Efficient Construction of Hierarchical Overlap Graphs

Sung Gwan Park
Seoul National University, Korea
Bastien Cazaux
LIRMM, Univ Montpellier, CNRS, France
Kunsoo Park
Seoul National University, Korea
Eric Rivals
LIRMM, Univ Montpellier, CNRS, France

Outline

- Introduction
- Preliminaries
- Main algorithm
- Improvement using segment tree

Introduction

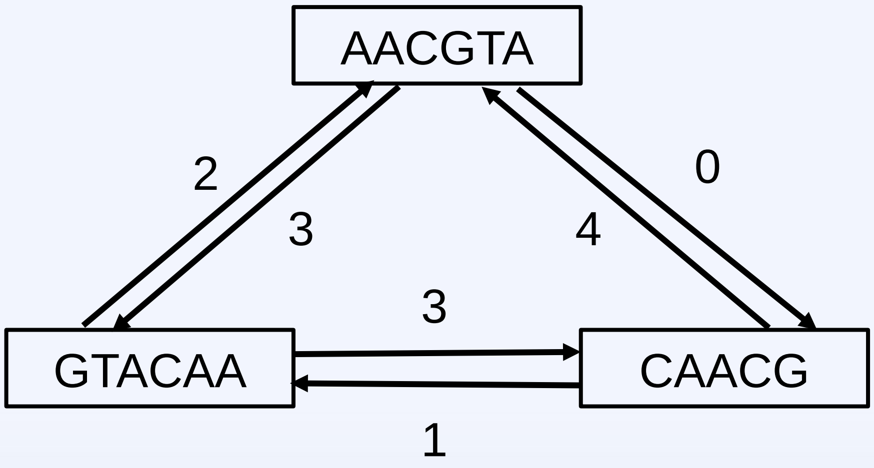
- Genome sequencing
  - DNA assembly: Obtaining a whole genome sequence from sequencing reads
  - Seeking some path in a graph that encodes suffix-prefix overlaps

- Overlap encoding graphs
  - Overlap graph
  - De Bruijn graph

Introduction

- Genome sequencing
  - DNA assembly: Obtaining a whole genome sequence from sequencing reads
  - Seeking some path in a graph that encodes suffix-prefix overlaps

- Overlap encoding graphs
  - Overlap graph
  - De Bruijn graph



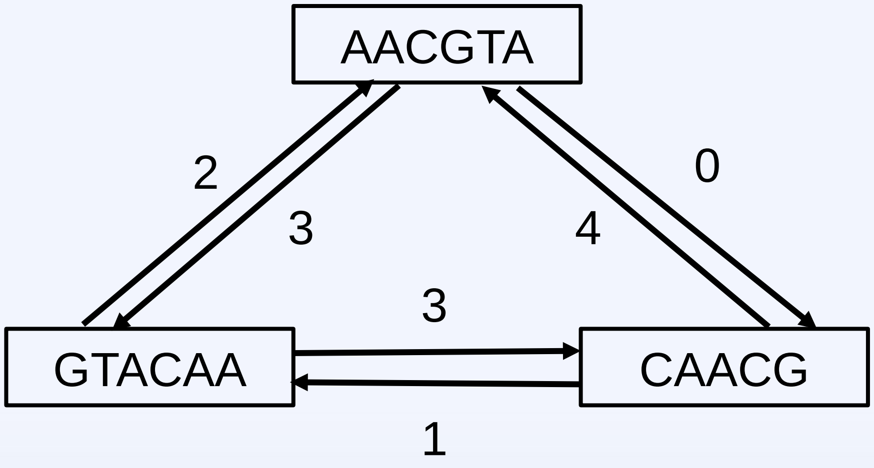
Shortest Superstring problem

- Shortest Superstring problem
  - Let P be a set of input strings.
  - Find the shortest string that contains all input strings as substring.

- Equivalent to finding a maximum weighted Hamiltonian path in the overlap graph.

- Example P := { AACGTA, GTACAA, CAACG }

```
CAACGTACAA
CAACG
 AACGTA
    GTACAA
```



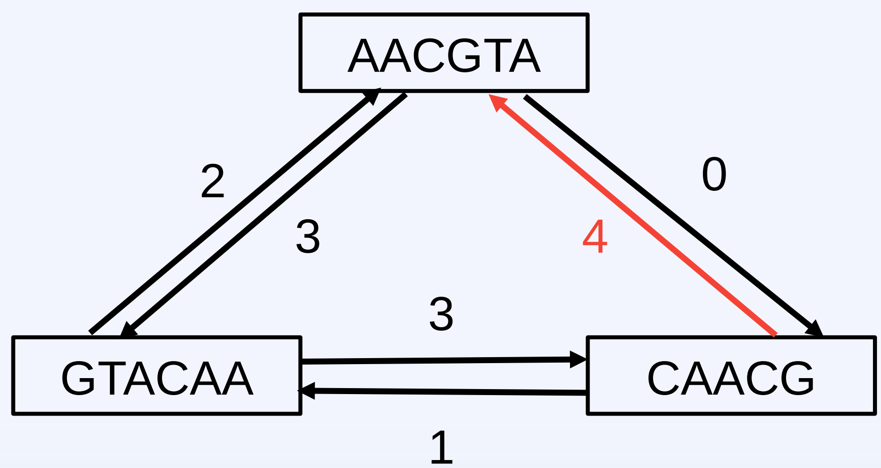
Shortest Superstring problem

- Shortest Superstring problem
  - Let P be a set of input strings.
  - Find the shortest string that contains all input strings as substring.

- Equivalent to finding a maximum weighted Hamiltonian path in the overlap graph.

- Example P := { AACGTA, GTACAA, CAACG }

```
CAACGTACAA
CAACG
 AACGTA
    GTACAA
```



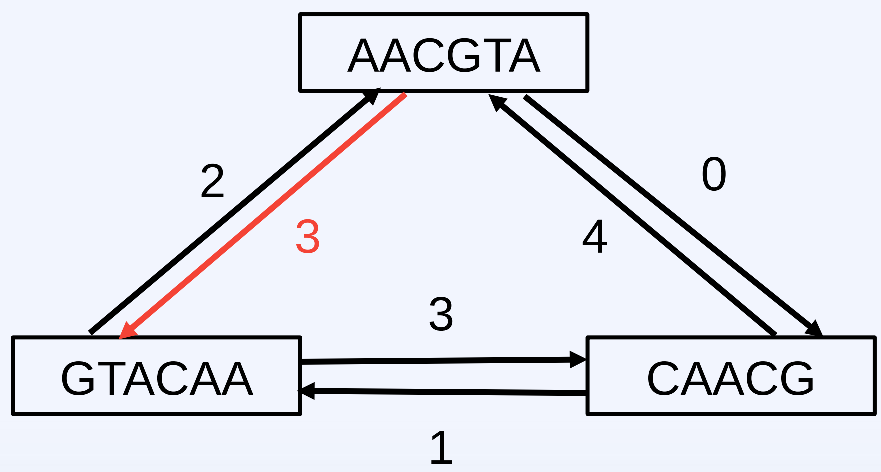
Shortest Superstring problem

- Shortest Superstring problem
  - Let P be a set of input strings.
  - Find the shortest string that contains all input strings as substring.

- Equivalent to finding a maximum weighted Hamiltonian path in the overlap graph.

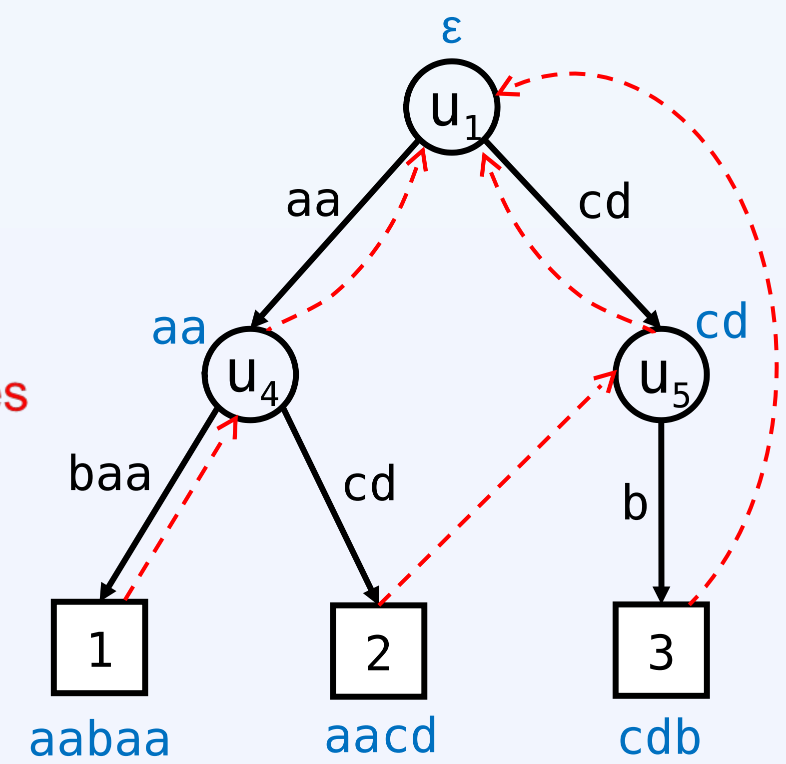
- Example P := { AACGTA, GTACAA, CAACG }

```
CAACGTACAA
CAACG
  AACGTA
     GTACAA
```



Hierarchical Overlap Graph (HOG)

- Hierarchical overlap graph (HOG)
  - First proposed by Cazaux and Rivals
  - Given a set of strings,
    - Nodes: Maximal overlaps between strings
    - Arcs: Prefix or suffix relations between nodes
  - We can divide arcs into tree edges (prefix relations) and failure links (suffix relations).



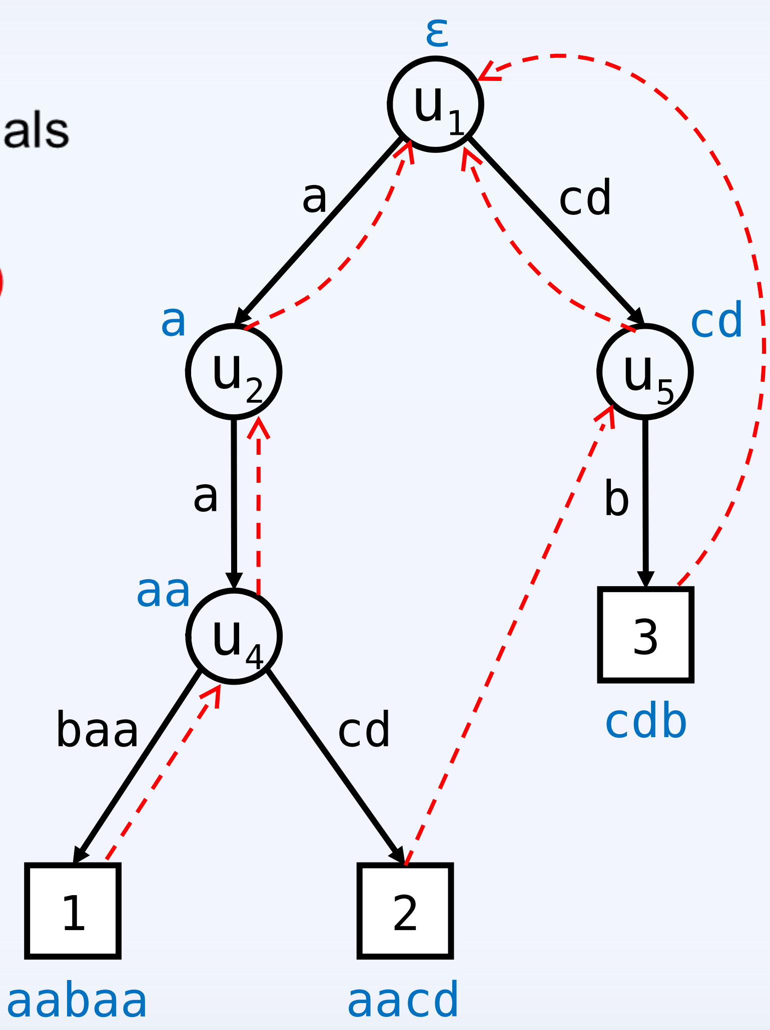
- Example
  - $S = \{aabaa, aacd, cdb\}$

Hierarchical Overlap Graph (HOG)

- Extended HOG (EHOG)
  - First proposed by Cazaux and Rivals
  - Given a set of strings,
    - Nodes: (Possibly not maximal) Overlaps between strings
    - Arcs: Prefix or suffix relations between nodes

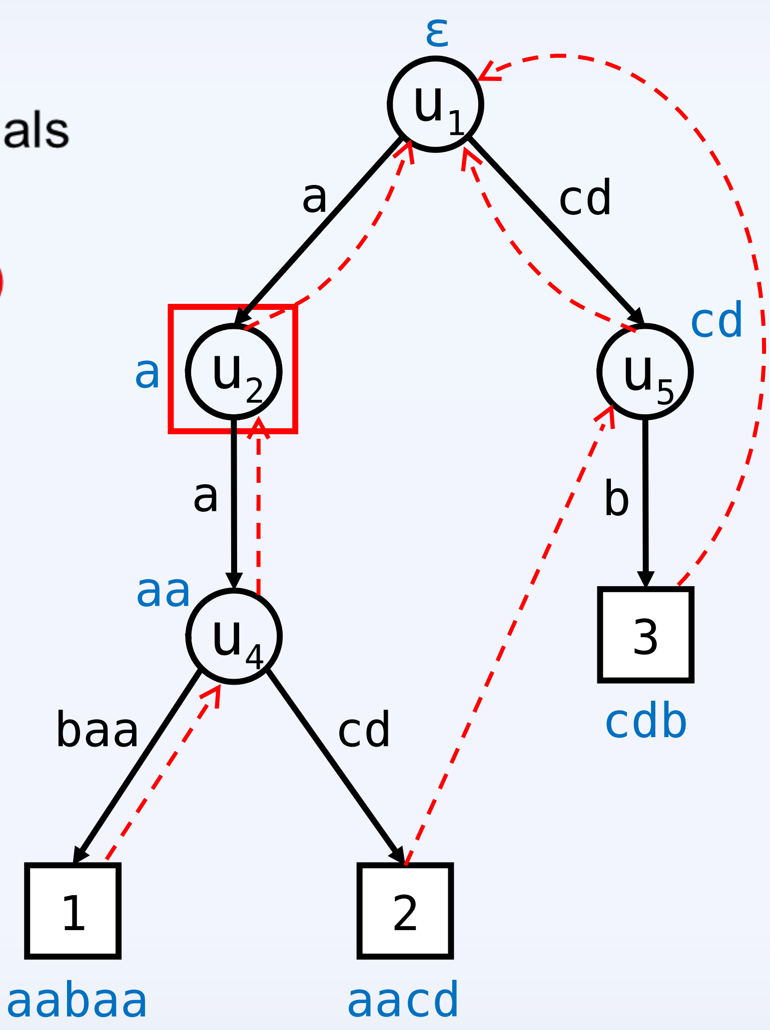
- Example
  - $S = \{aabaa, aacd, cdb\}$

- EHOG may use considerably more space than HOG.



Hierarchical Overlap Graph (HOG)

- Extended HOG (EHOG)
  - First proposed by Cazaux and Rivals
  - Given a set of strings,
    - Nodes: (Possibly not maximal) Overlaps between strings
    - Arcs: Prefix or suffix relations between nodes
- Example
  - $S = \{aabaa, aacd, cdb\}$
- EHOG may use considerably more space than HOG.



Hierarchical Overlap Graph (HOG)

- There are many advantages of HOG.

  ||P|| denotes the sum of lengths of strings in P

- HOG uses less space than overlap graph.
  - Overlap graph: $O(||P|| + n^2)$
  - HOG: $O(||P||)$

- HOG has more information than the overlap graph.
  - HOG encodes a relationship between the overlaps.

    e.g. All identical overlaps are encoded into a unique node in HOG.

- HOG has a great potential in studying the shortest superstring problem.

HOG: construction

- EHOG can be constructed in $O(||P||)$ time and space. (Cazaux B. and Rivals E., 2020)
- HOG uses more time and space.
  - Time: $O(||P|| + n^2)$
  - Space: $O(||P|| + n \times \min(n, \max\{|s| : s \in P)\}))$
- We present an algorithm using less time and space.
  - Time: $O(||P|| \log n)$ or $O(||P|| \frac{\log n}{\log \log n})$
  - Space: $O(||P||)$

Preliminaries

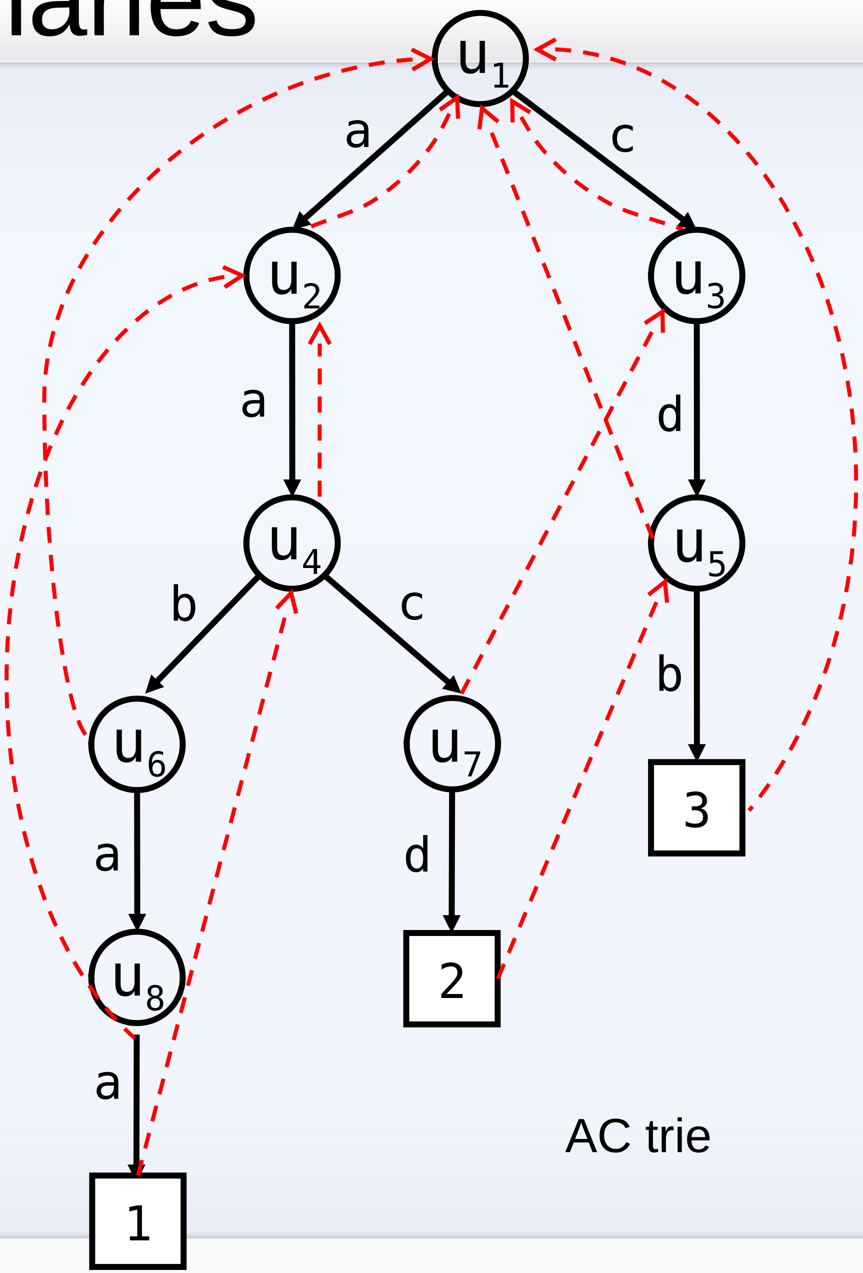
- Given a set of strings $P = \{s_1, s_2, \ldots, s_\mathrm{n}\}$,
  - $Ov^+(P)$: Set of all overlaps from s_i to s_j for $1 \leq i, j \leq n$
  - $Ov(P)$: Set of the longest overlap from s_i to s_j for $1 \leq i, j \leq n$

- In order to compute HOG, we need to compute $Ov(P)$.

- If we have $Ov(P)$ and $EHOG(P)$, we can compute $HOG(P)$ in $O(||P||)$ time. (Cazaux B. and Rivals E., 2020)

Preliminaries

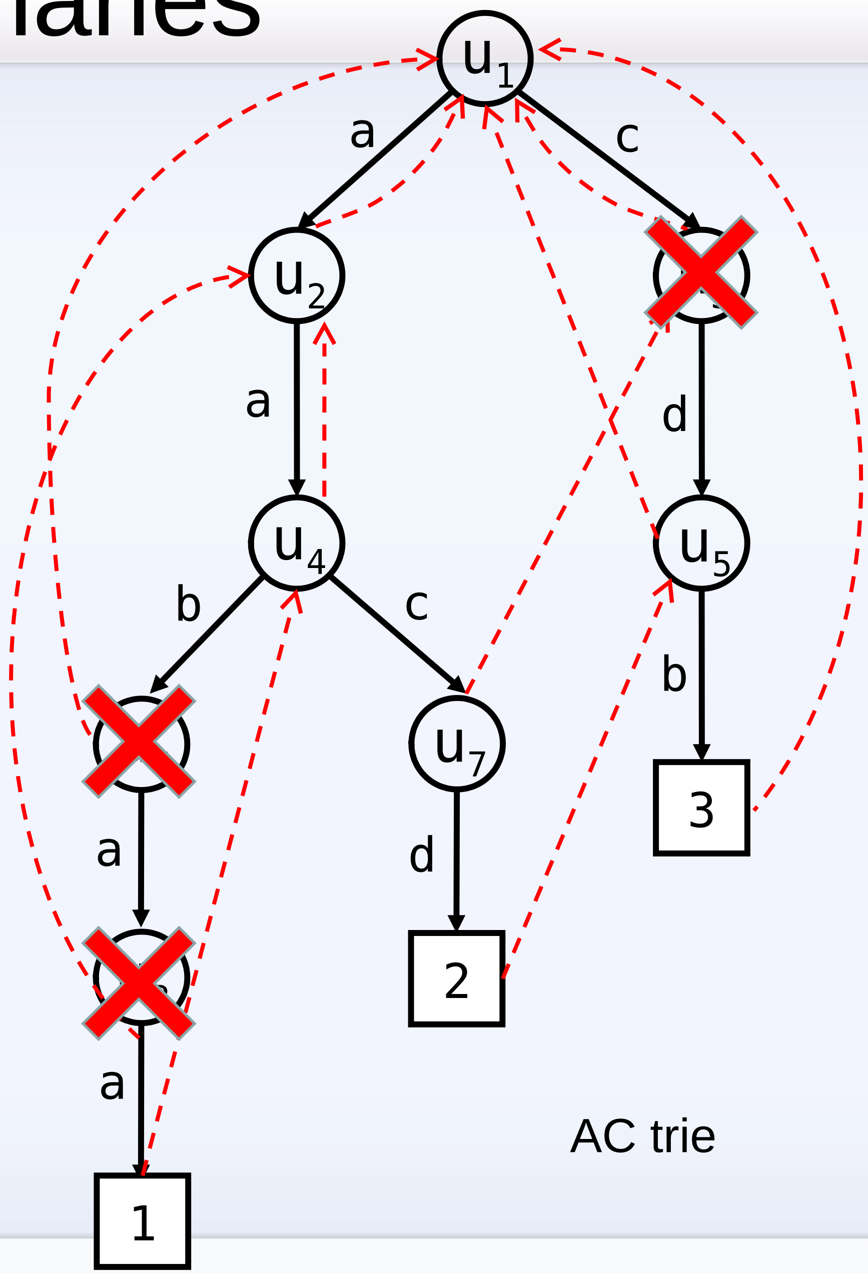
- EHOG is a contracted form of an Aho-Corasick trie.
- HOG is a contracted form of an EHOG.
- Example
  - $S = \{aabaa, aacd, cdb\}$



AC trie

Preliminaries

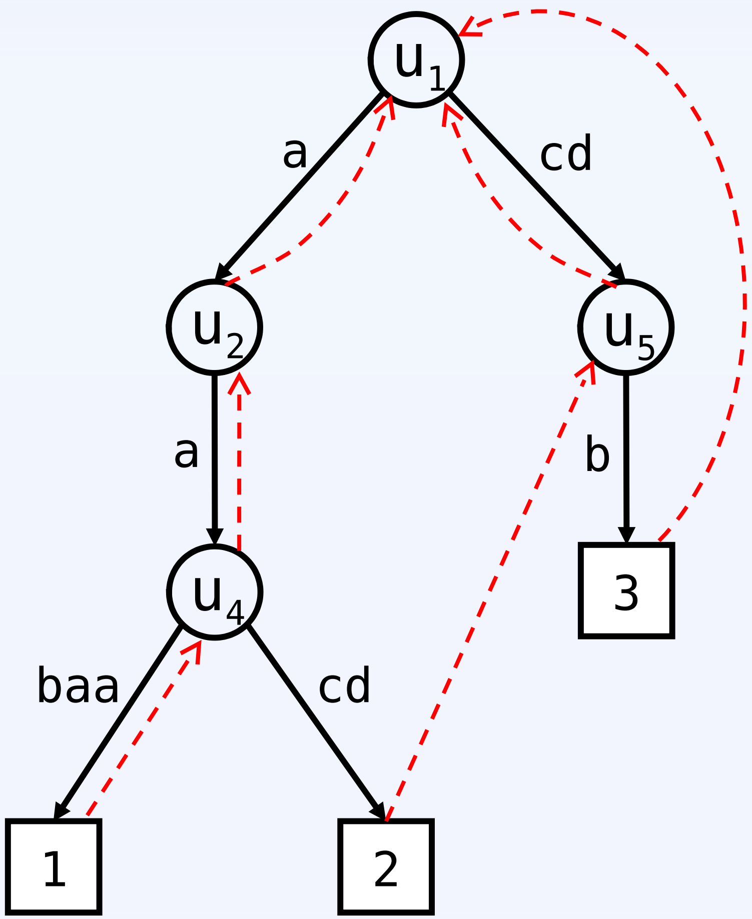
- EHOG is a contracted form of an Aho-Corasick trie.
- HOG is a contracted form of an EHOG.
- Example
  - $S = \{aabaa, aacd, cdb\}$



AC trie

Preliminaries

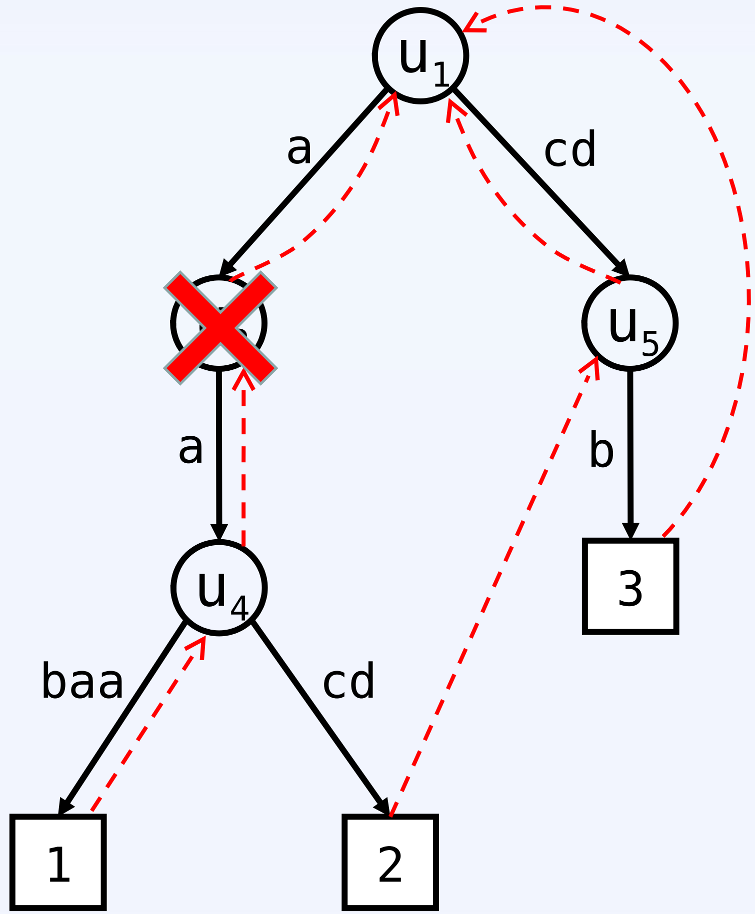
- EHOG is a contracted form of an Aho-Corasick trie.
- HOG is a contracted form of an EHOG.
- Example
  - $S = \{aabaa, aacd, cdb\}$



EHOG

Preliminaries

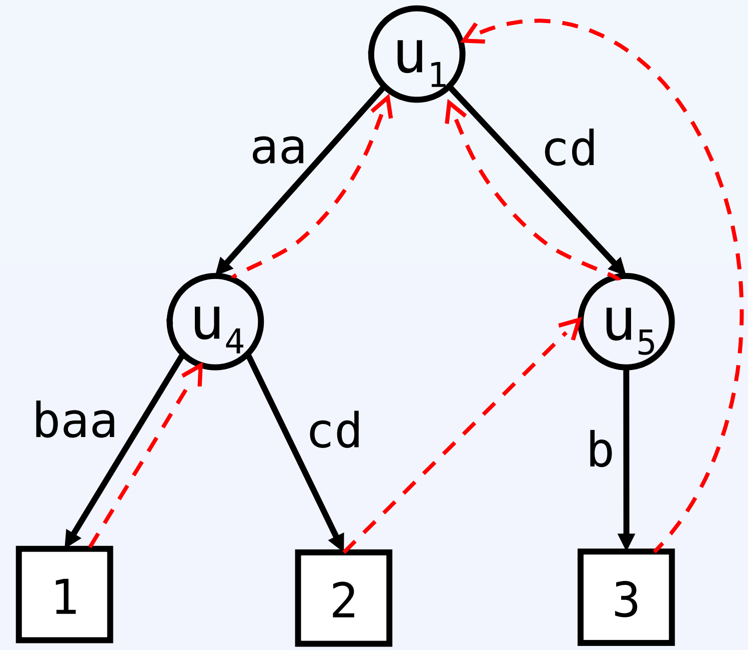
- EHOG is a contracted form of an Aho-Corasick trie.
- HOG is a contracted form of an EHOG.
- Example
  - $S = \{aabaa, aacd, cdb\}$



EHOG

Preliminaries

- EHOG is a contracted form of an Aho-Corasick trie.
- HOG is a contracted form of an EHOG.
- Example
  - $S = \{aabaa, aacd, cdb\}$



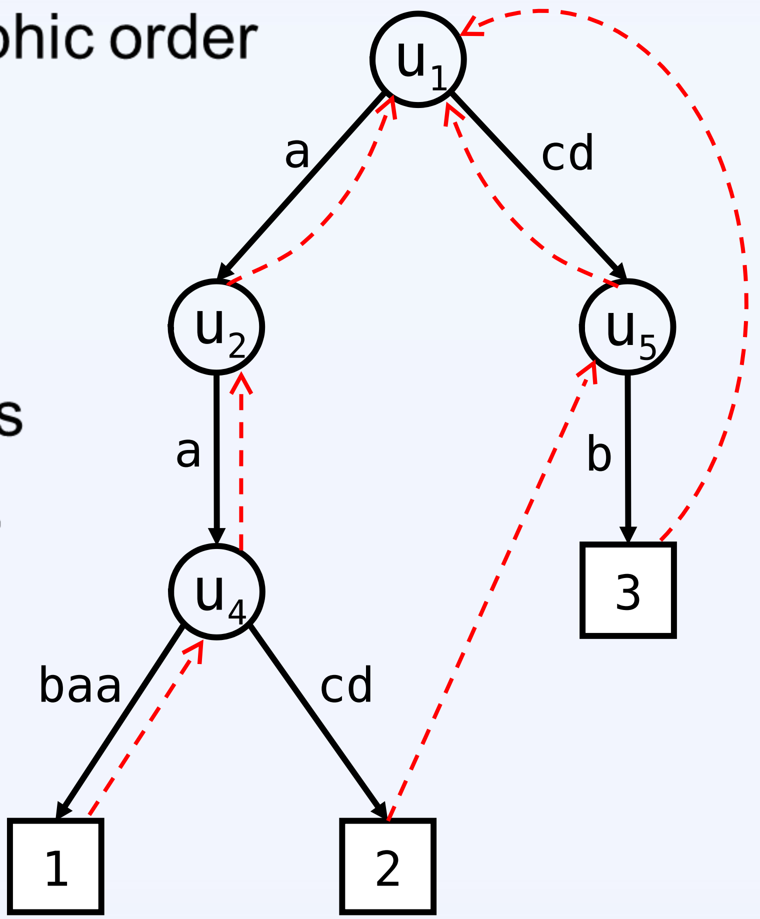
HOG

Main Algorithm

- Build an Aho-Corasick trie of P
- Renumber the strings in lexicographic order
- Build EHOG(P) in $O(||P||)$ time

- For each node u in EHOG(P), define an interval $I(u)$ that contains every leaf node in the subtree of u.

- Example: $I(u_1) = \{1, 2, 3\}$
  $I(u_2) = \{1, 2\}$
  $I(u_5) = \{3\}$



EHOG

Main Algorithm

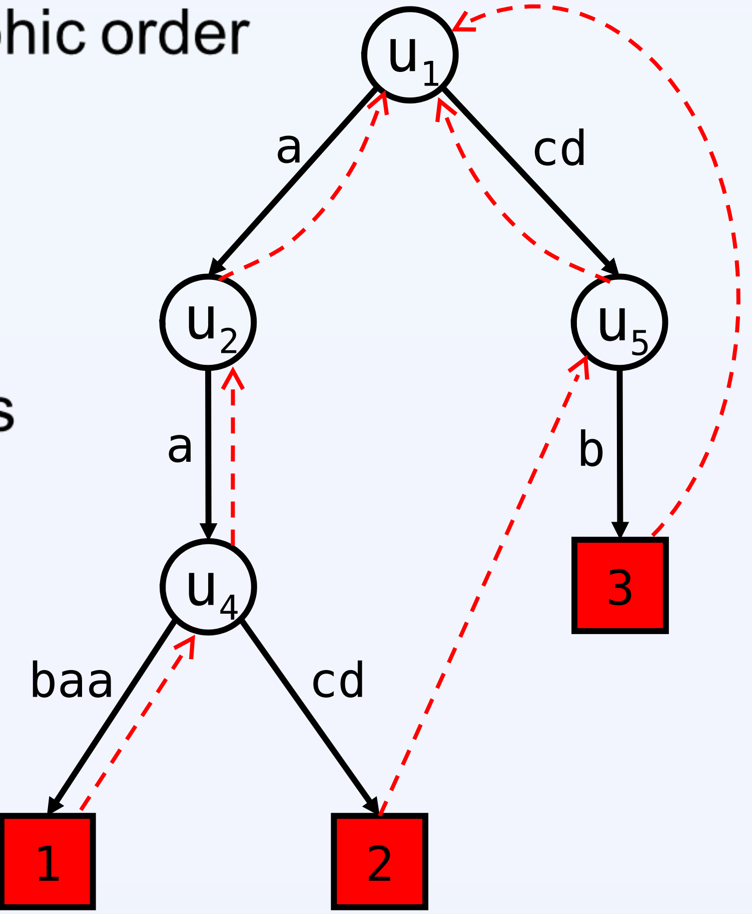
- Build an Aho-Corasick trie of P
- Renumber the strings in lexicographic order
- Build EHOG(P) in $O(||P||)$ time

- For each node u in EHOG(P), define an interval $I(u)$ that contains every leaf node in the subtree of u.

- Example: $I(u_1) = \{1, 2, 3\}$

  $I(u_2) = \{1, 2\}$

  $I(u_5) = \{3\}$

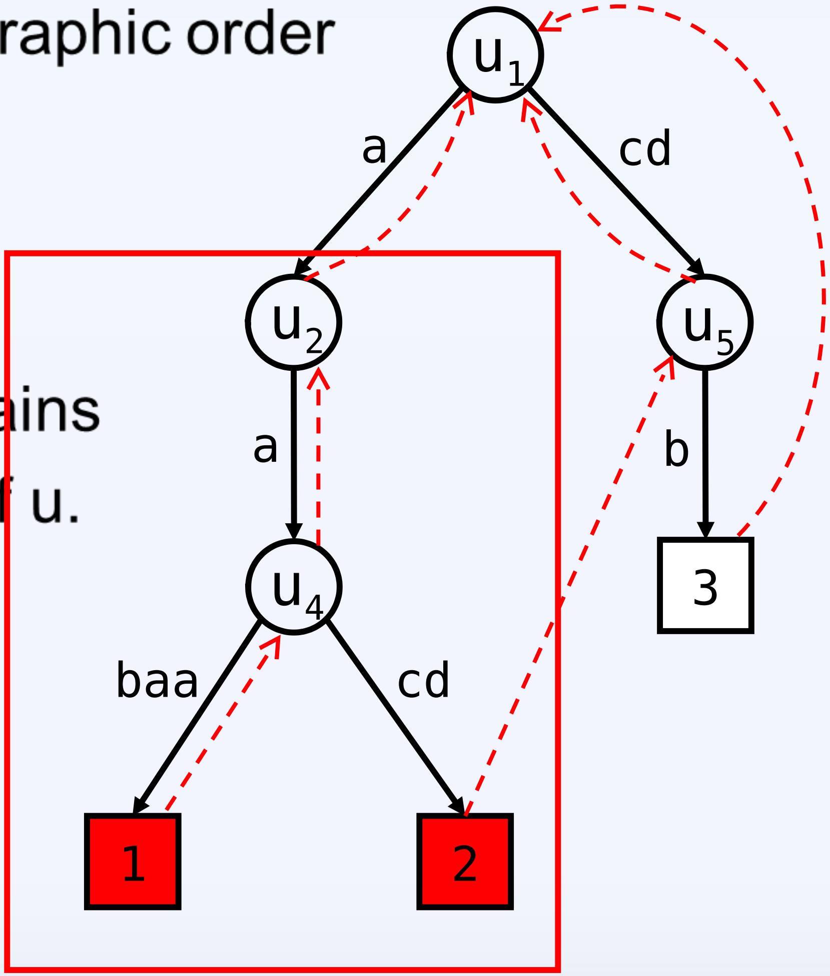


Main Algorithm

- Build an Aho-Corasick trie of P
- Renumber the strings in lexicographic order
- Build EHOG(P) in $O\left(\left|\left|P\right|\right|\right)$ time

- For each node u in EHOG(P), define an interval $I(u)$ that contains every leaf node in the subtree of u.

- Example: $I(u_1) = \{1, 2, 3\}$
  $I(u_2) = \{1, 2\}$
  $I(u_5) = \{3\}$



Main Algorithm

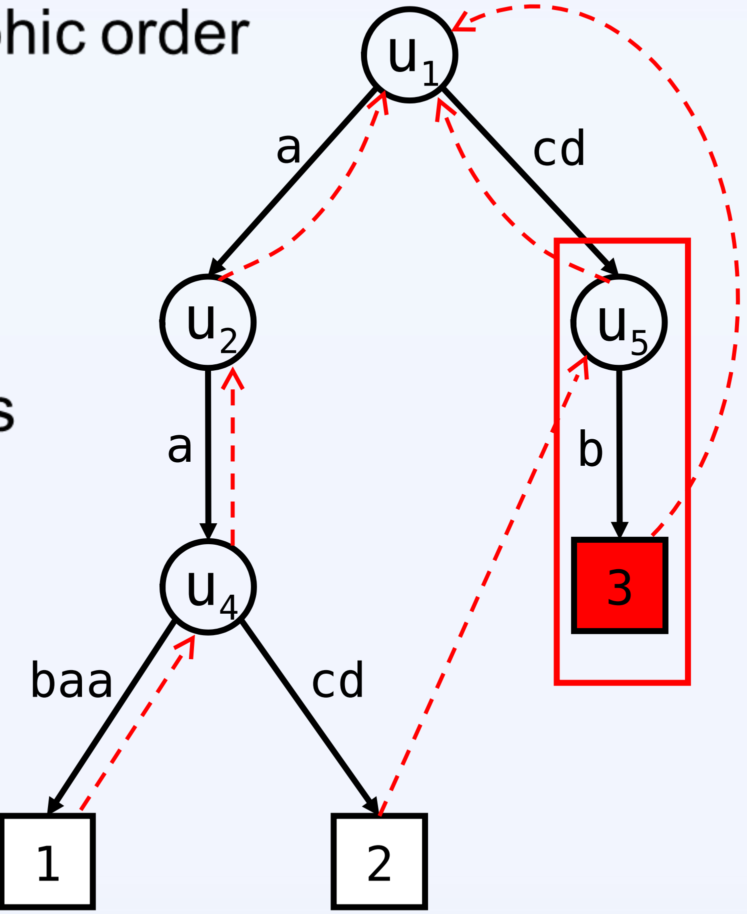
- Build an Aho-Corasick trie of P
- Renumber the strings in lexicographic order
- Build EHOG(P) in $O\left(\left|\left|P\right|\right|\right)$ time

- For each node u in EHOG(P), define an interval $I(u)$ that contains every leaf node in the subtree of u.

- Example: $I(u_1) = \{1, 2, 3\}$

  $I(u_2) = \{1, 2\}$

  $I(u_5) = \{3\}$



Main Algorithm

- We will compute $Ov(P)$ from $EHOG(P)$.

- What happens if $u \in Ov(P)$?
- For some $s_i, s_j \in P$, u is the longest overlap from s_i to s_j.
  - u is a proper suffix of s_i.
  - u is a proper prefix of s_j.
  - There are no longer overlaps from s_i to s_j than u.

Main Algorithm

- If we follow the failure link from s_i, we get every suffixes of s_i in a decreasing order of lengths.

- If we have a failure link chain $v_0 = s_i, v_1, v_2, \dots, v_k = root$, v_x is the longest overlap from s_i to s_j if v_x is the first node that is a prefix of s_j during the traversal.

- In other words, v_x is the longest overlap from s_i to s_j if:
  - v_x is a prefix of s_j
  - v_y is not a prefix of s_j for $1 \leq y < x$

Main Algorithm

- Maintain a bit vector B of length n.

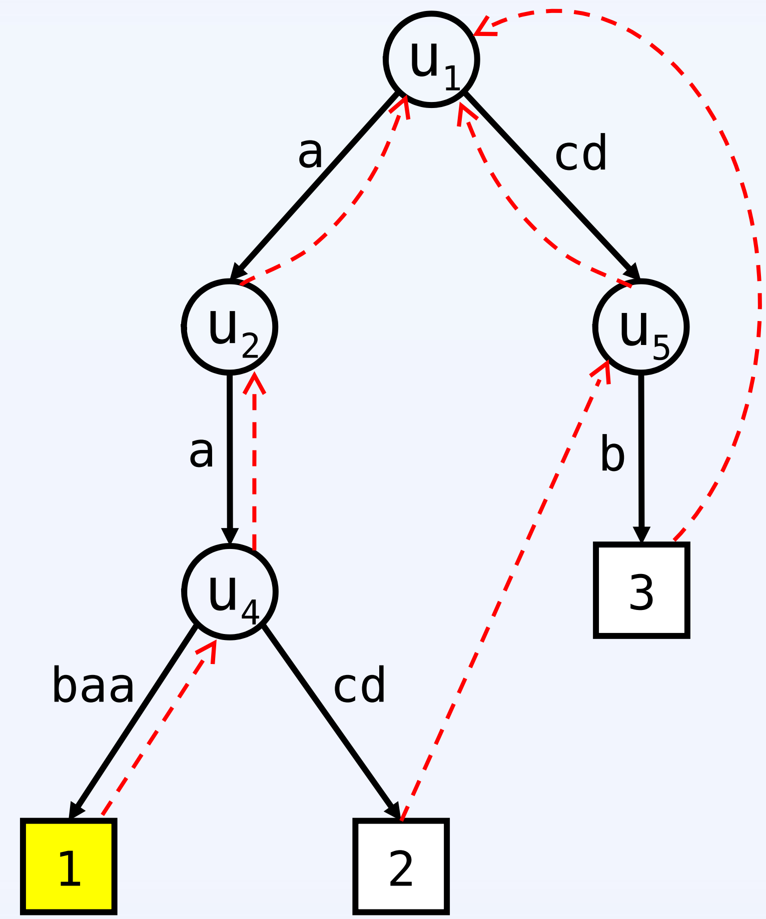
- At the end of iteration with v_x, $B[j] = true$ if and only if there exists $1 \leq y \leq x$ such that v_y is a prefix of s_j.

- We can check whether v_x should be included in $HOG(P)$ using B.

Main Algorithm

- We start with $i = 1, s_i = aabaa$.
- Initialize B with false.

- Follow the failure link repeatedly.



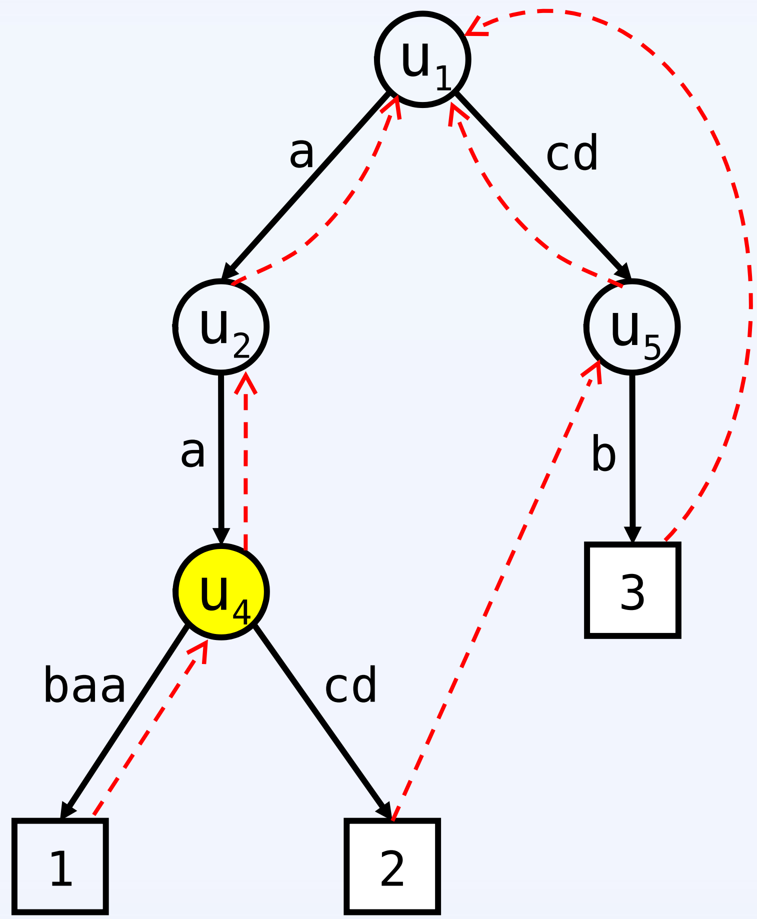
j	1	2	3
B[j]	false	false	false

Main Algorithm

- We start with $i = 1, s_i = aabaa$.
- Initialize B with false.

- Follow the failure link repeatedly.
- $v_1 = u_4$: $I(u_4) = \{1,2\}$

j	1	2	3
B[j]	false	false	false

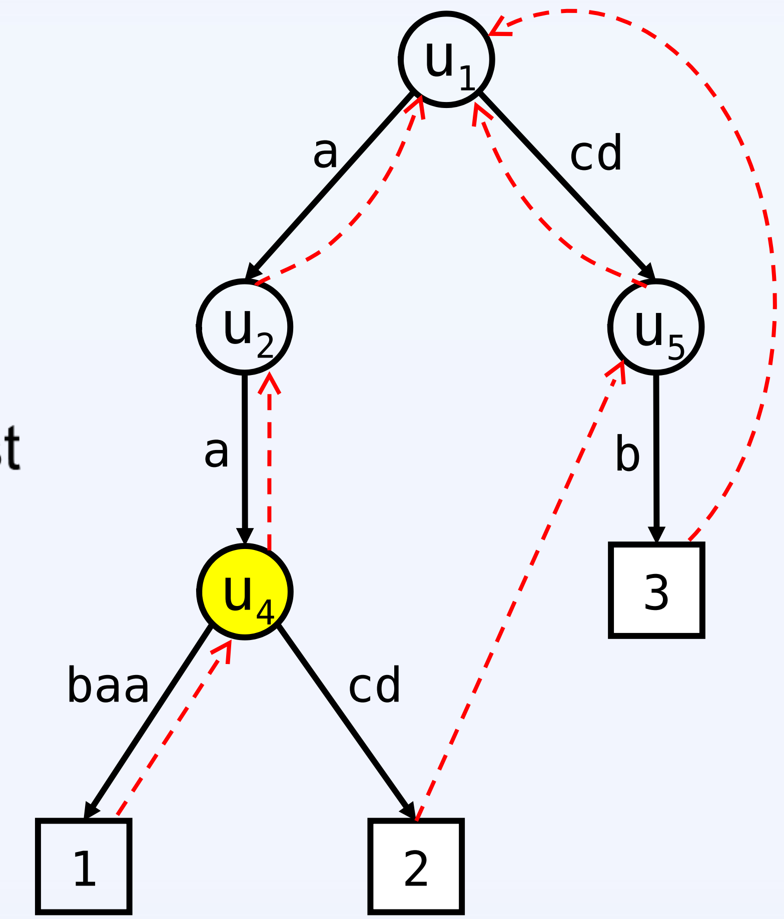


Main Algorithm

- We start with $i = 1, s_i = aabaa$.
- Initialize B with false.

- Follow the failure link repeatedly.
- $v_1 = u_4$: $I(u_4) = \{1,2\}$
- B[1] = B[2] = false: u_4 is the longest overlap from s_1 to s_1 and s_2.

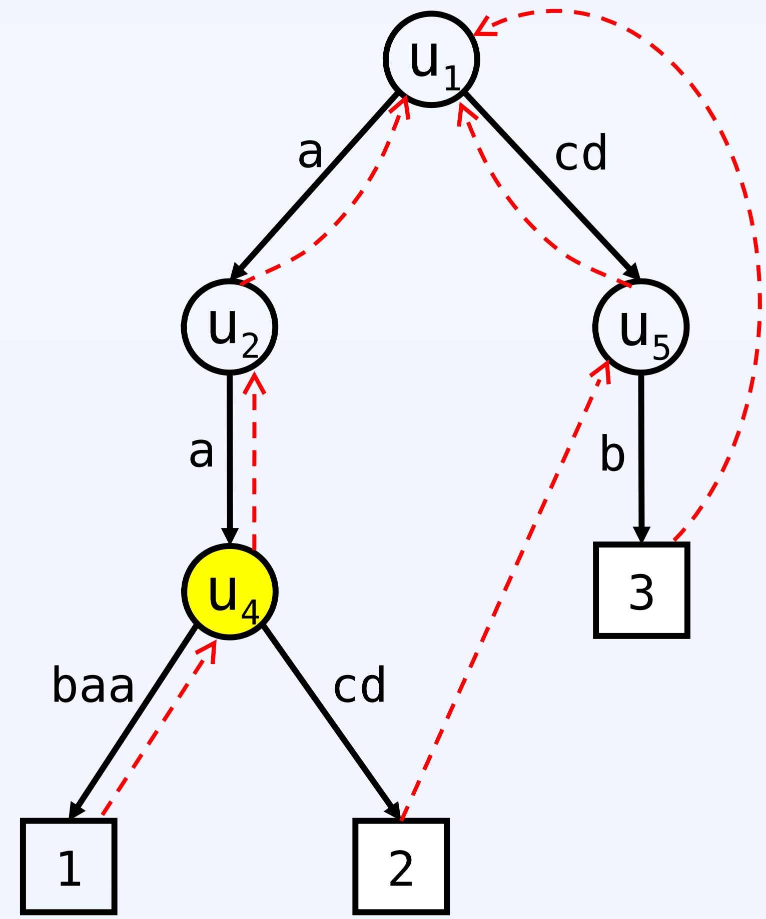
j	1	2	3
B[j]	false	false	false



Main Algorithm

- We start with $i = 1, s_i = aabaa$.
- Initialize B with false.
- Follow the failure link repeatedly.
- $v_1 = u_4$: $I(u_4) = \{1,2\}$
- Update B[1] and B[2] as true.

j	1	2	3
B[j]	true	true	false

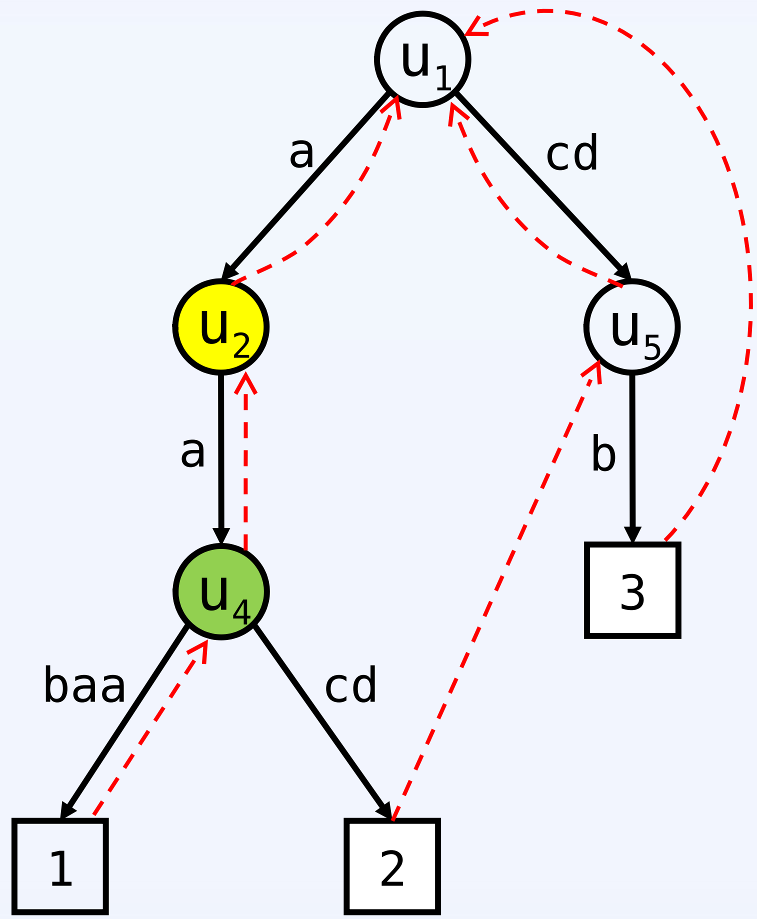


Main Algorithm

- We start with $i = 1, s_i = aabaa$.
- Initialize B with false.

- Follow the failure link repeatedly.
- $v_2 = u_2$: $I(u_2) = \{1,2\}$
- B[1] and B[2] are already true: u_2 is an overlap from s_1 to s_1 and s_2, but not the longest one.

j	1	2	3
B[j]	true	true	false

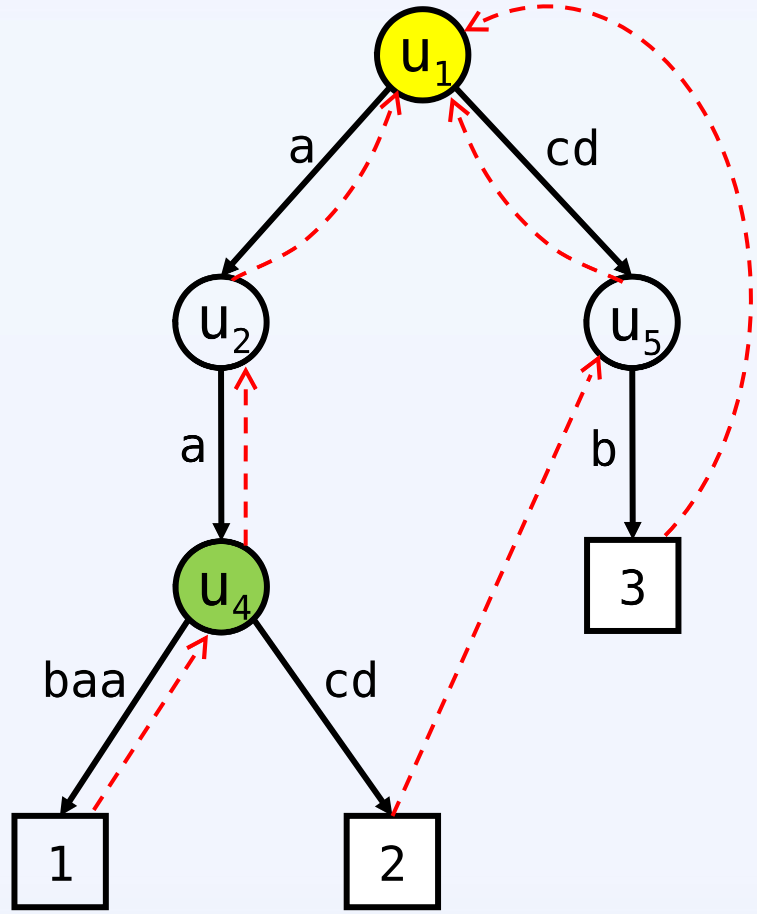


Main Algorithm

- We start with $i = 1, s_i = aabaa$.
- Initialize B with false.

- Follow the failure link repeatedly.
- $v_3 = u_1$: $I(u_1) = \{1,2,3\}$
- B[3] = false: u_1 is the longest overlap from s_1 to s_3.
- Update B[3] as true.

j	1	2	3
B[j]	true	true	true

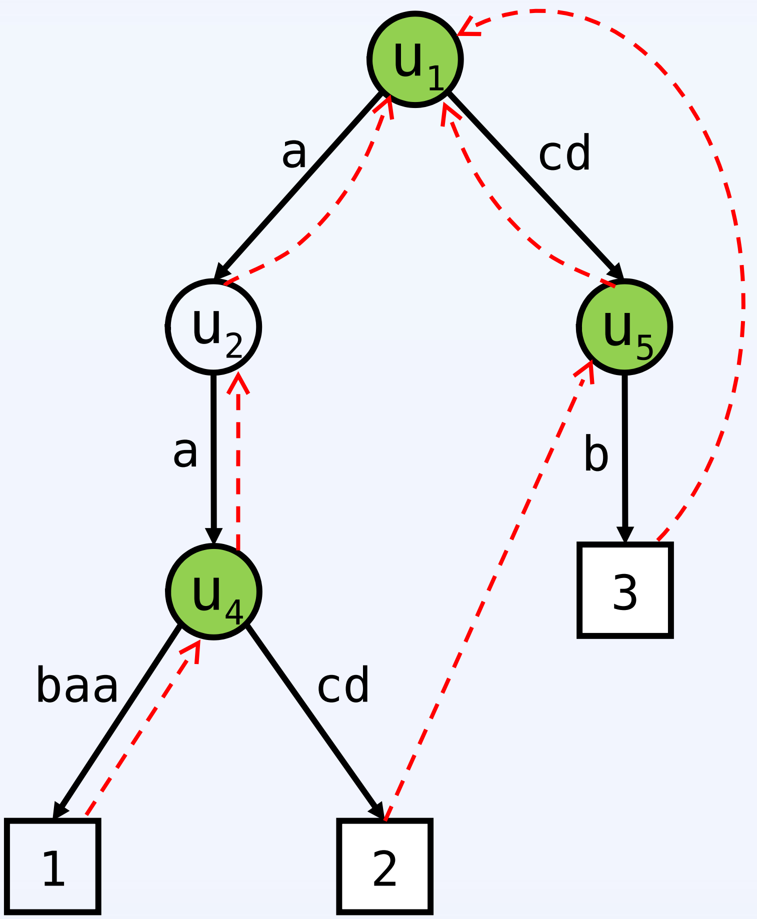


Main Algorithm

- We start with $i = 1, s_i = aabaa$.
- Initialize B with false.

- We do the same procedure starting with $s_2 = aacd$ and $s_3 = cdb$.

- $Ov(P) = \{u_1, u_4, u_5\}$



Main Algorithm (summary)

- For each s_i, do the following algorithm separately.

- Initialize B[1..n] to false
- Starting from node s_i, follow the failure links while doing the following works:

  i) If there exists $j \in I(u)$ such that B[j] = false, mark u to be included in $Ov(P)$.

  ii) For $j \in I(u)$, update B[j] as true.

- Build a HOG with marked nodes and EHOG.

Improvement using segment tree

- We need to process these two types of queries on B:

  i) If there exists $j \in I(u)$ such that B[j] = false, mark u to be included in $Ov(P)$.

  ii) For $j \in I(u)$, update B[j] as true.

- Consider an integer array A and following queries on A.

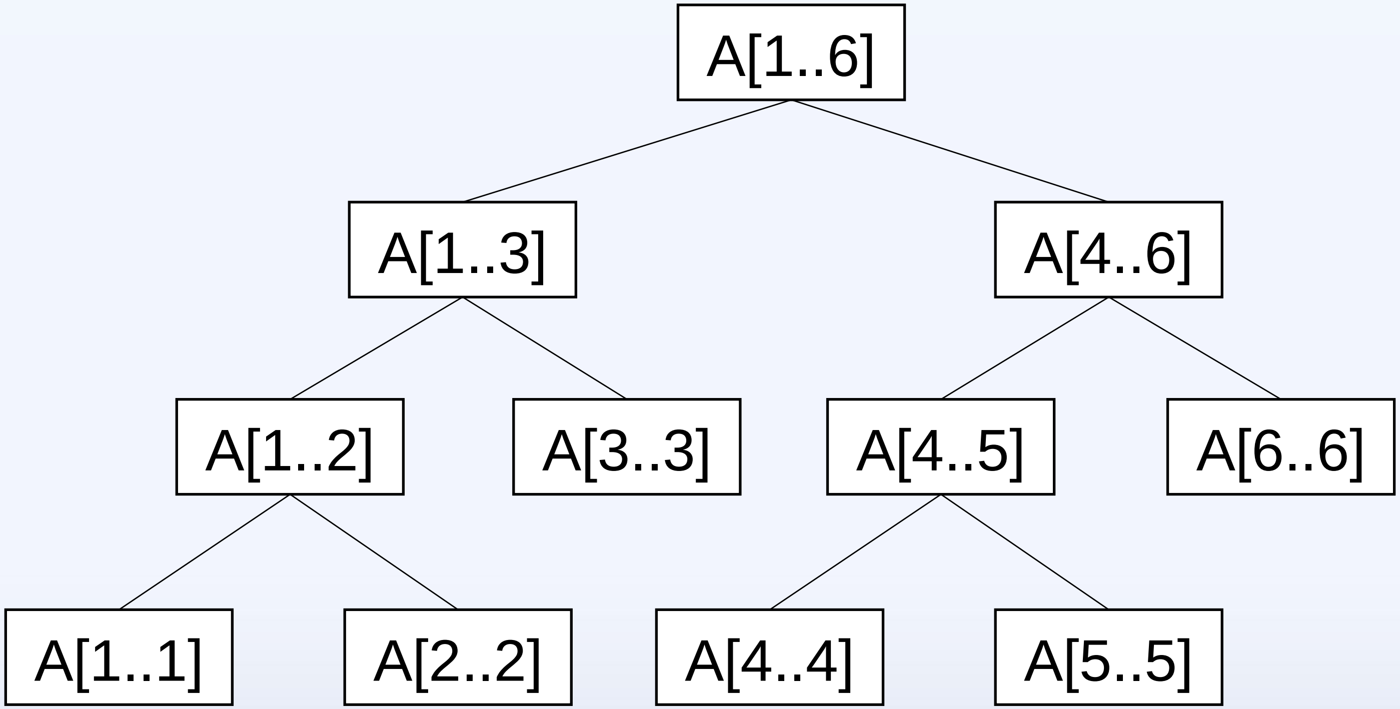
  i) Given an interval $[a..b]$, compute the minimum value among $A[a..b]$ (and check whether it is zero or not)

  ii) Given an interval $[a..b]$, add 1 to each element of $A[a..b]$.

- $A[j] = 0 \leftrightarrow B[j] = \text{false}, \text{A[j]} > 0 \leftrightarrow B[j] = \text{true}$

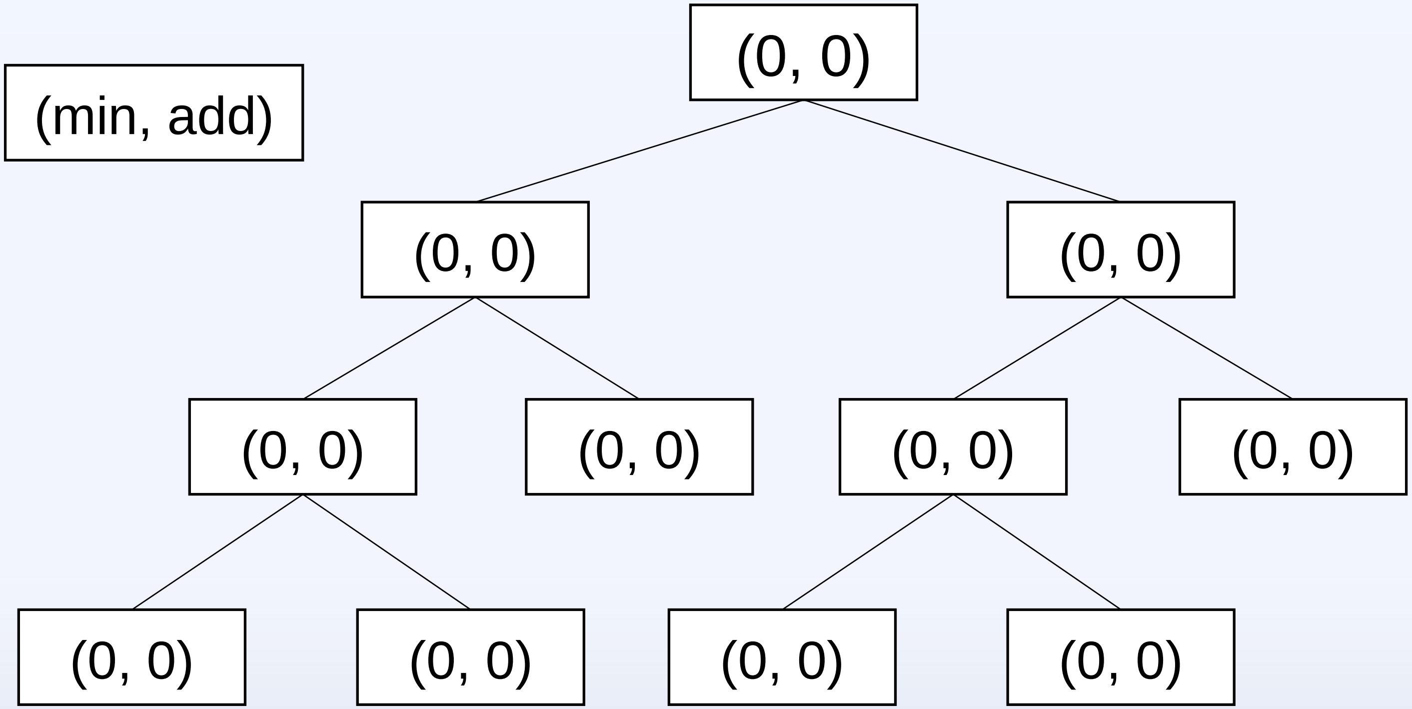
Improvement using segment tree

- We use segment tree to process queries on A.
- Segment tree: A binary tree which has n leaf nodes and has $O(\log n)$ height.
- Each internal node u corresponds to an interval $u.int$



Improvement using segment tree

- Each node stores two values, min and add.
  - $u.min$: Minimum value among the elements in $u.int$
  - $u.add$: Collectively added value to the elements in $u.int$



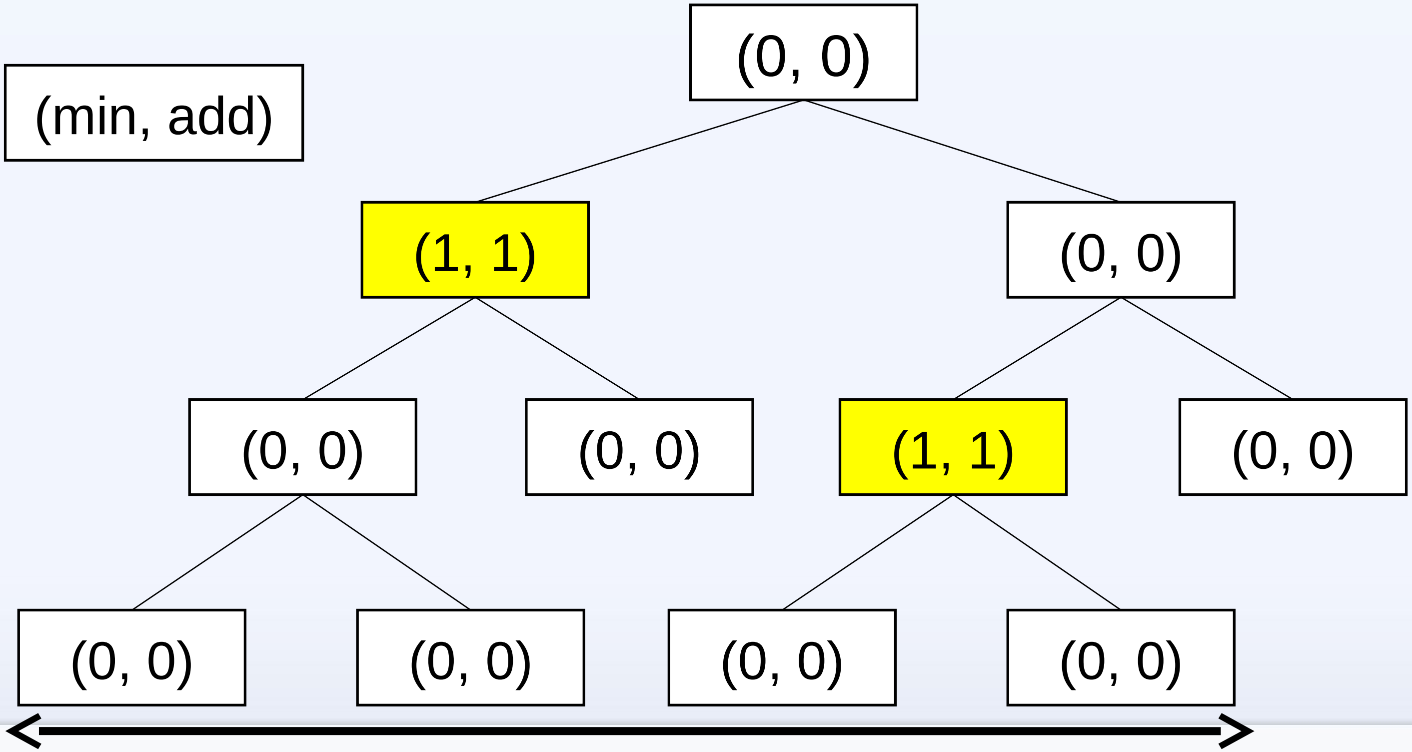
Improvement using segment tree

- We use lazy propagation technique.
- If query 1 occurs, we follow the nodes recursively from top to down, starting from the root.

- Considering a node u, we split the cases
- $u.int$ is included in the query interval: return $u.min$
- $u.int$ is disjoint with the query interval: return ∞
- Otherwise:
  - Propagate $u.add$ to child nodes
  - Continue with child nodes, and return minimum among them.

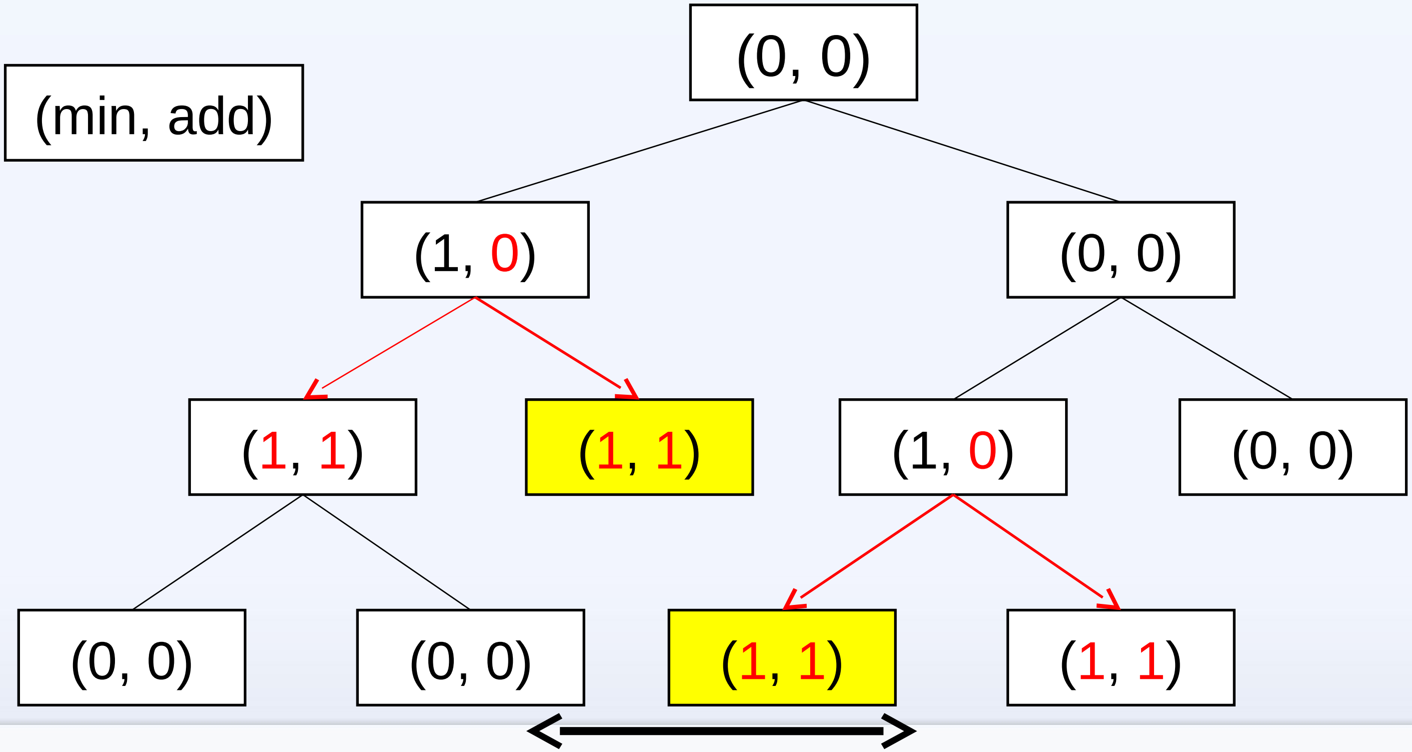
Improvement using segment tree

- Example: Query 2 on A[1..5]
  - Update two nodes, representing A[1..3] and A[4..5]
  - Their children are not updated yet



Improvement using segment tree

- Example: Query 1 on A[3..4]
  - We get min from two nodes, A[3..3] and A[4..4]
  - add value of parent nodes are propagated to their child nodes, ensuring the appropriate min values.



Improvement using segment tree

- Any interval $[a..b]$ can be represented by $O(\log n)$ nodes in the segment tree.

- Both queries can be done in $O(\log n)$ time.
- The total number of queries are $O(||P||)$.

- Time complexity: $O(||P|| \log n)$

- Both HOG and segment tree costs $O(||P||)$ space.
- Space complexity: $O(||P||)$

Conclusion

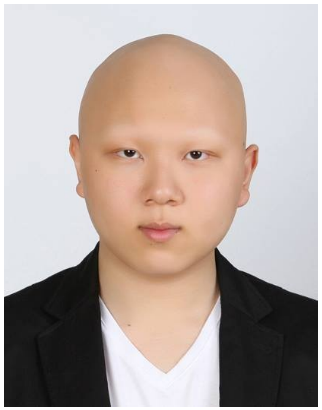
- We have presented a new algorithm to compute HOG in $O(\|P\| \log n)$ time and $O(\|P\|)$ space, using segment tree.

- We improved the time complexity of our algorithm to $O\left(\|P\| \frac{\log n}{\log \log n}\right)$, using word RAM model / w-segment tree.

Open questions :

- can one get a linear time algorithm?
- can one extend the data structure to encode approximate overlaps?

Thanks for your attention



Sung Gwan Park

Kunsoo Park

Bastien Cazaux

Eric Rivals

Funding:

IITP
Institute of Information & Communications Technology Planning & Evaluation

LabEx NUMEV

Paper: SPIRE, LNCS, 12303, doi: 10.1007/978-3-030-59212-7_20

Outline

- Introduction
- Preliminaries
- Main algorithm
- Improvement using segment tree
- Improvement using word RAM model

Improvement using word RAM

- Word RAM model: can read/write/do a bitwise operation for w-bit machine words in $O(1)$ time. $(w \geq \log n)$

- By using bitwise operations, we can improve the running time of two queries from $O(\log n)$ to $O\left(\frac{\log n}{\log \log n}\right)$.

- We use the w-segment tree, which is the w-ary version of the segment tree.

Improvement using word RAM

- Word RAM model: can read/write/do a bitwise operation for w-bit machine words in $O(1)$ time. ($w \geq \log n$)

- By using bitwise operations, we can improve the running time of two queries from $O(\log n)$ to $O\left(\frac{\log n}{\log \log n}\right)$.

- We use the w-segment tree, which is the w-ary version of the segment tree.

Improvement using word RAM

- Instead of $u.min$ and $u.add$, we store two bit vectors of length w, $u.Vmin$ and $u.Vadd$.
- If a node u is the j-th child of its parent p, $p.Vmin[j]$ is true if and only if $u.min = 0$.
- We can update the w-segment tree similarly to the (binary) segment tree, but in $O(\log_w n) = O\left(\frac{\log n}{\log\log n}\right)$ time.
- We can simulate w-segment tree using arrays, which results in using $O(n)$ bits in total.

Pointers

Lazy Propagation in Segment Tree

https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/