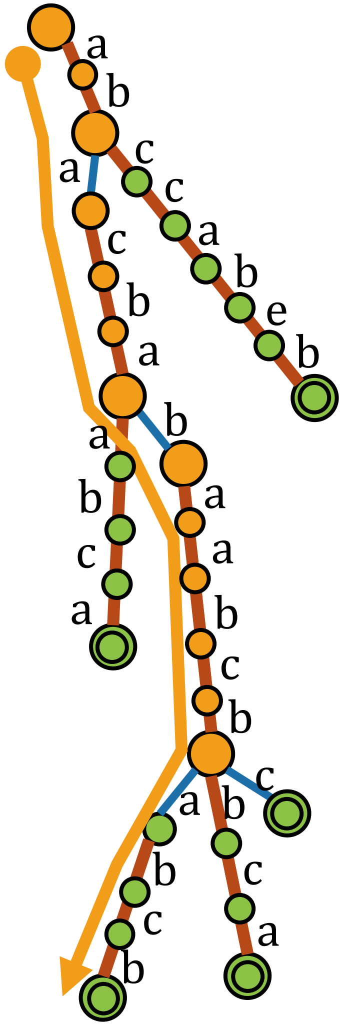
Heavy Path: $H = \mathrm{aabcbbca}$



The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. Compare the pattern string and its heavy path string and jump to the first mismatched point.
   - We find mismatched point by comparison of packed strings.
2. Jump along one light edge.
   - We find the edge by biased search trees.



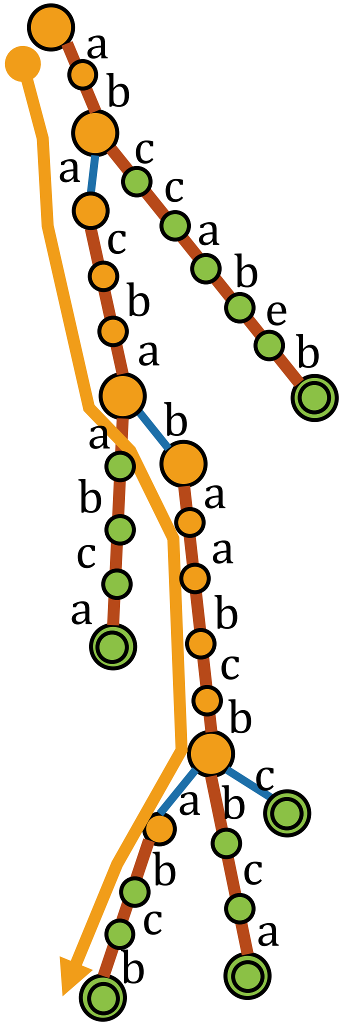
Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$

Heavy Path: $H = \mathrm{aabcbbca}$

The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. Compare the pattern string and its heavy path string and jump to the first mismatched point.
   - We find mismatched point by comparison of packed strings.
2. Jump along one light edge.
   - We find the edge by biased search trees.



Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$

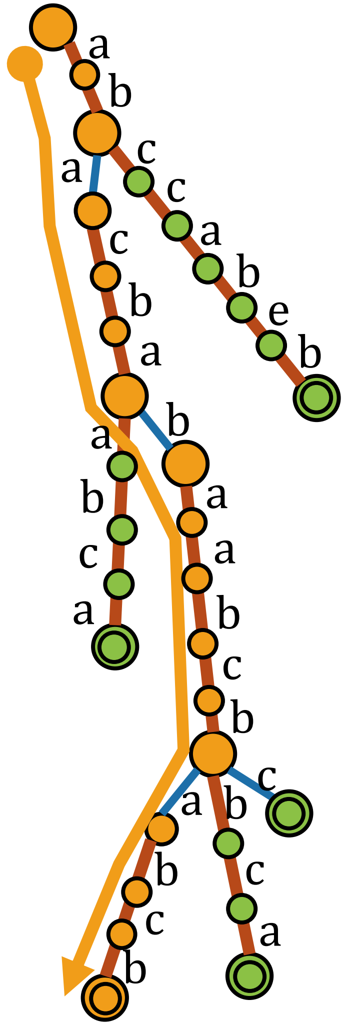
Heavy Path: $H = \mathrm{bcb}$

The algorithm for pattern-searching with packed trie

Repeat the following procedure:

1. **Compare the pattern string and its heavy path string and jump to the first mismatched point.**
   - We find mismatched point by comparison of packed strings.
2. **Jump along one light edge.**
   - We find the edge by biased search trees.

Ex. Pattern: $P = \mathrm{abacbabaabcbabcb}$



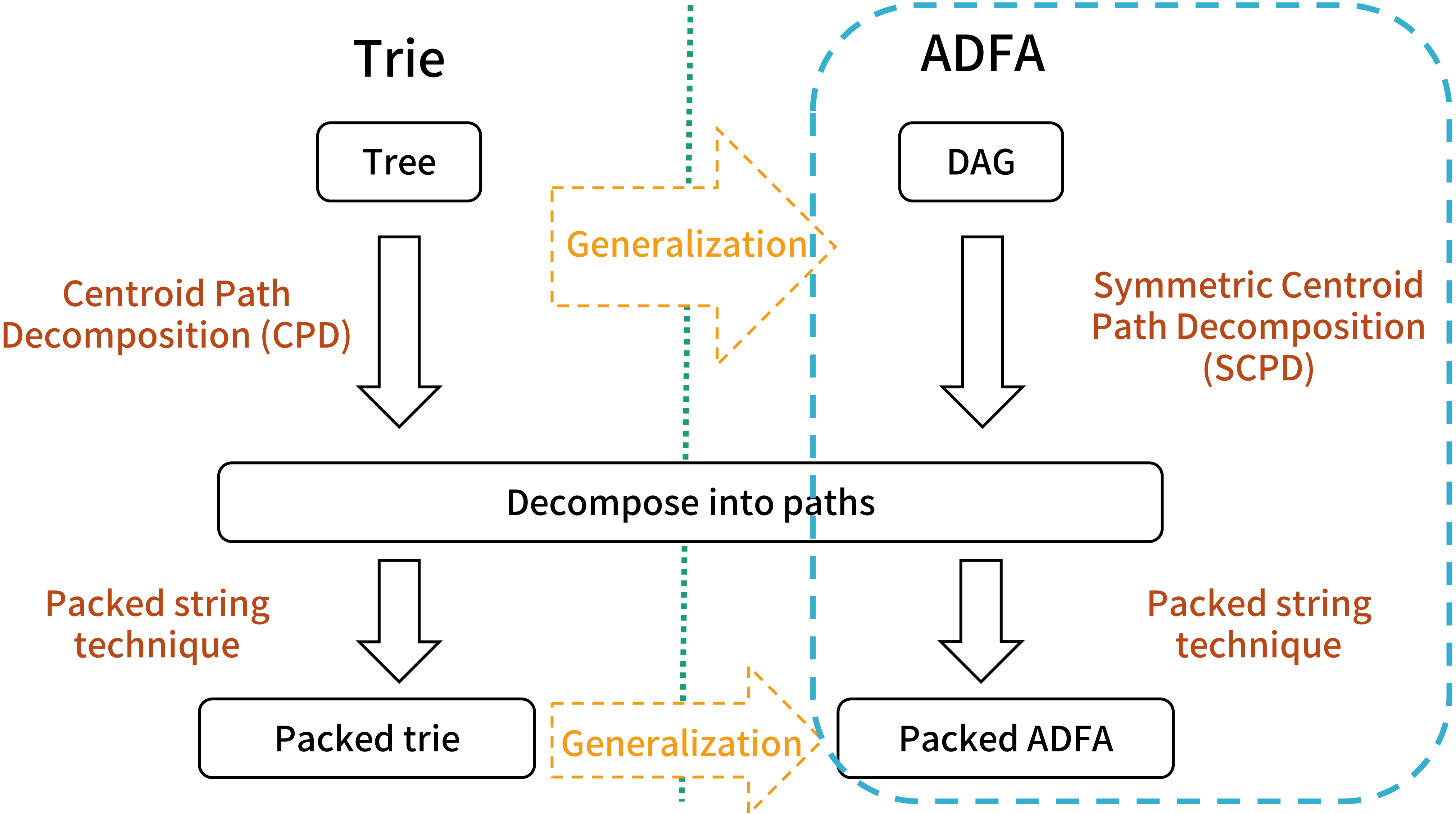
Time complexity for pattern-searching

- Time complexity for heavy edges: $O\left(\frac{|P|}{\alpha} + \lg k\right)$
  - The number of heavy paths is $O(\lg k)$, and searching time for each heavy path is α times faster than normal.
- Time complexity for light edges: $O(\lg k)$
  - Move along an edge (u, v) : $O\left(\lg \frac{s(u)}{s(v)}\right)$ time by biased search trees
  - Since the sequence of subtree size on a path is decreasing, the total time complexity is bounded by telescoping sum method. $\left(O\left(\lg \frac{s(u)}{s(v)} + \lg \frac{s(v)}{s(w)}\right) = O\left(\lg \frac{s(u)s(v)}{s(v)s(w)}\right) = O\left(\lg \frac{s(u)}{s(w)}\right)\right)$

→ **Overall time complexity:** $O\left(\frac{|P|}{\alpha} + \lg k\right)$

Packed pattern-searching for ADFA

Overview of our methods



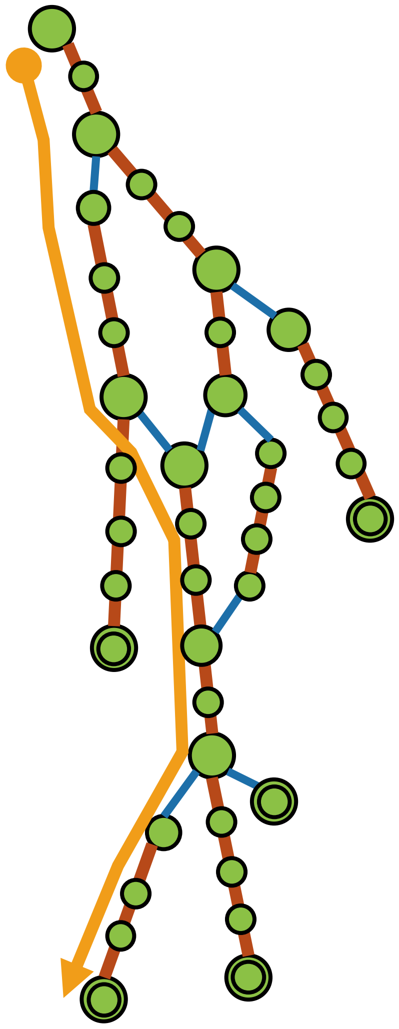
Symmetric centroid path decomposition (SCPD) [Ganardi et al., 2022]: A generalized CPD for DAGs

We cannot directly apply CPD for DAGs.

→ We use generalized method called SCPD.

Properties of SCPD (as with normal CPD)

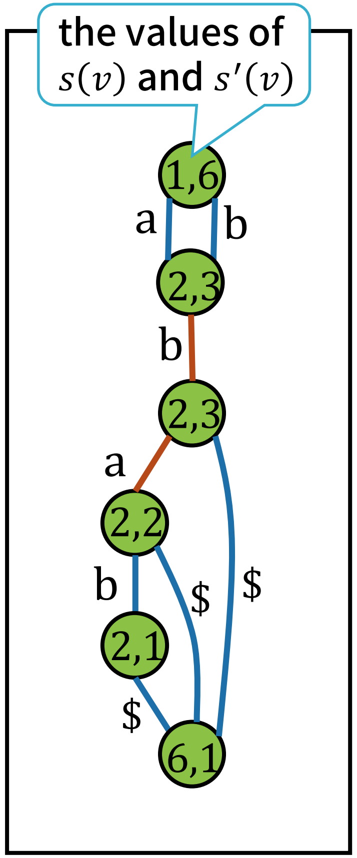
- Heavy edges forms disjoint paths.
- Any path contains a small number of light edges.



Symmetric centroid path decomposition (SCPD) [Ganardi et al., 2022]: A generalized CPD for DAGs

SCPD classifies the edges of a DAG into heavy edges and light edges by the following rules:

- **Definitions:**
  - $s(v)$: the number of paths from the root to v
  - $s'(v)$: the number of paths from v to the sink
- **Classification rules:**
  1. Iff an edge (u, v) satisfies both $\lfloor \lg s(u) \rfloor = \lfloor \lg s(v) \rfloor$ and $\lfloor \lg s'(u) \rfloor = \lfloor \lg s'(v) \rfloor$, (u, v) is a heavy edge.
     - Because of this rule, all heavy edges have different starting/ending point and heavy edges form disjoint paths.
  2. Otherwise, (u, v) is a light edge.



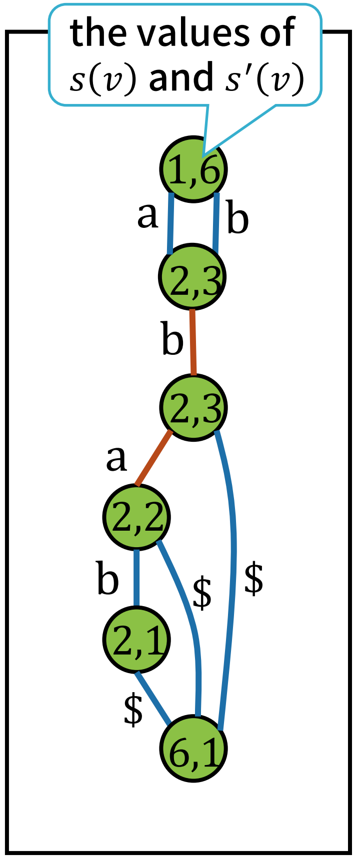
Symmetric centroid path decomposition (SCPD) [Ganardi et al., 2022]: A generalized CPD for DAGs

Key Property

Each path in a DAG having k root-sink paths contains at most $2\lfloor \lg k \rfloor$ light edges.

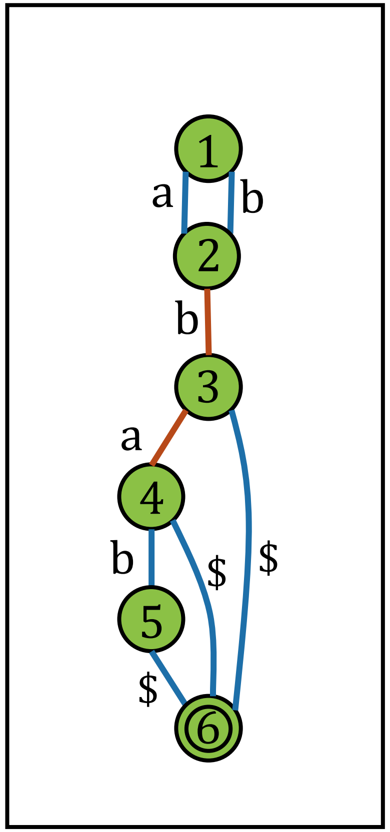
We can observe the key property by the following facts.

1. **For any node v, $0 \le \lfloor \lg s(v) \rfloor, \lfloor \lg s'(v) \rfloor \le \lfloor \lg k \rfloor$.**
2. **For any heavy edge (u, v), $\lfloor \lg s(u) \rfloor = \lfloor \lg s(v) \rfloor$ and $\lfloor \lg s'(u) \rfloor = \lfloor \lg s'(v) \rfloor$ holds.**
3. **For any light edge (u, v), $\lfloor \lg s(u) \rfloor > \lfloor \lg s(v) \rfloor$ or $\lfloor \lg s'(u) \rfloor < \lfloor \lg s'(v) \rfloor$ holds.**



Packed ADFA: packed pattern-searching index for ADFA

Packed ADFA stores edge labels of each heavy path by a packed string, and store light edges by biased search trees.



Implementation techniques for reducing memory usage:

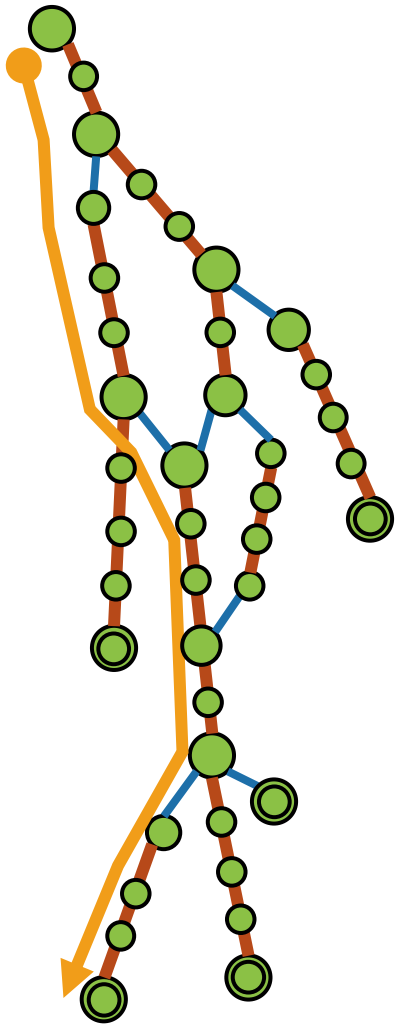
- Concatenating all heavy edge labels as a one string.
- Reducing the number of biased search trees.
  - We omit biased search trees for a node having no light edges starting from it.

The algorithm for pattern-searching with packed ADFA (as with packed trie)

Repeat the following procedure:

1. Compare the pattern string and its heavy path string and jump to the first mismatched point.
   - We find mismatched point by comparison of packed strings.
2. Jump along with one light edge.
   - We find the edge by biased search trees.

→ Overall Time Complexity: $O\left(\frac{|P|}{\alpha} + \lg k\right)$



Experiments

Experimental Settings

- **Evaluation measures:**
  - Memory: the amount of memory consumption for created indexes.
  - Time: total elapsed time for pattern-searching for all $S_i \in \mathcal{S}$.
- **Datasets: three types of dataset (URL, city, prot)**
- **Data structures:**
  - Normal: standard trie / ADFA
  - Pref: minimal prefix trie [Aoe, 1989]
  - Packed: packed trie / ADFA
- **Bit width: $w = 64$ and $\lg \sigma = 8$ bits.**

It handles unary paths to leaves by packed strings.

Characteristics of datasets and the size of their tries / ADFAs

The characteristics for each data set and the size of tries/ADFAs.

	dictionary				Trie		ADFA	
	σ	k	total len.	ave. len.	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V\rvert$	$\lvert E\rvert$
url	93	862,665	72,540,387	84.089	10,146,553	10,146,552	1,612,336	2,040,555
city	78	177,030	1,970,082	11.183	846,550	846,549	198,195	333,800
protein	25	157,237	46,687,247	295.046	35,028,185	35,028,184	32,905,500	33,030,196

- All datasets have about 10^5~10^6 strings and average length is about 10~300.
- ADFAs can reduce the graph size about 2.5~5 times compared to tries for url and city dataset.

Memory usage of tries / ADFAs

The memory usage of tries/ADFAs.

	Memory [MiB]				
	Tries			ADFAs	
	Normal	Pref	Packed	Normal	Packed
url	59.269	20.751	13.625	11.676	4.773
city	4.945	2.045	1.618	1.910	1.270
protein	204.609	38.729	34.130	189.000	32.757

- Packed ADFA is about 4~13 times memory efficient than normal tries.
- We observe both packed data structure and converting tries to minimal DFAs are effective.

Computation time for pattern-searching

The computation time for pattern searching of tries/ADFAs.

	Time [ms]				
	Tries			ADFAs	
	Normal	Pref	Packed	Normal	Packed
url	5911.507	5056.601	1046.047	5691.689	1219.044
city	221.616	179.772	133.276	233.288	161.726
protein	2614.497	363.963	175.183	2780.915	313.974

- For all datasets, packed trie is the fastest, followed by packed ADFAs.
- Packed indexes performs pattern-searching 1.3~5 times faster than using standard tries even if pattern is not sufficiently long.

Conclusion

- We proposed packed ADFA, a time and space-efficient index for pattern-searching.
  - It can achieve optimal searching time for sufficiently long pattern.
  - It is more space-efficient than trie-based indexes.
- Experimental result shows that it is practically efficient.
  - It performs faster pattern-searching even if the pattern is not sufficiently long.
  - Both packing techniques and minimizing tries to ADFA contributes to reducing memory consumption.