2024/7/24 文字列ゼミ

CDAWGによる LZ78部分文字列圧縮

柴田 紘希（九州大学）

本研究の概要

研究でやったこと

CDAWGを用いて、LZ78分解の部分文字列圧縮を

省領域で高速に行う手法を提案した

- CDAWG: 文字列の省領域索引
- LZ78分解: 文字列の圧縮表現のひとつ
- 部分文字列圧縮:事前にテキスト文字列が与えられ、

  部分文字列の圧縮結果を返すクエリを高速に処理する問題

準備: LZ78分解と部分文字列圧縮問題

LZ78分解 [Lempel & Ziv, '78]

- 文字列の圧縮手法の一つ
- 文字列 T を特定のアルゴリズムにしたがって

  $T = F_0F_1 \ldots F_f$ に分解する（ただし、F_0 は空文字列）

$$T = \mathrm{abbabaaab} = F_0F_1 \ldots F_f$$

$$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{ba}, \mathrm{baa}, \mathrm{ab})$$

LZ78分解 [Lempel & Ziv, '78]

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  - $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ とし、T' の接頭辞となる最長の F_j $(j < i)$ を求める
  - $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ とする

$$T = \text{abbabaaaab} = F_0 F_1 \ldots F_f \ (F_0 = \varepsilon)$$

$$T' = \text{abbabaaaab}$$
$$F = (\varepsilon)$$

LZ78分解 [Lempel & Ziv, '78]

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  - $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ とし、T' の接頭辞となる最長の F_j $(j < i)$ を求める
  - $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ とする

$$T' = \mathrm{abbabaaab}$$
$$F = (\varepsilon, \mathrm{a})$$

LZ78分解 [Lempel & Ziv, '78]

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  - $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ とし、T' の接頭辞となる最長の F_j $(j < i)$ を求める
  - $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ とする

$$T' = \mathrm{abbabaaab}$$
$$F = (\varepsilon, \mathrm{a}, \mathrm{b})$$

LZ78分解 [Lempel & Ziv, '78]

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  - $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ とし、T' の接頭辞となる最長の F_j $(j < i)$ を求める
  - $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ とする

$$T' = \mathrm{abbabaaab}$$
$$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{ba})$$

LZ78分解 [Lempel & Ziv, '78]

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  - $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ とし、T' の接頭辞となる最長の F_j $(j < i)$ を求める
  - $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ とする

$$T' = \text{abbabaaaab}$$
$$F = (\varepsilon, \text{a}, \text{b}, \text{ba}, \text{baa})$$

LZ78分解 [Lempel & Ziv, '78]

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  - $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ とし、T' の接頭辞となる最長の F_j $(j < i)$ を求める
  - $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ とする

$$T' = \text{abbabaaab}$$
$$F = (\varepsilon, \text{a}, \text{b}, \text{ba}, \text{baa}, \text{ab})$$

LZ78分解 [Lempel & Ziv, '78]

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  - $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ とし、T' の接頭辞となる最長の F_j $(j < i)$ を求める
  - $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ とする

$$T' = \mathrm{abbabaaab}$$
$$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{ba}, \mathrm{baa}, \mathrm{ab})$$

※もし $T' = F_j$ となったら $F_i = F_j$ とする

LZ78分解 [Lempel & Ziv, '78]

- Factor F_i は整数 j_i と文字c_iを用いて (j_i, c_i) と表現できる
  - (j_i, c_i) の列 F' から元の文字列 T を復元できるため、この列が圧縮表現になる！

$$
\begin{aligned}
T &= \mathrm{abbabaaab} \\
F &= (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{ba}, \mathrm{baa}, \mathrm{ab}) \\
F' &= ((0, \mathrm{a}), (0, \mathrm{b}), (2, \mathrm{a}), (3, \mathrm{a}), (\mathrm{a}, \mathrm{b}))
\end{aligned}
$$

部分文字列圧縮問題

- 長さ n のテキスト T が事前に与えられる
  - T についての索引構造を事前に構築しておいてよい
- 以下のクエリを処理:
  - 入力: 整数 l, r $(1 \leq l \leq r \leq n)$
  - 出力: $T[l..r]$ の圧縮表現

$T = \mathrm{abbabaaaab}, (l, r) = (2, 7)$

→ $T[l..r]$ のLZ78分解は $(\mathrm{b}, \mathrm{ba}, \mathrm{baa})$

LZ78分解の部分文字列圧縮

LZ78分解の部分文字列圧縮 [Köppl, '21]

LZ78分解の部分文字列圧縮は、1クエリあたり

$$O(z_{l,r}) \text{ time} \cdot O(n) \text{ space}$$

で解くことができる

※ $n = |T|$ で、 $z_{l,r}$ は $T[l..r]$ のLZ78分解のfactor数

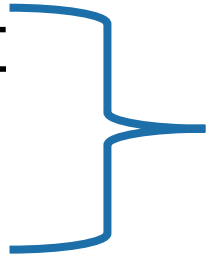
- 接尾辞木（Suffix Tree）を用いた手法

本研究はこの手法の省領域化

※ワードサイズ $\Omega(n)$ のword-RAM modelを仮定、空間計算量はワード数で表記

発表の流れ

1. 接尾辞木によるLZ78分解の計算
2. 1. の部分文字列圧縮版
3. CDAWGによるLZ78分解の計算
4. 3. の部分文字列圧縮版



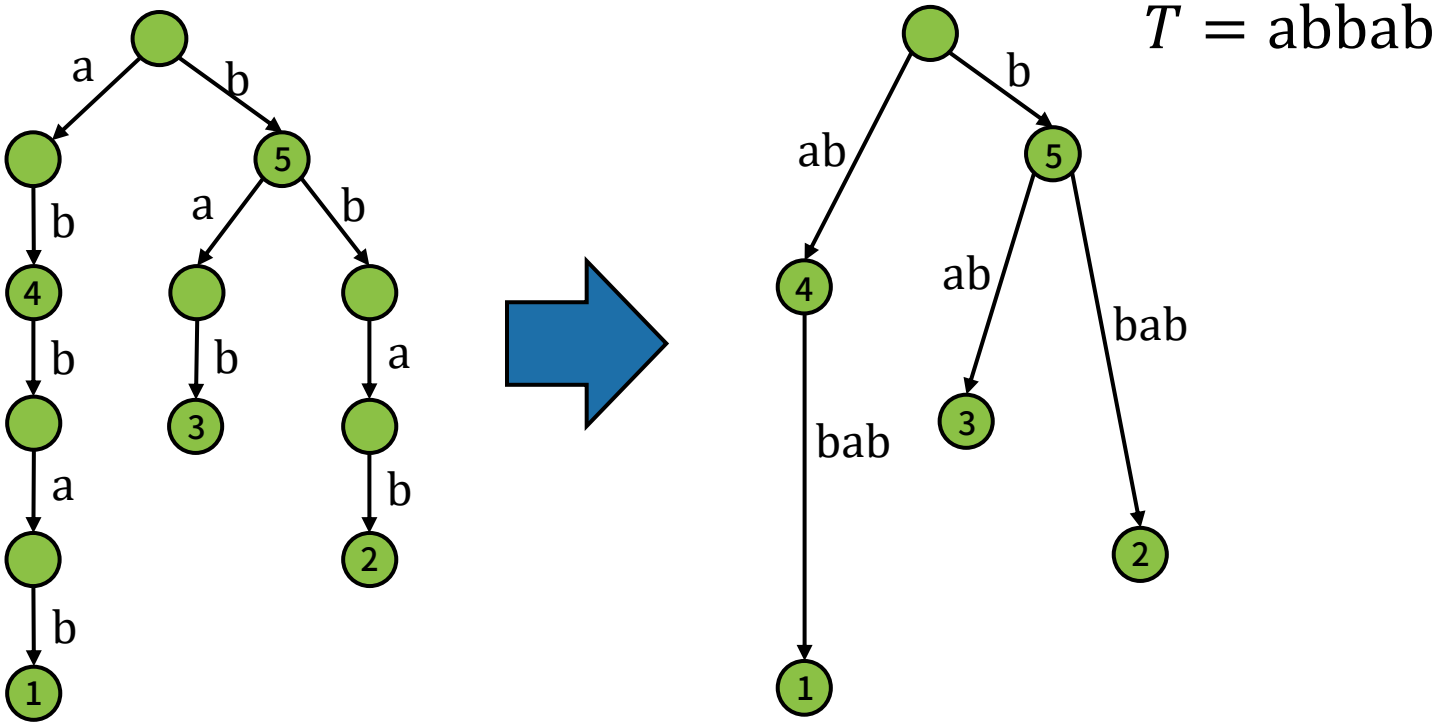
本研究の貢献

既存手法: 接尾辞木を使ったLZ78分解

+ 部分文字列圧縮

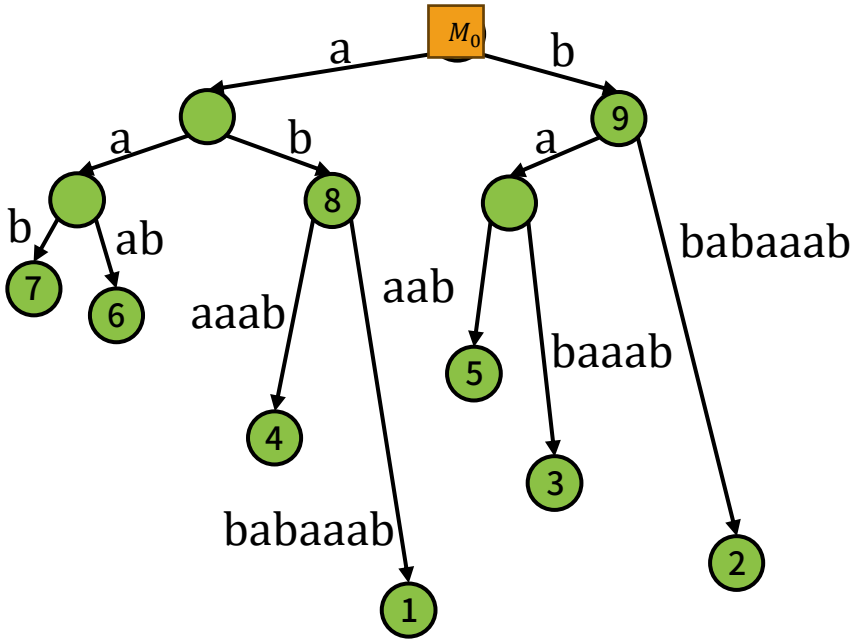
接尾辞木 (Suffix Tree)

文字列の接尾辞集合を表すトライ木から、接尾辞を表さない出次数1のノードをひとまとめにすることで得られる辺ラベル付き木



接尾辞木によるLZ78分解の計算

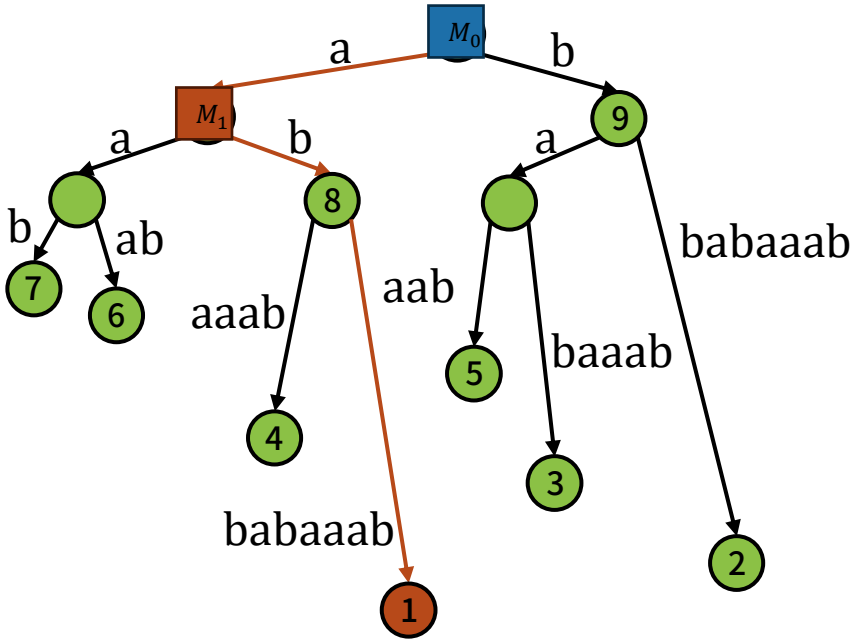
- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  1. $T' = T[1 + |F_0| + |F_1| + \cdots, |F_{i-1}|..n]$ を表す接尾辞木上のパスを見つける
  2. パス上に存在する中で最深のマーク M_j を探す
  3. $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ として、接尾辞木中の F_i に対応する地点にマーク M_i をつける



$$T' = \text{abbabaaaab}$$
$$F = (\varepsilon)$$

接尾辞木によるLZ78分解の計算

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  1. $T' = T[1 + |F_0| + |F_1| + \cdots, |F_{i-1}|..n]$ を表す接尾辞木上のパスを見つける
  2. パス上に存在する中で最深のマーク M_j を探す
  3. $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ として、接尾辞木中の F_i に対応する地点にマーク M_i をつける

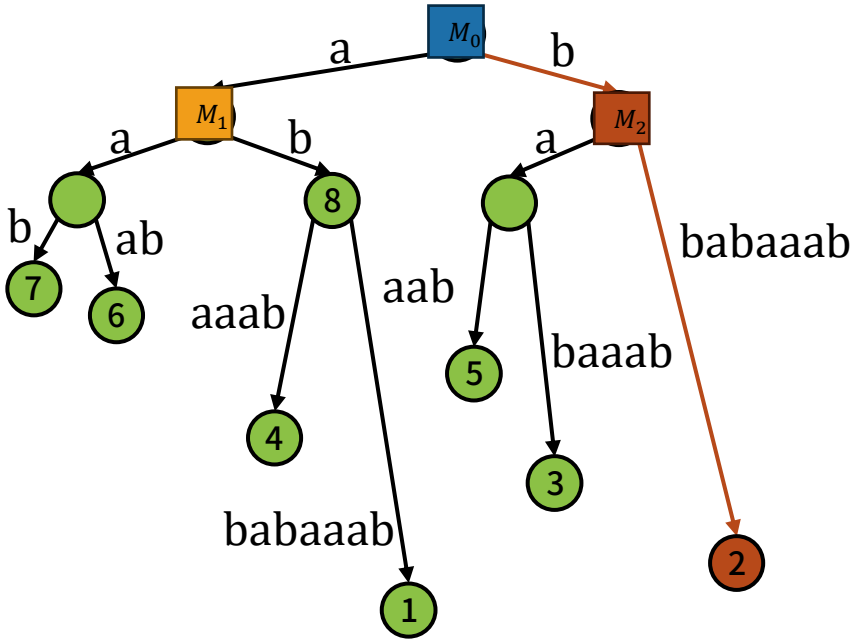


$T' = \mathrm{abbabaaab}$

$F = (\varepsilon, \mathrm{a})$

接尾辞木によるLZ78分解の計算

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  1. $T' = T[1 + |F_0| + |F_1| + \cdots, |F_{i-1}|..n]$ を表す接尾辞木上のパスを見つける
  2. パス上に存在する中で最深のマーク M_j を探す
  3. $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ として、接尾辞木中の F_i に対応する地点にマーク M_i をつける

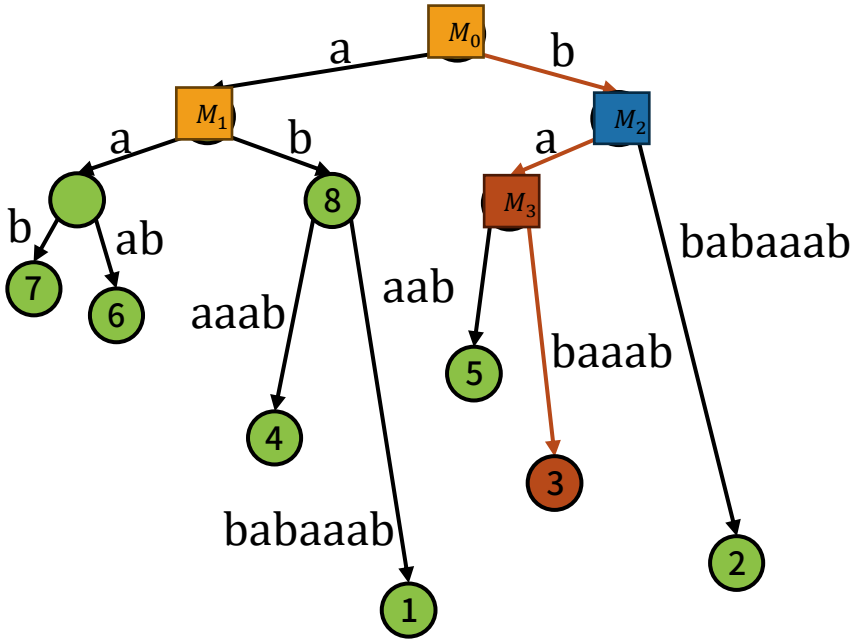


$T' = \mathrm{abbabaaab}$

$F = (\varepsilon, \mathrm{a}, \mathrm{b})$

接尾辞木によるLZ78分解の計算

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  1. $T' = T[1 + |F_0| + |F_1| + \cdots, |F_{i-1}|..n]$ を表す接尾辞木上のパスを見つける
  2. パス上に存在する中で最深のマーク M_j を探す
  3. $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ として、接尾辞木中の F_i に対応する地点にマーク M_i をつける

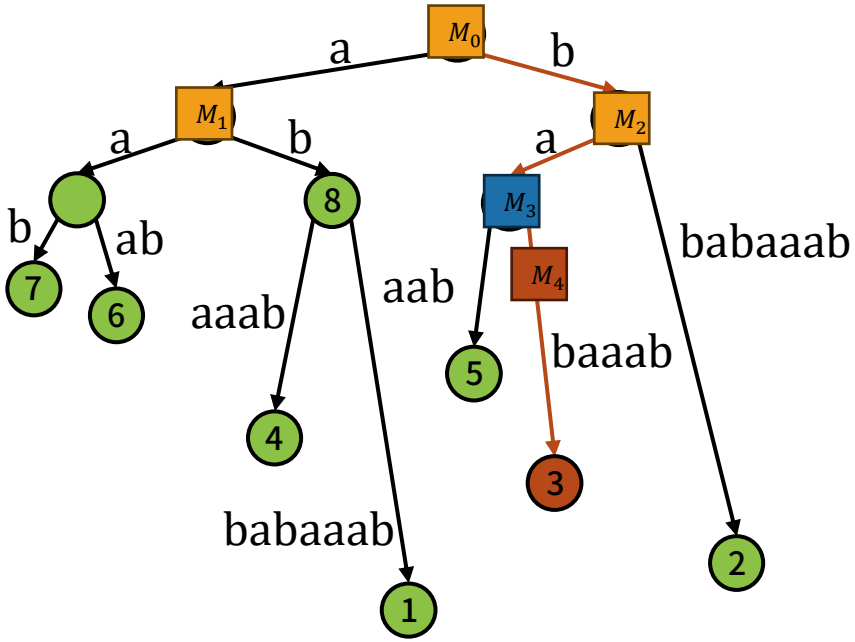


$T' = \mathrm{abbabaaaab}$

$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{ba})$

接尾辞木によるLZ78分解の計算

- F_i $(i \geq 1)$ を計算するアルゴリズムは以下の通り:
  1. $T' = T[1 + |F_0| + |F_1| + \cdots, |F_{i-1}|..n]$ を表す接尾辞木上のパスを見つける
  2. パス上に存在する中で最深のマーク M_j を探す
  3. $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ として、接尾辞木中の F_i に対応する地点にマーク M_i をつける

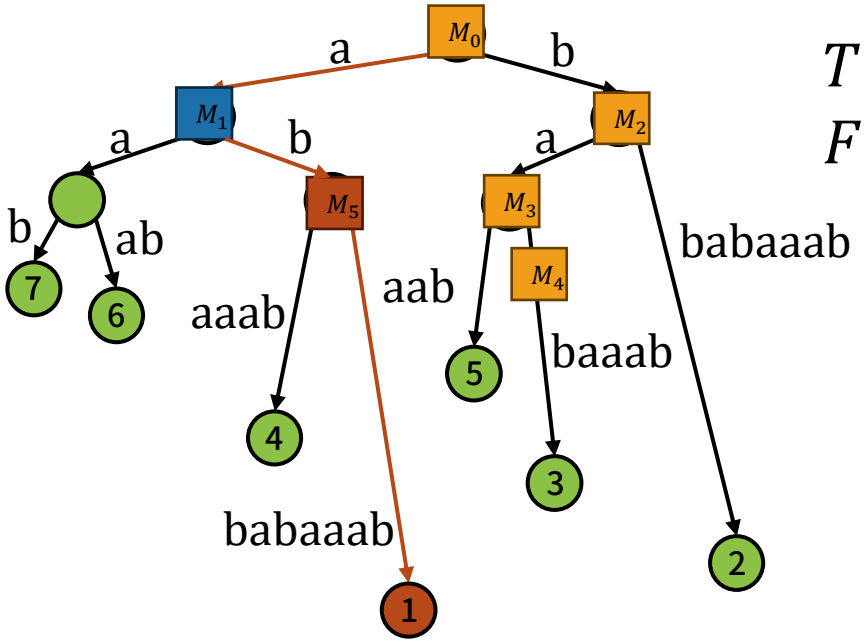


$T' =$ abbabaaaab

$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{ba}, \mathrm{baa})$

接尾辞木によるLZ78分解の計算

- $F_i\ (i \geq 1)$ を計算するアルゴリズムは以下の通り:
  1. $T' = T[1 + |F_0| + |F_1| + \cdots, |F_{i-1}|..n]$ を表す接尾辞木上のパスを見つける
  2. パス上に存在する中で最深のマーク M_j を探す
  3. $F_i = F_j T'\left[\left|F_j\right| + 1\right]$ として、接尾辞木中の F_i に対応する地点にマーク M_i をつける



$T' =$ abbabaaab

$F = (\varepsilon, a, b, ba, baa, ab)$

接尾辞木によるLZ78分解の計算

アルゴリズム中で用いるテクニック

- T' を表すパス上の最深マークを見つける・マークをつける
  - Lowest Marked Ancestor Problemであり、O(1) 時間で処理 できる [Westbrook, 1992]
- F_i に対応する接尾辞木中の位置を得る
  - 接尾辞木上のWeighted Level Ancestor Problemであり、O(1) 時間で求まる [Gawrychowski et al., 2014, Bellazougui et al., 2021]

時間・空間計算量

前処理も $O(n)$ 時間

- 接尾辞木・LMA索引・WLA索引は全て $O(n)$ spaceなため、全体でも $O(n)$ space
- マーク地点の取得・マークが定数時間で行えるため、factorあたり $O(1)$ 時間

⇒ 全体の計算量は 空間 $O(n)$ ・時間 $O(z)$

※ z は T のLZ分解のfactor数

接尾辞木によるLZ78分解の部分文字列圧縮

部分文字列圧縮への対応: とても簡単

- 始点 l の対応: 前述のアルゴリズムの T' を

  $T' = T[\, l + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ に書き換えれば良い

- 終点 r の対応: 通常通り計算して、$T[l..r]$ の範囲をはみ出したら末尾のfactorを削れば良い

計算量: $O(z_{l,r})$ time・$O(n + z_{l,r})$ space

※索引自体に $O(n)$ space必要で、作業領域が $O(z_{l,r})$ space

※ $z_{l,r}$ は $T[l..r]$ のLZ分解のfactor数

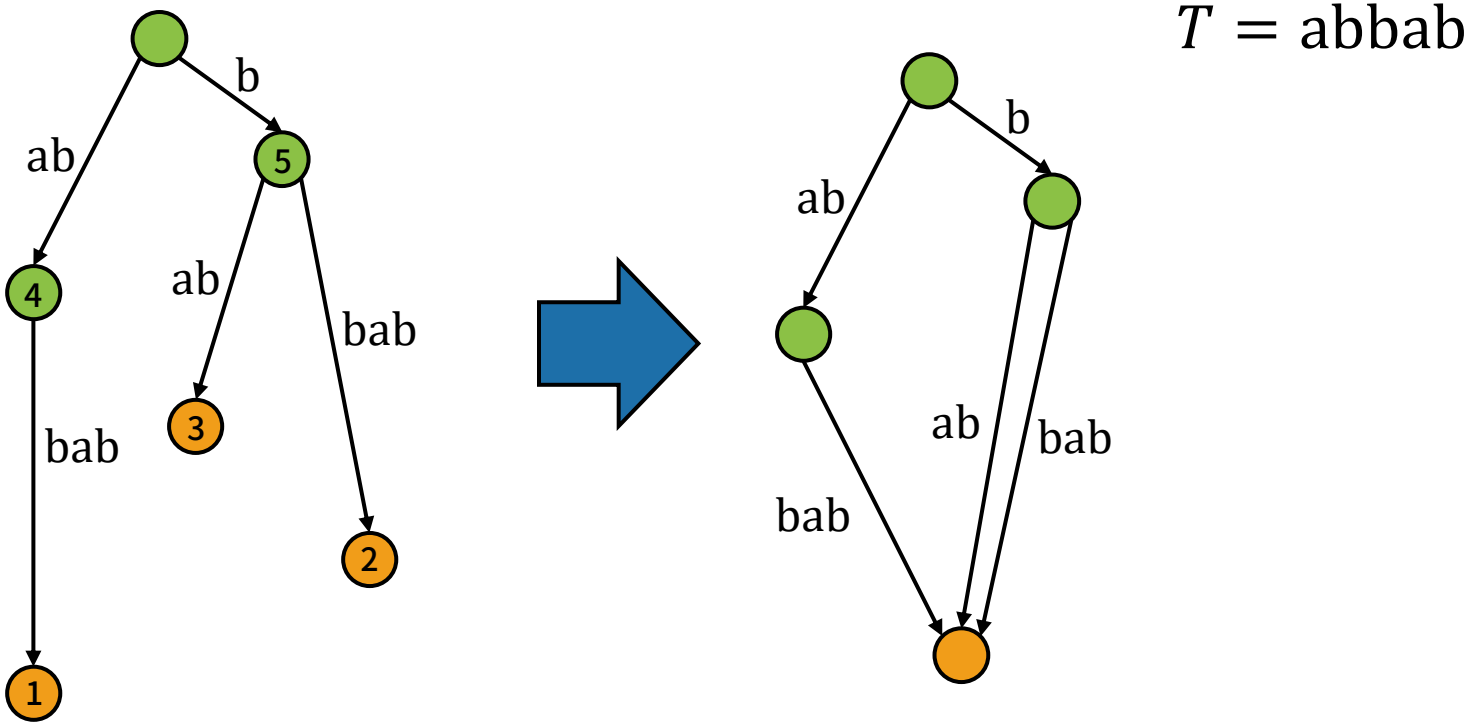
提案手法: CDAWGを使ったLZ78分解

+ 部分文字列圧縮

CDAWG (Compacted Directed Acyclic Word Graph)

接尾辞木の同型な部分木をひとまとめにすることで得られる

辺ラベル付きDAG

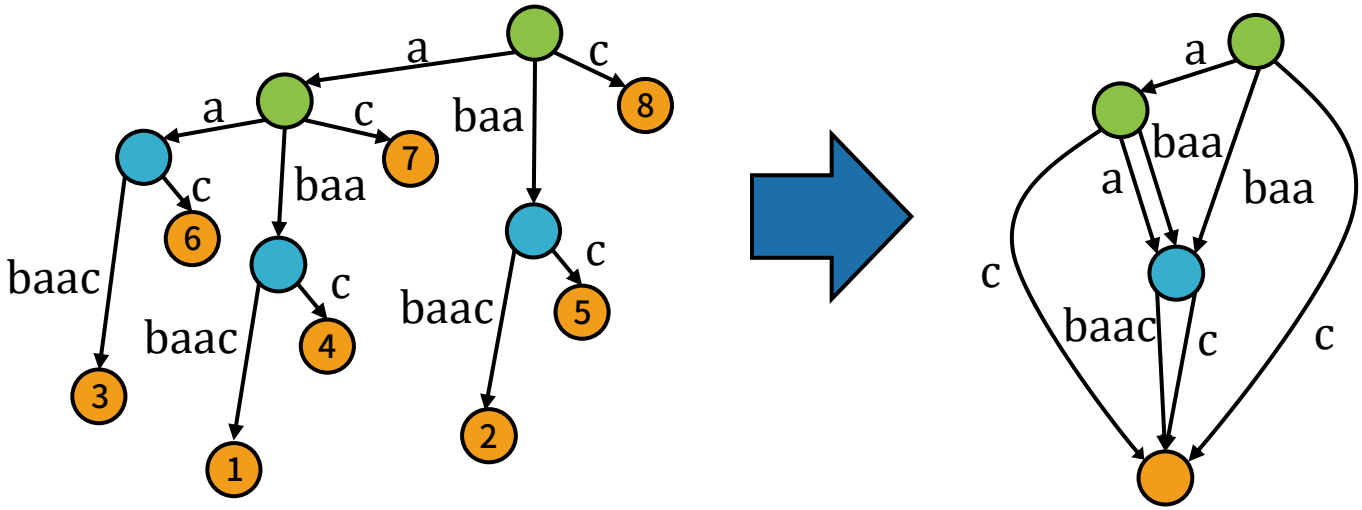


CDAWG (Compacted Directed Acyclic Word Graph)

接尾辞木の同型な部分木をひとまとめにすることで得られる

辺ラベル付きDAG

$T = \mathrm{abaabaac}$

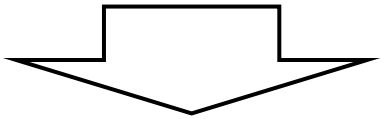


CDAWGによる文字列圧縮

- e : CDAWGの辺の数
  - 接尾辞木の辺数は $2n-1$ 以下なので、 $e \leq 2n-1$

性質: 文字列が繰り返し構造を持つと e は小さくなり、

$e \in \Theta(\log n)$ であるような文字列も存在



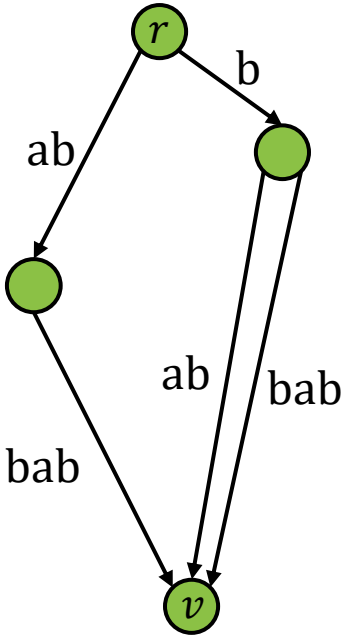
$O(e)$ spaceで索引を作ることができれば、元文字列より省領域な圧縮索引になる！

接尾辞木→CDAWG で置き換える際の課題

先行研究のアルゴリズムをCDAWGでシミュレートしたい → 課題あり

1. CDAWGでは**祖先が一意に定まらない**
2. CDAWGでは**1つの頂点が複数文字列を表す**

⇒ 別のアプローチを用いて解決する



$T = \mathrm{abbab}$

$S(v) = \{\mathrm{abbab}, \mathrm{bbab}, \mathrm{bab}\}$

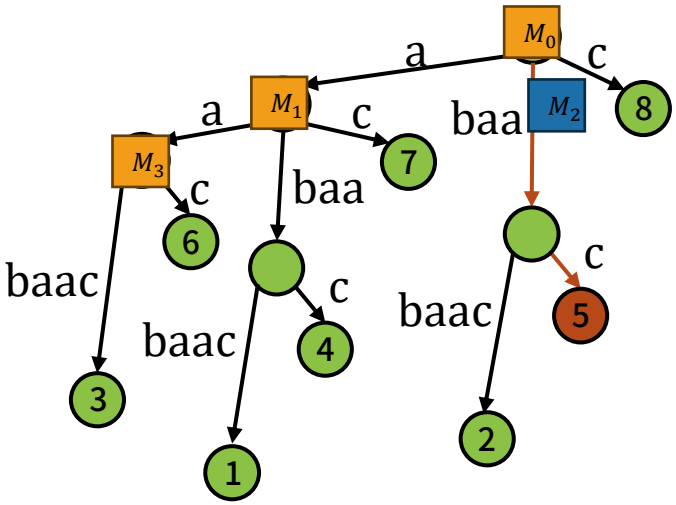
v の入次数は3で、祖先は2頂点ある

※ $S(v)$: CDAWGのroot r → 頂点 v のパスが表す文字列の集合

Lowest Marked Ancestorの帰着

先行研究のアルゴリズム中で扱うMarked Ancestor は以下のようなもの:

1. 頂点 v にマークをつける
2. 葉 ℓ を選び、ℓ から根へのパス上で最も深いマークを見つける



※ 辺 (u, v) へのマークは頂点 v へのマークとしてよい

※ 一般には接尾辞を表す頂点が葉とは限らないが、
T の末尾にuniqueな文字を追加することで
葉であることを保証できる

Lowest Marked Ancestorの帰着

アルゴリズム中で扱うMarked Ancestor Problemは以下のようなもの:

1. 頂点 v にマークをつける
2. 葉 ℓ を選び、ℓ から根へのパス上で最も深いマークを見つける

辺の探索順序は辞書順にしておく

木の葉を行きがけ順に並べて $\ell_1, \ell_2, \ldots, \ell_n$ とすることで、上の2つの処理は以下の2つの処理に置き換えられる:

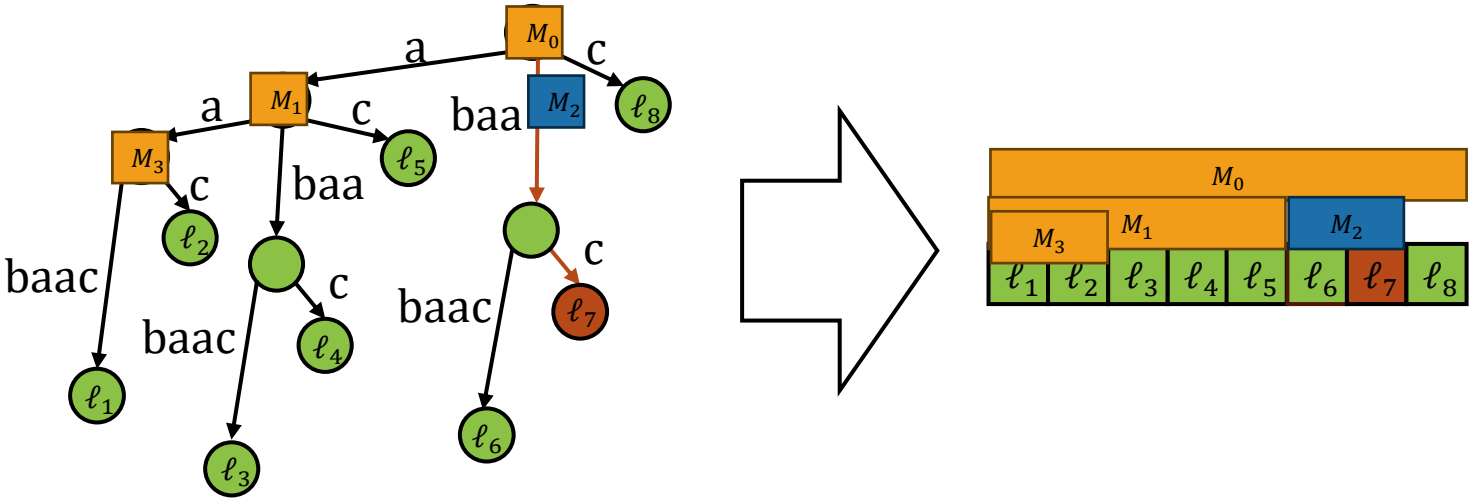
1. v の子孫である葉 $\ell_a, \ell_{a+1}, \ldots, \ell_b$ に重み $\mathrm{depth}(v)$ でマークを付ける
2. ℓ_k についたマークのうち、重みが最大のものを求める

※$\mathrm{depth}(v)$: 頂点 v の深さ

Lowest Marked Ancestorの帰着

木の葉を行きがけ順（辺順序は辞書順）に並べて $\ell_1, \ell_2, \ldots, \ell_n$ とする

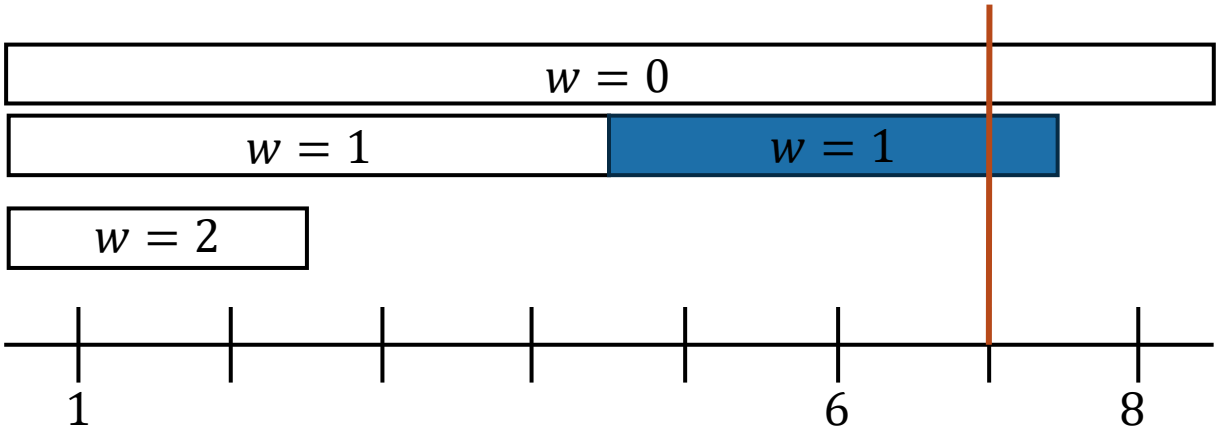
1. v の子孫である葉 $\ell_a, \ell_{a+1}, \ldots, \ell_b$ に重み $\mathrm{depth}(v)$ でマークを付ける
2. ℓ_k についたマークのうち、重みが最大のものを求める



Lowest Marked Ancestorの帰着

問題をさらに抽象化すると、以下の問題に帰着できる:

1. 重み w の区間 $[a, b]$ を集合に追加
2. 整数 k を包含するような区間のうち、重みが最大のものを求める



※頂点に対応する $[a, b]$ や葉に対応する k は高速に求まると仮定

Lowest Marked Ancestorの帰着

問題をさらに抽象化すると、以下の問題に帰着できる:

1. 重み w の区間 $[a, b]$ を集合に追加
2. 整数 k を包含するような区間のうち、重みが最大のものを求める

この問題は1次元のstabbing-max problemであり、以下の条件を満たすようなデータ構造が存在 [Nekrich, 2011]

- 各クエリの時間計算量: ならし $O(\log n)$
- 空間計算量: $O(m)$ words

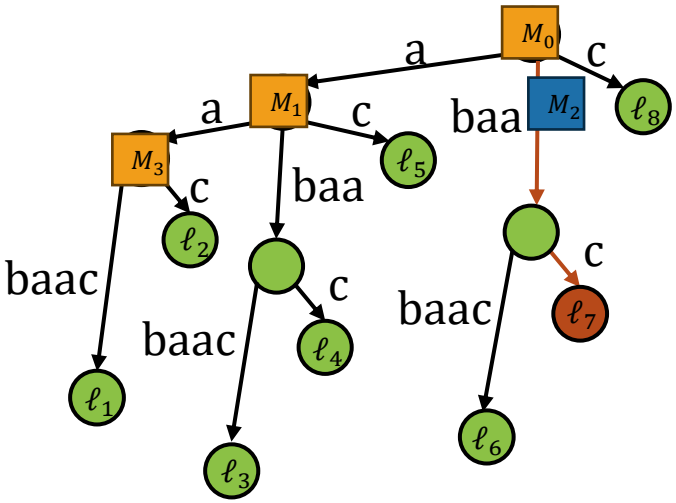
※ m : 区間の個数

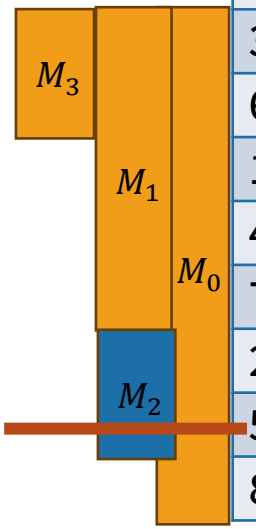
$T[l..r]$ に対応する区間の取得

以下の2つを高速に処理する必要がある:

1. (l, r) が与えられたとき、$T[l..r]$ に対応する葉の区間 $[a, b]$ を求める
2. i が与えられたとき、$T[i..n]$ に対応する葉 ℓ_k を求める

葉を辞書順に並べると、これらは接尾辞配列上の操作に対応





SA[i]	T[SA[i], n]
3	aabaac
6	aac
1	abaabaac
4	abaac
7	ac
2	baabaac
5	baac
8	c

$T[l..r]$ に対応する区間の取得

以下の性質を満たすCDAWG-basedの圧縮索引が存在する

[Bealazzougui and Cunial, CPM 2017]

- $T[i], \mathrm{ISA}[i]$の取得: $O(\log n)$ time
- $T[l..r]$ に対応する区間の取得: $O(\log n)$ time
- 空間計算量: $O(e)$ space

※ISA: 接尾辞配列 SA の逆配列 ($\mathrm{ISA}[\mathrm{SA}[i]] = i$)

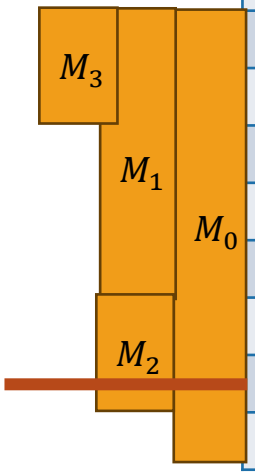
CDAWGによる計算: 処理のまとめ

以下のような手続きの繰り返しでLZ78のfactorを計算できる:

1. $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ に対応する SA 上の位置

   $k = \mathrm{ISA}[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|]$ を探す
2. k を包含する重み w が最大の区間 $[a, b]$ を求める
3. $F_i = F_j T'[w + 1]$ とする
4. F_i に対応する SA 上の区間 $[a, b]$ を探し、重み $w + 1$ の区間 $[a, b]$ を追加

$$T' = \mathrm{abaabaac}$$
$$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa})$$



$\mathrm{SA}[i]$	$T[\mathrm{SA}[i], n]$
3	aabaac
6	aac
1	abaabaac
4	abaac
7	ac
2	baabaac
5	baac
8	c

CDAWGによる計算: 処理のまとめ

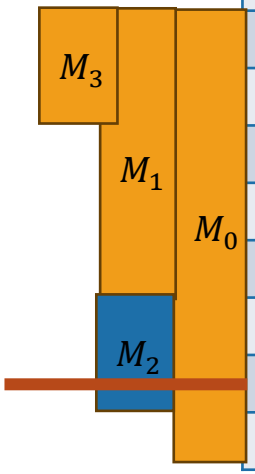
以下のような手続きの繰り返しでLZ78のfactorを計算できる:

1. $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ に対応する SA 上の位置

   $k = \mathrm{ISA}[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|]$ を探す
2. k を包含する重み w が最大の区間 $[a, b]$ を求める
3. $F_i = F_j T'[w + 1]$ とする
4. F_i に対応する SA 上の区間 $[a, b]$ を探し、重み $w + 1$ の区間 $[a, b]$ を追加

$T' = \mathrm{abaabaac}$

$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa})$



$\mathrm{SA}[i]$	$T[\mathrm{SA}[i], n]$
3	aabaac
6	aac
1	abaabaac
4	abaac
7	ac
2	baabaac
5	baac
8	c

CDAWGによる計算: 処理のまとめ

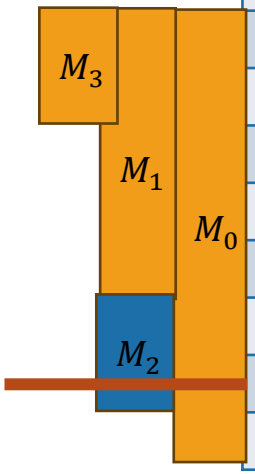
以下のような手続きの繰り返しでLZ78のfactorを計算できる:

1. $T' = T[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|..n]$ に対応する SA 上の位置

   $k = \mathrm{ISA}[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|]$ を探す
2. k を包含する重み w が最大の区間 $[a, b]$ を求める
3. $F_i = F_j T'[w + 1]$ とする
4. F_i に対応する SA 上の区間 $[a, b]$ を探し、重み $w + 1$ の区間 $[a, b]$ を追加

$T' = \mathrm{abaababaac}$

$F = (\varepsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ba})$



$\mathrm{SA}[i]$	$T[\mathrm{SA}[i], n]$
3	aabaac
6	aac
1	abaabaac
4	abaac
7	ac
2	baabaac
5	baac
8	c

CDAWGによる計算: 処理のまとめ

以下のような手続きの繰り返しでLZ78のfactorを計算できる:

1. $T' = T[1 + |F_0| + |F_1| + \cdots, |F_{i-1}|..n]$ に対応する SA 上の位置
   $k = \mathrm{ISA}[1 + |F_0| + |F_1| + \cdots + |F_{i-1}|]$ を探す
2. k を包含する重み w が最大の区間 $[a, b]$ を求める
3. $F_i = F_j T'[w + 1]$ とする
4. F_i に対応する SA 上の区間 $[a, b]$ を探し、重み $w + 1$ の区間 $[a, b]$ を追加

計算量:

- 時間: ならし $O(\log n)$ / factor → 全体で $O(z \log n)$ time
- 空間: $O(e + z)$ space
  - e（CDAWGの辺数）はCDAWG-basedの索引の分
  - z（T のLZ分解のfactor数）はstabbing-maxのためのデータ構造の分

factorの個数分だけ区間を保存する

※ n は T の長さ

LZ78部分文字列圧縮への対応

接尾辞木の場合と同様に、部分文字列圧縮に対応できる

- 始点 l の対応: 前述のアルゴリズムの T' を

  $T' = T[\, l + |F_0| + |F_1| + \cdots + |F_{i-1}| .. n]$ に書き換えれば良い

- 終点 r の対応: 通常通り計算して、 $T[l..r]$ の範囲を

  はみ出したら末尾のfactorを削れば良い

接尾辞木と同じ

計算量: $O(z_{l,r} \log n)$ time・$O(e + z_{l,r})$ space

クエリの度にstabbing-maxのための
データ構造を構築する

※索引自体は $O(e)$ space、作業領域が $O(z_{l,r})$ space

※ $z_{l,r}$ は $T[l..r]$ のLZ分解のfactor数

まとめ・展望

LZ78分解の部分文字列圧縮を以下の計算量で行う手法を提案:

- 時間: $O(z_{l,r} \log n)$
- 空間: $O(e)$

既存手法（$O(z_{l,r})$ time, $O(n)$ space）からの省領域化を達成

展望

- 他の問題も $O(e)$ sizeの索引で解けないか
  - CDAWG上のパスクエリ等も圧縮領域で行えるため、応用は幅広い
- CDAWG以外の索引・他の部分文字列圧縮への一般化
  - 1文字アクセス・ISA・SA範囲の3処理が行える索引ならCDAWGの代わりに使える
  - 他の部分文字列圧縮もこの枠組みで解けるかもしれない（一部は既知）

補足: CDAWGを用いた索引の紹介

[Bealazzougui and Cunial, CPM 2017]

CDAWGを用いた索引: 定理

定理（[Belazzougui and Cunial, CPM 2017]）

以下の処理を $O(\log n)$ 時間で行える

$O(e)$ 領域のデータ構造が存在する:

1. $\mathrm{SA}[i]$ の計算
2. $\mathrm{ISA}[i]$ の計算
3. $T[i]$ の計算
4. $T[l..r]$ に対応する SA 上の区間の取得

説明の流れ

実は前述の4クエリのうち $\mathrm{SA}[i]$ 以外は、以下の方法で求められる

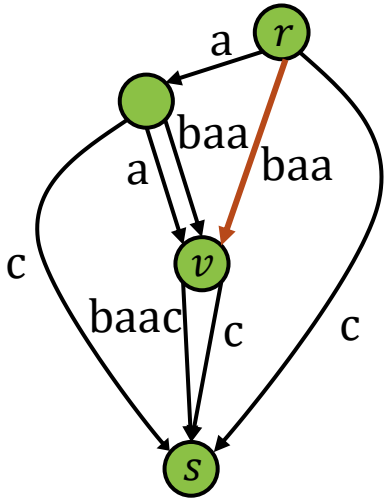
1. T の**長さ k の接尾辞を表すパス**を $O(\log n)$ 時間で求める
   - パスの長さは $O(e)$ になってしまうが、上手い表現（後述）で求める
2. パス上で何らかの処理を行ってクエリの答えを求める

計算時間を気にしない手法から紹介

そのため、まずは長さ k の接尾辞を表す**パスの取得方法**を説明

準備: CDAWGに関する諸定義

- root r : 入次数が 0 の頂点（空文字列を表す頂点）
- sink s :出次数が 0 の頂点（接尾辞を表す頂点）
- $R(v), S(v)$: $r-v$ パス, $v-s$ パスの個数
  - 便宜上 $R(s) = S(s) = 1$ とする
- CDAWGの各辺は (始点, 終点, 辺ラベルの先頭文字, 辺ラベル長) で表現



$T = \mathrm{abaabaac}$

$R(v) = 3$

$S(v) = 2$

赤色の辺は $(r, v, \mathrm{b}, 3)$ のように表現

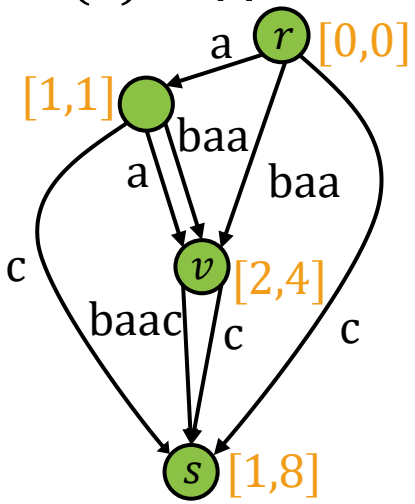
※以降では T の末尾にuniqueな文字があると仮定

準備: CDAWGの性質

- $\mathrm{str}(v)$: CDAWGのroot r → 頂点 v のパスが表す文字列の集合
  - $|\mathrm{str}(v)| = R(v)$ が成り立つ
- $\mathrm{long}(v)$:$\mathrm{str}(v)$ 中で最長の文字列

性質: $\mathrm{str}(v)$ は $\mathrm{long}(v)$ の先頭を $0, 1, \ldots, R(v) - 1$ 文字削った接尾辞の集合

→ $\mathrm{str}(v)$ に含まれる文字列の長さの集合は区間 $[\min_v, \max_v]$ として表せる



$T = \mathrm{abaabaac}$

$\mathrm{str}(v) = \{\mathrm{abaa}, \mathrm{baa}, \mathrm{ba}\}$

$[\min_v, \max_v] = [2,4]$

$T[l,n]$ を表すパスの計算

$v = s$ の状態からroot方向に、

u と v を結ぶ、辺ラベルの先頭文字が c で長さ ℓ の辺

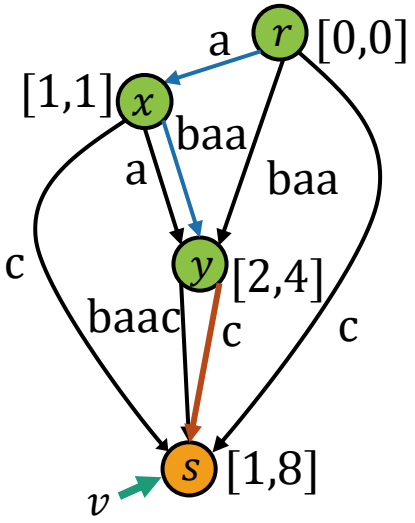
「長さ k の $r - v$ パスにおける**最後の辺** (u, v, c, ℓ)」を見つける処理を繰り返す

$l = 3,$　　$T[l,n] =$ abaabaac

長さ $k = 5$ のパスを見つけたい

最後の辺は $(y, s, \mathrm{c}, 1)$

辺の見つけ方はこれから説明



$T[l,n]$ を表すパスの計算

$v = s$ の状態からroot方向に、

u と v を結ぶ、辺ラベルの先頭文字が c で長さ ℓ の辺

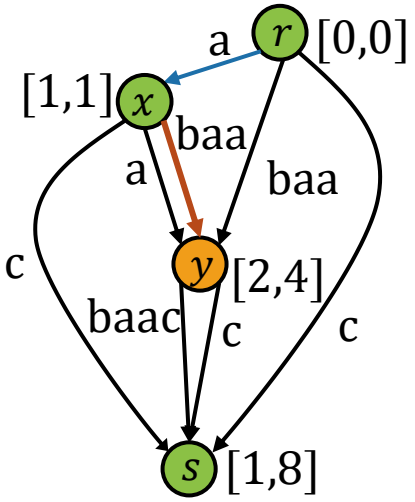
「長さ k の $r - v$ パスにおける最後の辺 (u, v, c, ℓ) 」を見つける処理を繰り返す

$l = 3,$　　$T[l,n] =$ abaabaac

長さ $k = 4$ のパスを見つけたい

最後の辺は $(x, y, \mathrm{b}, 3)$

辺の見つけ方はこれから説明



$T[l,n]$ を表すパスの計算

$v = s$ の状態からroot方向に、

u と v を結ぶ、辺ラベルの先頭文字が c で長さ ℓ の辺

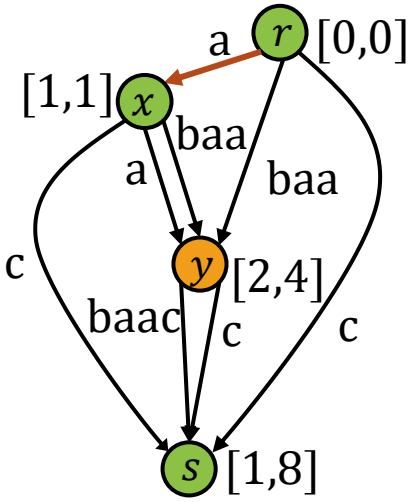
「長さ k の $r - v$ パスにおける**最後の辺** (u, v, c, ℓ)」を見つける処理を繰り返す

$l = 3,$　　$T[l,n] =$ abaabaac

長さ $k = 1$ のパスを見つけたい

最後の辺は $(r, x, \mathrm{a}, 1)$

辺の見つけ方はこれから説明



最後の辺 (u,v,c,ℓ) の特定

長さ k の $r-v$ パス P が存在すると仮定する

このとき、P から最後の辺 (u,v,c,ℓ) を取り除いた $r-u$ パスの長さは $k-\ell$

→ $k-\ell_i \in [\min_{u_i}, \max_{u_i}]$ である辺 (u_i,v,c_i,ℓ_i) が条件を満たす辺の候補

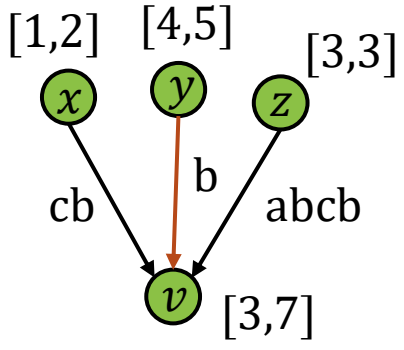
長さ $k-\ell_i$ の $r-u$ パスが存在する

右図の例で長さ 5 のパスを探す場合:

$(x,v,\mathrm{c},2)$: $5-2=3 \notin [1,2]$

$(y,v,\mathrm{b},1)$: $5-1=4 \in [4,5]$

$(z,v,\mathrm{a},4)$: $5-4=1 \notin [3,3]$ より、辿るべき辺は $(y,v,\mathrm{b},1)$



最後の辺 (u, v, c, ℓ) の特定

長さ k の $r-v$ パス P が存在すると仮定する

このとき、P から最後の辺 (u, v, c, ℓ) を取り除いた $r-w$ パスの長さは $k-\ell$

→ $k-\ell_i \in [\min_{u_i}, \max_{u_i}]$ である辺 (u_i, v, c_i, ℓ_i) が条件を満たす辺の候補

長さ $k-\ell_i$ の $r-u$ パスが存在する

重要な性質: $k-\ell_i \in S(u_i)$ を満たす辺 (u_i, v, c_i, ℓ_i) は必ず存在し、一意に定まる

証明: 仮定より長さ k の $r-v$ パスは必ず存在する。$r-v$ パスが表す文字列の集合は接尾辞集合になるため、同じ長さの $r-v$ パスは一意に定まる。

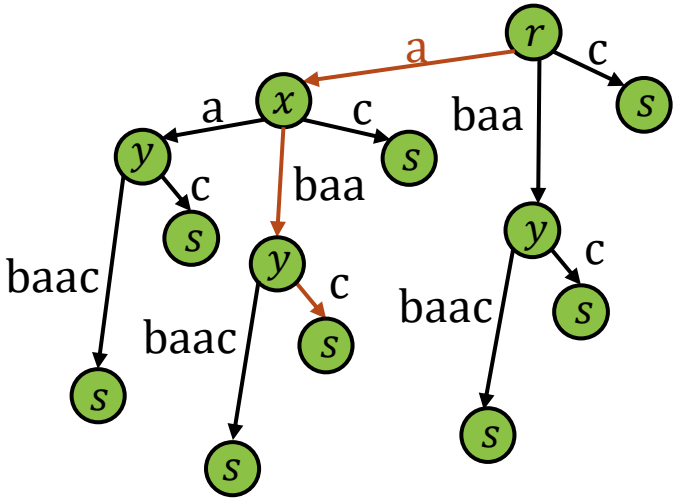
条件を満たす辺がない・複数存在するとパスの存在性・一意性と矛盾（証明終）

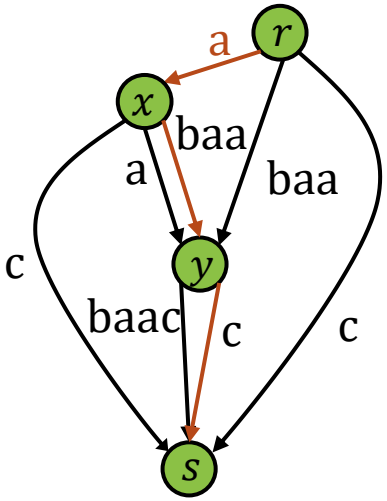
→ 条件を満たす辺を使ったroot方向への移動を繰り返せばパスが計算できる！

$T[l,n]$ を表すパスからの $T[l]$ の計算

1. 前述の方法で $T[l,n]$ に対応するパスを得る
2. パス中でrootと接続している辺が (u,v,c,ℓ) のとき、 $T[l]=c$

※ CDAWG索引の各辺は（始点, 終点, 辺ラベルの1文字目, 長さ）の組





$T[l,n]$ より辞書順で小さい接尾辞の数+1

$T[l,n]$ を表すパスからの $\mathrm{ISA}[l]$ の計算

辺 $e=(u,v,c,\ell)$ について、u を始点とするような、辺ラベルが c より小さい辺の集合を $L(e)$ とする

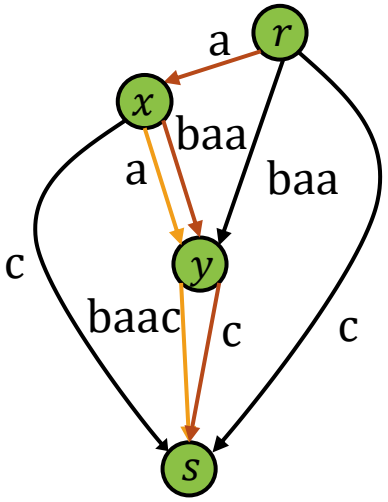
このとき、$\mathrm{ISA}[l]=1+\sum_{e\in P}\sum_{(u,v,c,\ell)\in L(e)} S(v)$ が成り立つ

$l=3,\qquad T[l,n]=\mathrm{abaabaac}$

$L((x,y,\mathrm{b},3))=\{(x,y,\mathrm{a},1)\}$

$L(y,s,\mathrm{c},1))=\{(y,s,\mathrm{b},4)\}$

$\mathrm{ISA}[l]=1+S(y)+S(s)=1+2+1=4$

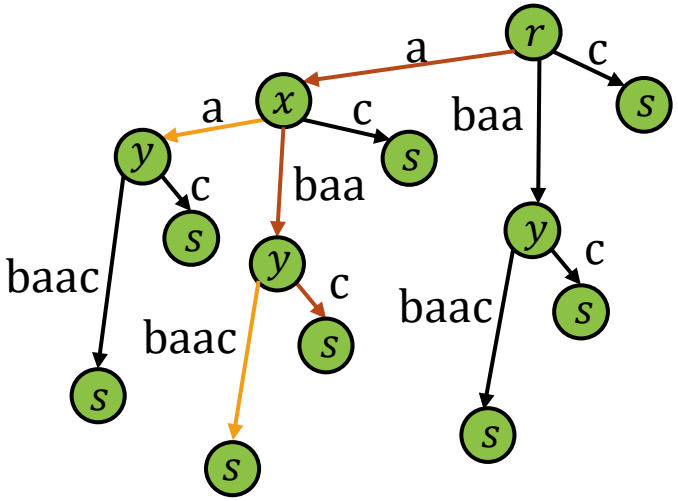


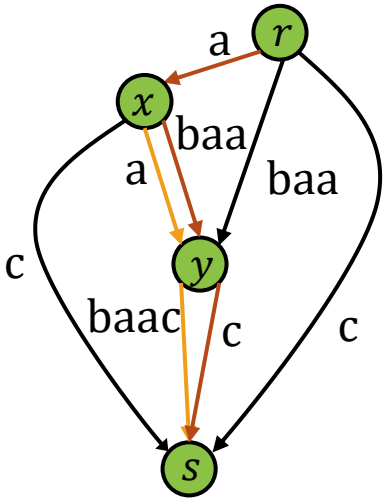
$T[l,n]$ より辞書順で小さい接尾辞の数+1

$T[l,n]$ を表すパスからの $\mathrm{ISA}[l]$ の計算

辺 $e=(u,v,c,\ell)$ について、u を始点とするような、辺ラベルが c より小さい辺の集合を $L(e)$ とする

このとき、$\mathrm{ISA}[l]=1+\sum_{e\in P}\sum_{(u,v,c,\ell)\in L(e)} S(v)$ が成り立つ





内部頂点に対応する SA 上の区間の計算

$T[l..r]$ に対応する SA 上の区間 $[a, b]$ は以下のように求められる:

1. $T[l..n]$ を表すパスの深さ $r - l + 1$ 地点までを表すパス P' を得る
2. $\mathrm{ISA}[l]$ の計算と同じ方法で、SA 上の区間の左端 a を求める
3. 区間の右端 b を $b = a + S(v) - 1$ で求める

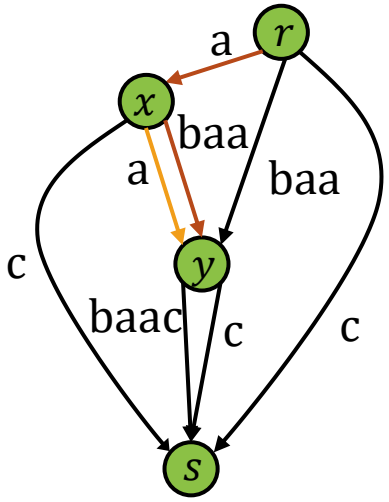
ただし、v はパス P' の終端点

$(l, r) = (3,6), \quad T[l, r] = \mathrm{abaabaac}$

$E(x) = \{(x, y, \mathrm{a}, 1)\}$

$a = 1 + S(y) = 3$

$b = a + S(y) - 1 = 4$

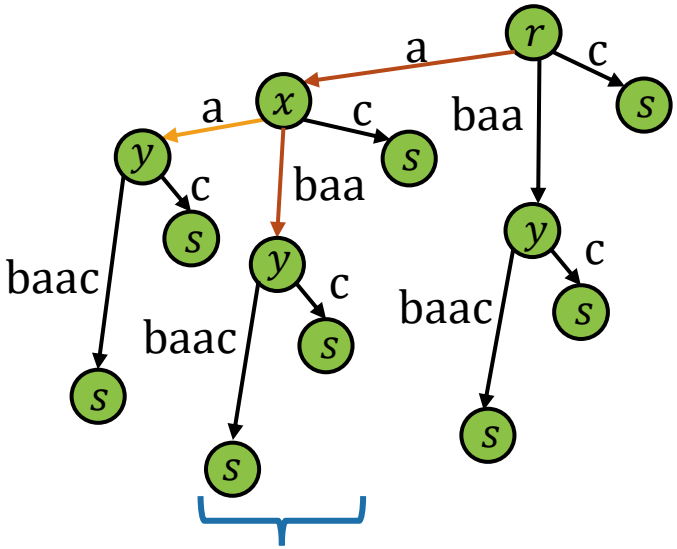


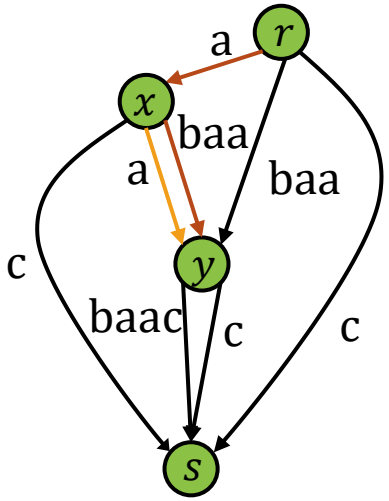
内部頂点に対応する SA 上の区間の計算

$T[l..n]$ を表すパスの深さ $r-l+1$ 地点までを表すパス P' を得る

1. $\mathrm{ISA}[l]$ の計算と同じ方法で、 SA 上の区間の左端 a を求める
2. 区間の右端 b を $b = a + S(v) - 1$ で求める

ただし、v はパス P' の終端点





処理の高速化

説明では計算量について触れなかったが、この手法はとても遅い
（愚直に実装すると$O(e)$時間かかる）

以降では、この処理を2段階に分けて高速化する

1. Symmetric Centroid Path Decomposition + 二分探索: $O(\log^2 n)$
2. ↑の手法 + Interval Biased Search Tree: $O(\log n)$

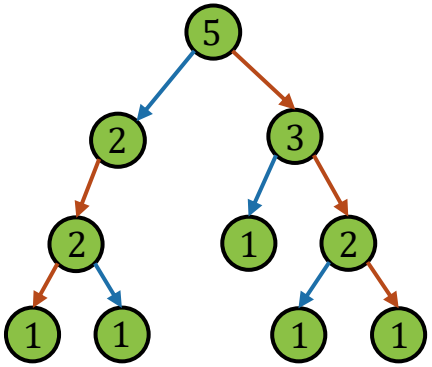
Centroid Path Decompositionとも呼ぶ

Heavy Path Decomposition (HPD)

HPD: 木の辺をHeavy Edgeからなるパスと Light Edgeに分解する手法

各頂点の子方向の辺を、以下の二種類に分類する

- Heavy Edge : $S(v)$ が最大の辺（複数ある場合はどれか一つ
  - $S(v)$ は v から葉へのパスの数の総和
- Light Edge: Heavy Edge以外の辺



HPD: 定理1

始点を共有する辺のうち $R(v)$ が最大の辺はHeavy Edge、そうでない辺はLight Edge

$\lfloor \log_2 S(r) \rfloor = m$ とおくと、以下の定理が成り立つ。

定理1: Light Edgeの個数

任意のパス中のLight Edgeの個数は m 以下

証明:

1. u の子の集合を $C(u)$ とおくと、 $S(u) = \sum_{v \in C(u)} S(v)$
2. $u - v$ にLight Edgeがあるとき、始点を共有するHeavy Edge $u - v'$ が必ず存在
3. $S(u) \geq S(v) + S(v'), S(v) \leq S(v')$ より、 $S(u) \geq 2S(v)$ が成り立つ
4. Light Edgeを通るたびに $S(v)$ が半分以下になるため、パス中のLight Edgeの個数は $\lfloor \log_2 S(r) \rfloor = m$ 以下

HPD: 定理2

始点を共有する辺のうち $R(v)$ が最大の辺はHeavy Edge、そうでない辺はLight Edge

定理2: Heavy Path

同じ頂点を始点・終点とするHeavy Edgeの組は存在しない

（→Heavy Edgeのみからなるグラフはパスの集合になる）

証明（ほぼ自明）：

1. 木なので、終点を共有するHeavy Edgeはない
2. Heavy Edgeは始点ごとに1つなため、始点を共有するHeavy Edgeもない

Symmetric Centroid Path Decomposition (SCPD)

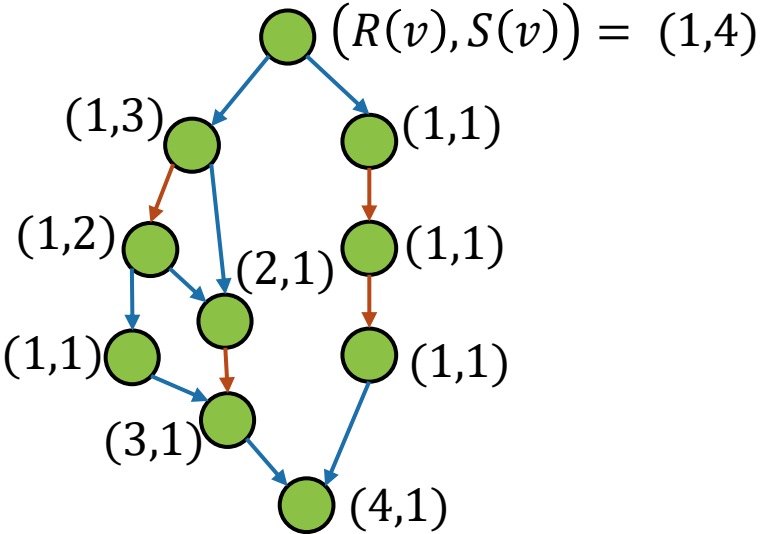
SCPD: root/sinkが1つのDAGに対するHeavy Path Decomposition

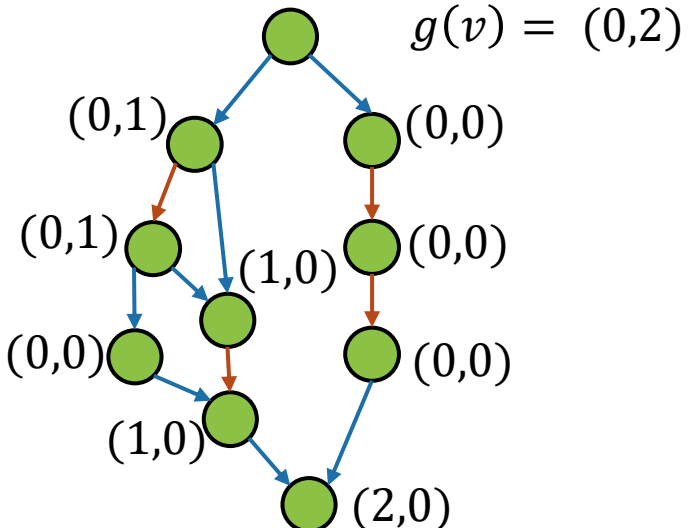
root, sinkからのパスの数を2進表記したときの最上位bitの位置

$g(v) = (\lfloor \log_2 R(v) \rfloor, \lfloor \log_2 S(v) \rfloor)$ とする

DAGにおいて $u - v$ を結ぶ辺のうち、

$g(u) = g(v)$ を満たす辺をHeavy Edge、そうでない辺をLight Edgeと呼ぶ





Symmetric Centroid Path Decomposition (SCPD)

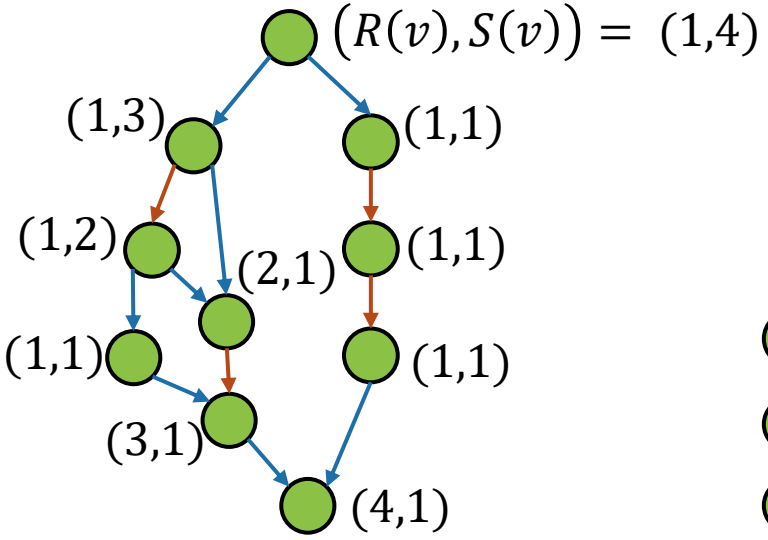
SCPD: root/sinkが1つのDAGに対するHeavy Path Decomposition

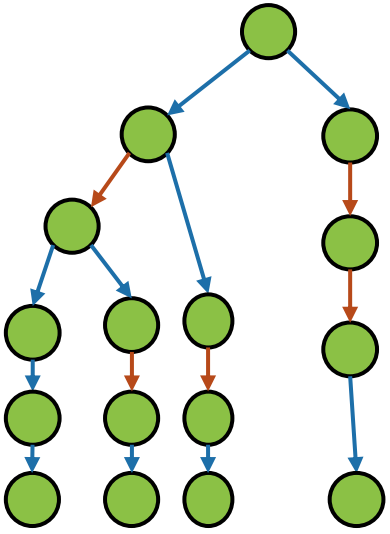
root, sinkからのパスの数を2進表記したときの最上位bitの位置

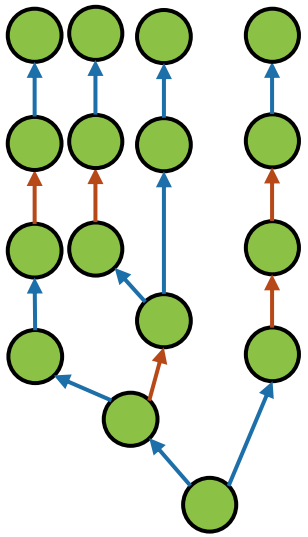
$g(v) = (\lfloor \log_2 R(v) \rfloor, \lfloor \log_2 S(v) \rfloor)$ とする

DAGにおいて $u - v$ を結ぶ辺のうち、

$g(u) = g(v)$ を満たす辺をHeavy Edge、そうでない辺をLight Edgeと呼ぶ







SCPD: 定理1

$g(u) = g(v)$ を満たす辺はHeavy Edge、そうでない辺はLight Edge

$\lfloor \log_2 R(s) \rfloor = \lfloor \log_2 S(r) \rfloor = m$ とおくと、以下の定理が成り立つ。

定理1: Light Edgeの個数

任意の $r - s$ パス中のLight Edgeの個数は $2m$ 以下

証明:

$g(r) = (0, m), g(s) = (m, 0)$ が成り立つ。
任意の辺 (u, v) について $R(u) \leq R(v), S(u) \geq S(v)$ であることから、$g(u) \neq g(v)$ である場合は$\lfloor \log_2 R(u) \rfloor < \lfloor \log_2 R(v) \rfloor$ または $\lfloor \log_2 S(u) \rfloor > \lfloor \log_2 S(v) \rfloor$ となる。

Light Edgeを通るたびに $\lfloor \log_2 R(v) \rfloor$ が増えるか $\lfloor \log_2 S(v) \rfloor$ が減り、パスの両端は $g(r) = (0, m), g(s) = (m, 0)$ なため、Light Edgeの個数は $2m$ 以下。（証明終）

SCPD: 定理2

$g(u) = g(v)$ を満たす辺はHeavy Edge、そうでない辺はLight Edge

以下の定理が成り立つ。

定理2: Heavy Path

同じ頂点を始点・終点とするHeavy Edgeの組は存在しない

（→Heavy Edgeのみからなるグラフはパスの集合になる）

証明:

同じ頂点を始点とするHeavy Edgeの組 $(u, v), (u, v')$ が存在すると仮定する。

Heavy Edgeの定義より $\lfloor \log_2 S(u) \rfloor = \lfloor \log_2 S(v) \rfloor = \lfloor \log_2 S(v') \rfloor$ だが、

$\lfloor \log_2 S(u) \rfloor \geq \lfloor \log_2 S(v) + \log_2 S(v') \rfloor = \lfloor \log_2 S(v) \rfloor + 1$ となるため、矛盾。

終点が一致する場合も $\lfloor \log_2 R(v) \rfloor$ を用いて同様の議論ができる。（証明終）

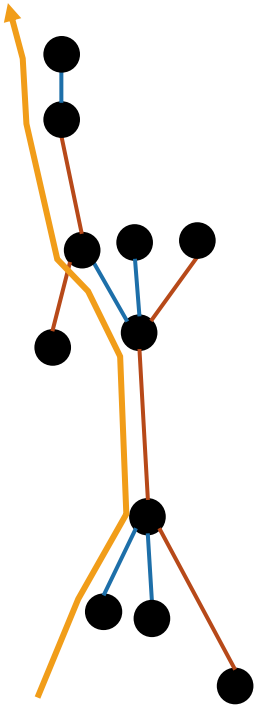
探したいパスとHeavy Pathの関係

探したいパスはいくつかのHeavy Path（Heavy Edgeからなる極大パス）とLight Edgeに分けられる

CDAWGでは $R(s) = S(r) = n$ なので、
パス上のLight Edgeは $2\log n$ 個以下（定理1）

Light Edgeの個数が $O(\log n)$ なため、Heavy Pathの個数も $O(\log n)$ 個

- パスをHeavy Path上の一部分 or Light Edgeの列の形で $O(\log n)$ wordsで表現できる
- Heavy PathとLight Edgeを $O(\log n)$ 時間で探索できれば、全体の計算量を $O(\log^2 n)$ 時間にできる



Heavy Path上の二分探索

Heavy Path（Heavy Edgeからなる極大パス）上の探索を高速化したい

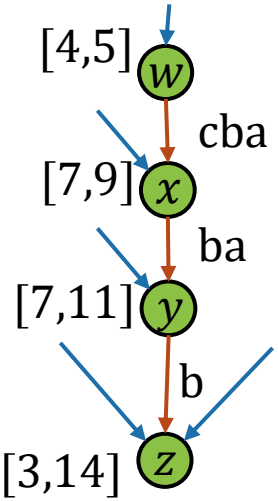
定義:

- $d(a,b)$：$a-b$ を結ぶHeavy Pathの長さ（パス文字列の長さ）
  - ただし、a,b は同じHeavy Pathに属するとする

長さ k の $r-v$ パス P と v を含むHeavy Pathの共通部分を探すことを考える

求めたいパス P がHeavy Path中の $u-v$ パスを含むかは、

$k-d(u,v) \in [\min_u, \max_u]$ によって判定できる



右図の例で長さ 9 の $r-z$ パスとの共通部分を探す場合

$9-d(w,z)=3 \notin [4,5]$
$9-d(x,z)=6 \notin [7,9]$
→ $r-z$ パスに含まれない

$9-d(y,z)=8 \in [7,11]$
$9-d(z,z)=9 \in [3,14]$
→ $r-z$ パスに含まれる

Heavy Path上の二分探索

Heavy Path（Heavy Edgeからなる極大パス）上の探索を高速化したい

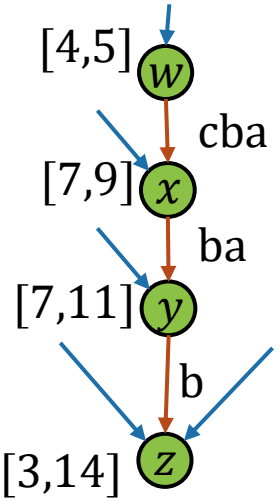
定義:

- $d(a,b)$: $a-b$ を結ぶHeavy Pathの長さ（パス文字列の長さ）
  - ただし、a,b は同じHeavy Pathに属するとする

長さ k の $r-v$ パス P と v を含むHeavy Pathの共通部分を探すことを考える

求めたいパス P がHeavy Path中の $u-v$ パスを含むかは、

$k-d(u,v) \in [\min_u, \max_u]$ によって判定できる



Heavy Pathとの共通部分を求める計算量:

- u を決めたときの判定は定数時間で可能
  各頂点 u について、$d(u,h)$ と $\min_u, \max_u$ を保存しておく
  （ただし、h は u を含むHeavy Pathの終点）
  $d(u,v)=d(u,h)-d(v,h)$ より、 $d(u,v)$ は定数時間で計算できる
- Heavy Path上で二分探索すると、Heavy Path毎に $O(\log n)$ time

Light Edgeの二分探索

Light Edgeの探索を高速化したい

長さ k の $r-v$ パス P が辺 (u,v,c,ℓ) を含むかは $k-\ell \in [\min_u, \max_u]$ で判定できる

$r-u_i-v$ パスが表す文字列の長さの最小値

v を終点として持つ各辺 (u_i,v,c_i,ℓ_i) を $\min_{u_i}+\ell_i$ でソートしておくことで、

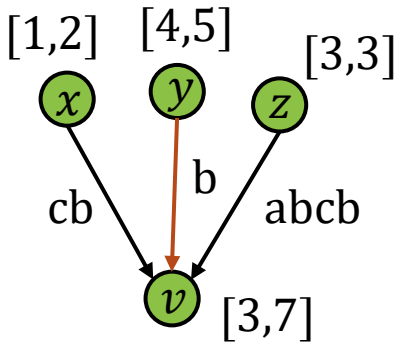
二分探索によって $O(\log n)$ 時間で目的のLight Edgeを特定できる

$(x,v,\mathrm{c},2)$: $\min_u+\ell_i = 1+2 = 3$

$(y,v,\mathrm{b},1)$: $\min_u+\ell_i = 4+1 = 5$

$(z,v,\mathrm{a},4)$: $\min_u+\ell_i = 2+4 = 7$

長さが 6 のパスを探す場合、 $\min_u+\ell_i$ が 6 以下で最大の辺 $(y,v,\mathrm{b},1)$ が答え



現時点の計算量

- Heavy Edgeの探索: Heavy Path毎に $O(\log n)$ 時間
- Light Edgeの探索: 辺1つの特定に $O(\log n)$ 時間

CDAWGでは $R(s) = S(r) = n$ なので、
パス上のLight Edgeは $2\log n$ 個以下（定理1）

$r - s$ パスは高々 $O(\log n)$ 個しかLight Edgeを持たず、Heavy Pathの個数も $O(\log n)$ となるため、計算量は合計で $O(\log^2 n)$

→ $O(\log n)$ に改善する手法を紹介

Biased Search Tree

- 重み付きの集合 $S = \{(a_1, w_1), \ldots, (a_k, w_k)\}$ 上の検索を高速に行うデータ構造
  - a_i はキー $(a_1 < \cdots < a_k)$
  - w_i は要素重み $(w_i \geq 1)$
- 要素 (a_i, w_i) を検索する計算量は $O\left(1 + \log \frac{W}{w_i}\right)$
  - W は要素重みの総和

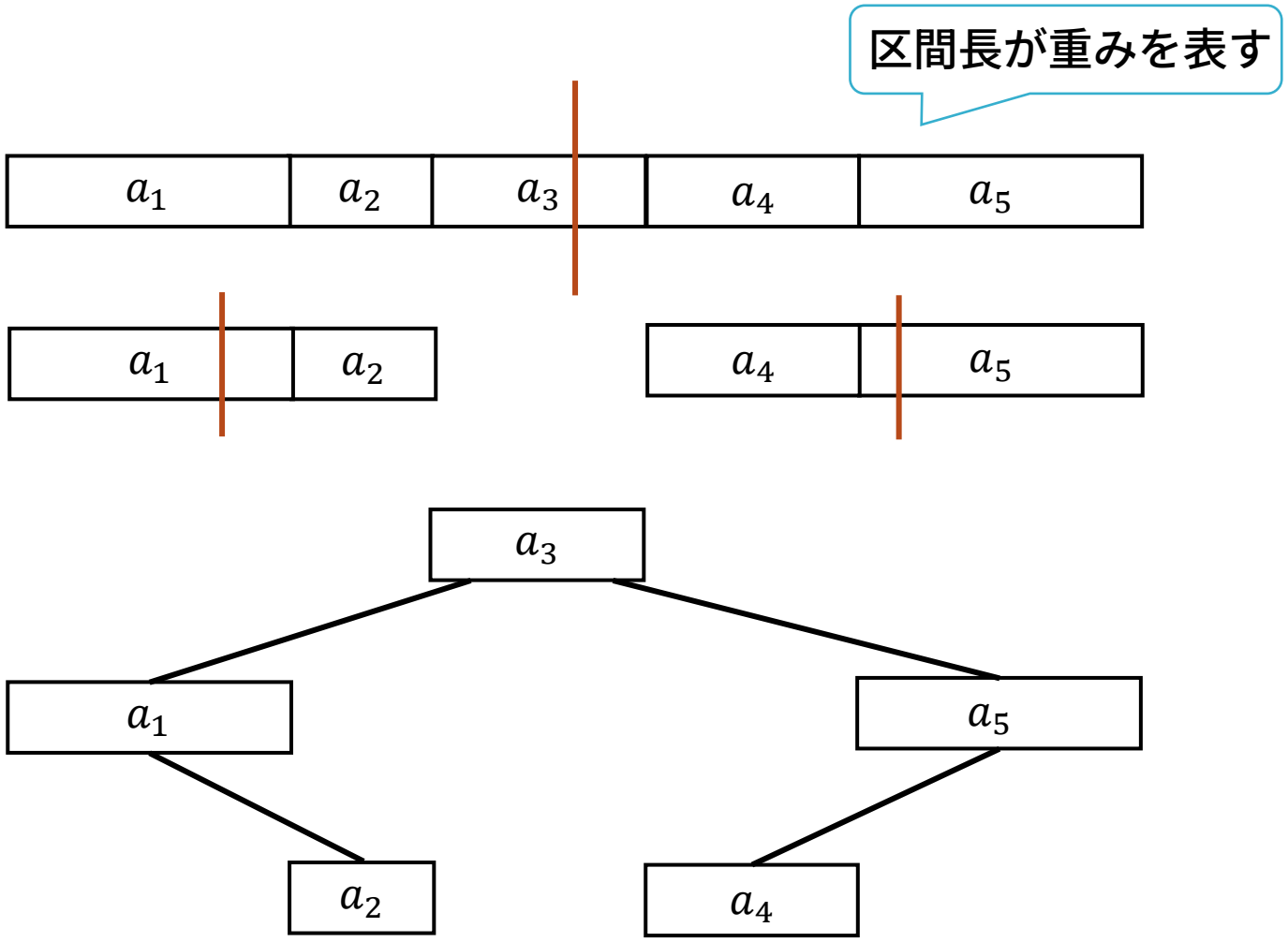
Biased Search Treeの構造

- $S = \{(a_1, w_1), \ldots, (a_k, w_k)\}$ について、要素 (a_i, w_i) の累積重み z_i を $z_i = \sum_{j=1}^{i} w_j$ で定義
- 集合 S のBST: 以下のように再帰的に定義
  - $S = \{(a_i, x_i)\}$ のとき: a_i のみからなる根のみの木
  - $|S| \geq 2$ のとき: 以下のように定義

重み和の半分で分割

  - ▲ 根頂点: 累積重みが $\frac{W}{2}$ 以上となる中で最小の要素 (a_i, x_i) の値 a_i
  - ▲ 左の子: $\{(a_1, w_1), \ldots, (a_{i-1}, w_{i-1})\}$ のBST
  - ▲ 右の子: $\{(a_{i+1}, w_{i+1}), \ldots, (a_k, w_k)\}$ のBST

Biased Search Treeの例



Biased Search Treeでの検索

- 検索: 通常の二分探索木と同様に比較しながら潜れば良い
- 要素 (a_i, w_i) を検索する計算量は $O\left(1 + \log \frac{W}{w_i}\right)$
  - 木を子方向へ移動する度、部分木の重み和は半分になる
    - 深さ d の頂点の重み和は $\frac{W}{2^d}$ 以下
  - 探索を続けると、a_i を表す頂点に必ず到達
  - a_i を表す頂点以下の部分木重み和は必ず w_i 以上

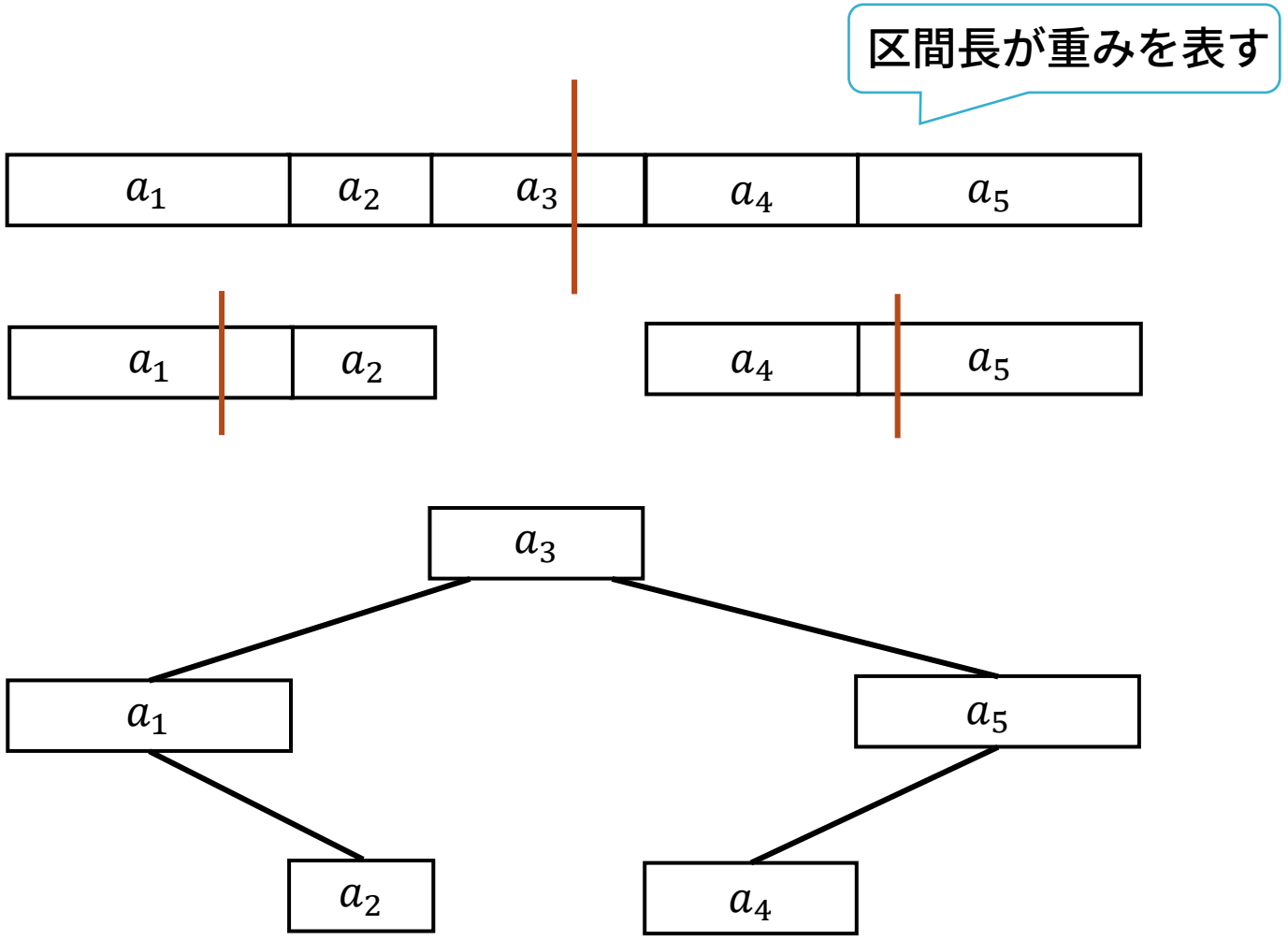
→ $w_i \leq$ 部分木重み和 $\leq \frac{W}{2^d}$ より、 $d \leq \log_2 \frac{W}{w_i}$

Interval Biased Search Tree (IBST)

IBST: Biased Search Treeを区間の集合に対応させた亜種

- 集合 $S = \{(a_1, [l_1, r_1]) \ \ldots, (a_k, [l_k, r_k])\}$ 上の検索を高速に行うデータ構造
  - 区間 $[l_i, r_i]$ に値 a_i が紐づいている
  - $[l_1, r_1], \ldots, [l_k, r_k]$ はある区間の分割である必要がある
- Biased Search Treeを区間に対応させただけなので詳細は省略
- 区間長の和が W で探したい区間の長さが w_i のとき、
  計算量は $O\left(1 + \log \frac{W}{w_i}\right)$

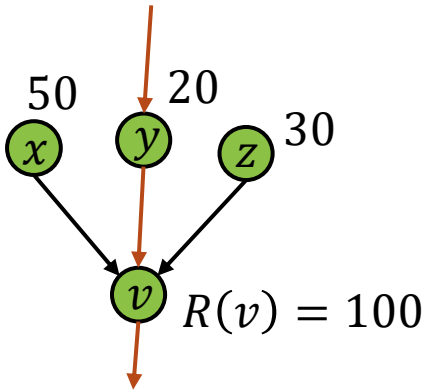
Biased Search Treeの例（再掲）



IBSTを使うアイデア

- CDAWGのsink→root方向の探索時の性質をおさらい
  - 探索中では $R(v)$ は広義単調減少で、 $R(s) = n,\ R(r) = 1$
  - v 終点の辺の始点の多重集合を S_v とすると、 $R(v) = \sum_{u \in S_v} R(u)$

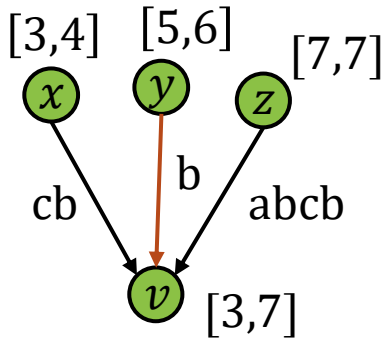
- 上の性質とIBSTを用いて、
  $O(\text{パス長} + \log n)$ でパスを得る手法を説明
  - SCPDを使わない手法
  - 後でSCPDと組み合わせて $O(\log n)$ に改善



IBSTによる高速化

- 全ての頂点について、その頂点に入る辺をIBSTで管理
  - 辺 (u, v, c, ℓ) は区間 $[\min_u + \ell, \max_u + \ell]$ に対応づける
    - 区間は辺 (u, v, c, ℓ)を通るような $r - u - v$ パスの長さの集合を表す
    - 区間長は $R(u)$ で、頂点 v のIBSTの区間長の総和は $R(v)$

- このときのパス検索の計算量: $O(\text{パス長} + \log n)$
  - 次ページで詳しく説明



IBSTによる高速化

- このときのパス検索の計算量: $O(\text{パス長} + \log n)$

  $r - s$ パス $v_1, \ldots, v_t$ を考える $(v_1 = r, v_t = s)$

  v_i のIBSTを用いて v_{i-1} を求める計算量は $O\left(1 + \log \frac{R(v_i)}{R(v_{i-1})}\right)$

  したがって、計算量の総和は

$$\sum_{i=t}^{i=2} O\left(1 + \log \frac{R(v_i)}{R(v_{i-1})}\right)$$

間の項が打ち消せる

$$= \sum_{i=t}^{i=2} O(1 + \log R(v_i) - \log R(v_{i-1}))$$

$$= O(t + \log R(v_t) - \log R(v_1))$$

$$= O(\mathrm{t} + \log n)$$

さらなる改善

- 現在のパス検索の計算量: $O(\text{パス長} + \log n)$
  - $O(\text{パス長})$ の部分がネック
- ボトルネックの原因: 辺を一本ずつ遡っていること

  → Heavy Pathを高速に探索したい

- これからやること: Heavy Pathの探索をIBSTで高速化

Heavy Path上の二分探索（再掲）

Heavy Path（Heavy Edgeからなる極大パス）上の探索を高速化したい

定義:

- $d(a,b)$：$a-b$ を結ぶHeavy Pathの長さ（パス文字列の長さ）
  - ただし、a,b は同じHeavy Pathに属するとする

長さ k の $r-v$ パス P と v を含むHeavy Pathの共通部分を探すことを考える

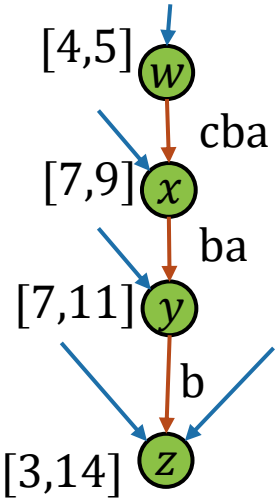
求めたいパス P がHeavy Path中の $u-v$ パスを含むかは、

$k-d(u,v) \in [\min_u, \max_u]$ によって判定できる

右図の例で長さ 9 の $r-z$ パスとの共通部分を探す場合

$9-d(w,z)=3 \notin [4,5]$
$9-d(x,z)=6 \notin [7,9]$
} $r-z$ パスに含まれない

$9-d(y,z)=8 \in [7,11]$
$9-d(z,z)=9 \in [3,14]$
} $r-z$ パスに含まれる



$O(\log n)$ 時間アルゴリズムの概略

頂点 v を含むHeavy Pathと求めたいパスの共通部分を $u-v$ とし、

求めたいパス上で u より1つroot側の頂点を t とする

- $u-v$ 間はHeavy Pathで、$t-u$ 間はLight Edge

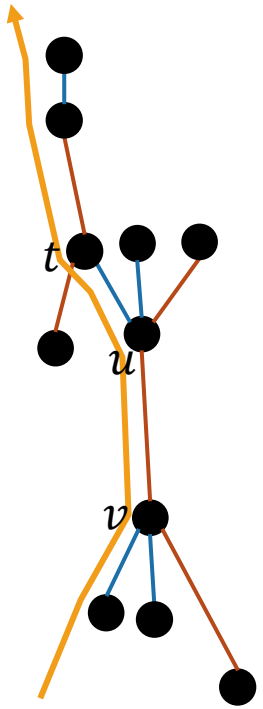
このとき、v から t を求める操作が $O\left(\log\frac{R(v)}{R(t)}\right)$ 時間で行える場合、

探索全体の計算量が $O(\log n)$ になる

- Biased Search Treeと同様の計算量評価によって示せる

※どのHeavy Pathに属さない頂点も存在するが、

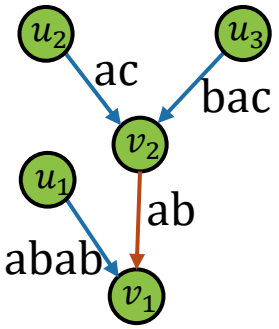
この場合は 1 頂点からなるHeavy Pathだと考えれば同様に扱える



頂点・Light Edgeと区間の対応付け

- Heavy Path上の頂点をsink側から $v_1, \ldots, v_t$ とおく
  - 通常のパスと順序が逆なことに注意
- Heavy Path上の頂点 v_i について、

  $I_H(v_i) = \left[\min_{v_i} + d(v_i, v_1), \max_{v_i} + d(v_i, v_1)\right]$ とする
  - $r - v_i - v_{i-1} - \cdots - v_1$ パスの文字列長の集合



- Heavy Pathに繋がるLight Edge $e = (u, v_i, c, \ell)$ について、

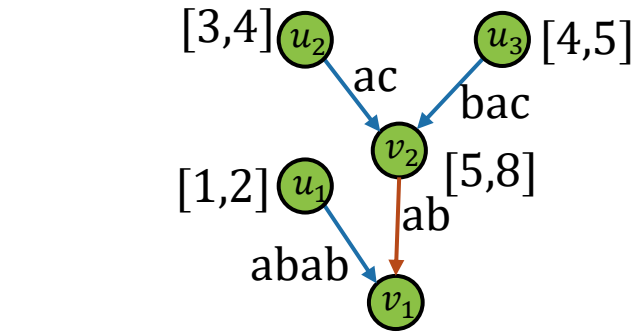
  $I_L(e) = \left[\min_{v_i} + d(v_i, v_1) + \ell, \max_{v_i} + d(v_i, v_1) + \ell\right]$ とする
  - e を通る $r - u - v_i - v_{i-1} - \cdots - v_1$ パスの文字列長の集合

頂点・Light Edgeと区間の対応付け

- $I_H(v_i) = \left[\min_{v_i} + d(v_i, v_1), \max_{v_i} + d(v_i, v_1)\right]$
  - $r - v_i - v_{i-1} - \cdots - v_1$ パスの文字列長の集合
- Light Edge $e = (u, v_i, c, \ell)$について、

  $I_L(e) = \left[\min_{v_i} + d(v_i, v_1) + \ell, \max_{v_i} + d(v_i, v_1) + \ell\right]$
  - e を通る $r - u - v_i - v_{i-1} - \cdots - v_1$ パスの文字列長の集合

$I_H(v_1) = [5,10]$
$I_H(v_2) = [7,10]$
$I_L\big((u_1, v_1, \mathrm{a}, 4)\big) = [5,6]$
$I_L\big((u_2, v_2, \mathrm{a}, 3)\big) = [7,8]$
$I_L\big((u_3, v_2, \mathrm{b}, 2)\big) = [9,10]$

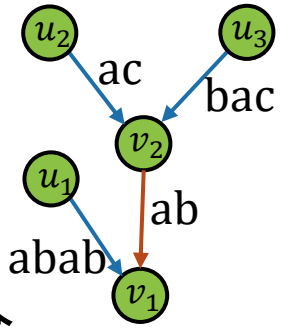


$\left[\min_{v_1}, \max_{v_1}\right] = [5,10]$

頂点・Light Edgeと区間の対応付け

- $I_H(v_i) = \left[\min_{v_i} + d(v_i, v_1), \max_{v_i} + d(v_i, v_1)\right]$
  - $r - v_i - v_{i-1} - \cdots - v_1$ パスの文字列長の集合
- Heavy Pathに繋がるLight Edge $e = (u, v_i, c, \ell)$について、

  $I_L(e) = \left[\min_{v_i} + d(v_i, v_1) + \ell, \max_{v_i} + d(v_i, v_1) + \ell\right]$
  - e を通る $r - u - v_i - v_{i-1} - \cdots - v_1$ パスの文字列長の集合



- **性質: 以下の2つが成り立つ**
  - $I_H(v_t) \subseteq I_H(v_{t-1}) \subseteq \cdots \subseteq I_H(v_1)$
  - v_i が終点のLight Edgeの集合を $L(v_i)$ とおいたとき、

    $I_H(v_i) \setminus I_H(v_{i+1}) = \cup_{X \in L(v_i)} I_L(X)$ であり、右辺の各区間は重ならない

    $i = t$ なら空集合

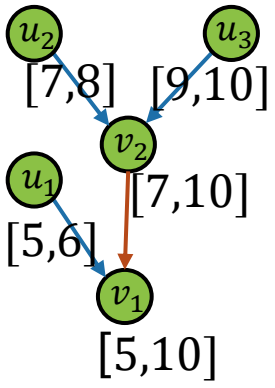
頂点・Light Edgeと区間の対応付け

- **性質**: 以下の2つが成り立つ（再掲）
  - $I_H(v_t) \subseteq I_H(v_{t-1}) \subseteq \cdots \subseteq I_H(v_1)$
  - v_i が終点のLight Edgeの集合を $L(v_i)$ とおいたとき、

    $I_H(v_i) \setminus I_H(v_{i+1}) = \bigcup_{X \in L(v_i)} I_L(X)$ であり、右辺の各区間は

    重なりをもたない

$I_H(v_1) = [5,10]$
$I_H(v_2) = [7,10]$
$I_L\big((u_1, v_1, \mathrm{a}, 4)\big) = [5,6]$
$I_L\big((u_2, v_2, \mathrm{a}, 3)\big) = [7,8]$
$I_L\big((u_3, v_2, \mathrm{b}, 2)\big) = [9,10]$

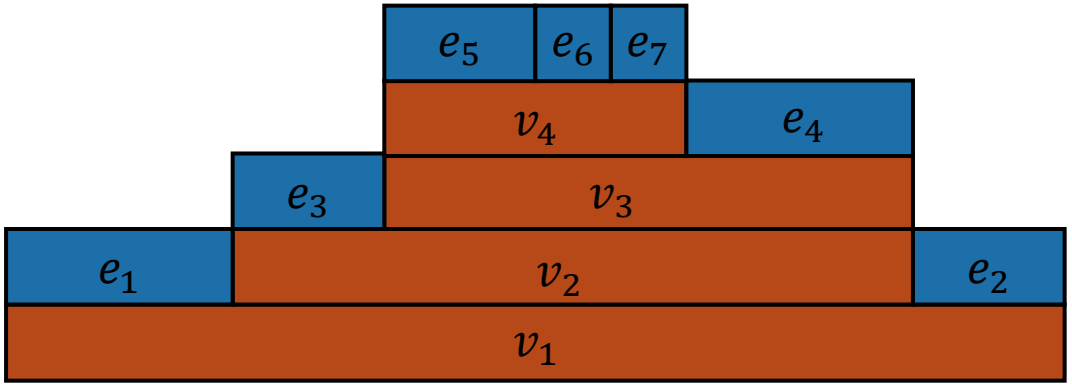


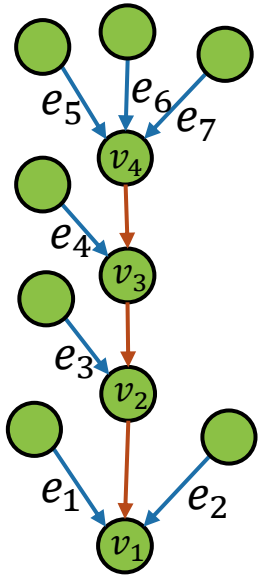
目標: Heavy Path→Light Edge の遷移を $O\left(\log \frac{R(\text{遷移元頂点})}{R(\text{遷移先頂点})}\right)$ に改善したい

Heavy Path探索のIBST化

- I_H, I_L の性質から、Heavy Pathの各頂点・Heavy Pathに接続する各Light Edgeが表す区間は、下のように表せる
  - Heavy Pathに接続するLight Edgeが表す区間の和集合が $I_H(v_1)$ と一致する
- Heavy Path毎に、Heavy Pathに接続するLight Edgeの集合をIBSTで保持することを考える

→ IBSTを単純に使うだけでは計算量保証ができない



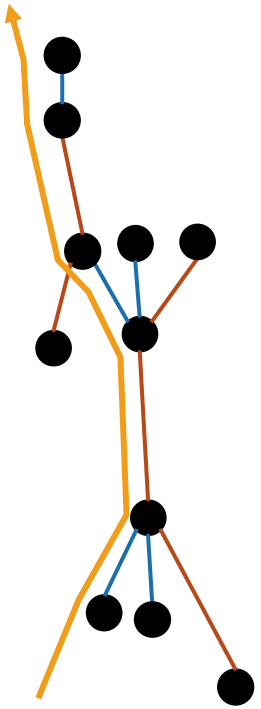


目標: Heavy Path→Light Edge の遷移を $O\left(\log \frac{R(\text{遷移元頂点})}{R(\text{遷移先頂点})}\right)$ に改善したい

Heavy Path + IBST の問題点

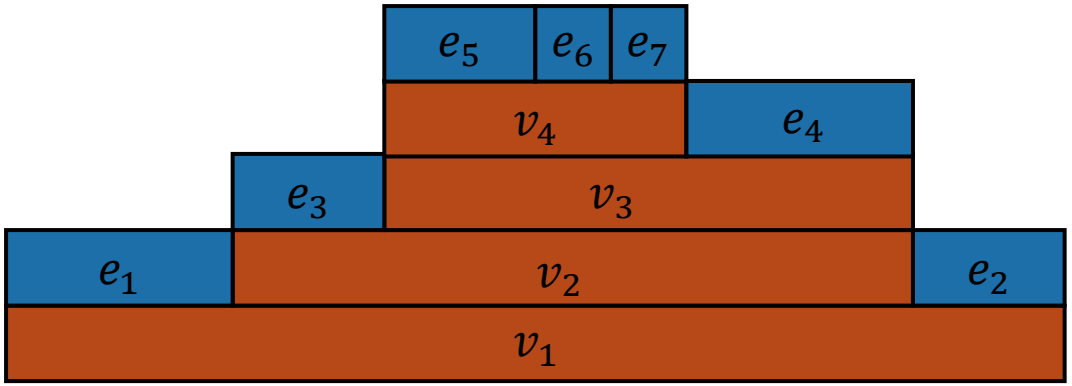
IBST化だけでは計算量は改善しない

原因: Light Edgeで遷移した先がHeavy Edgeの
途中になってしまうことがある



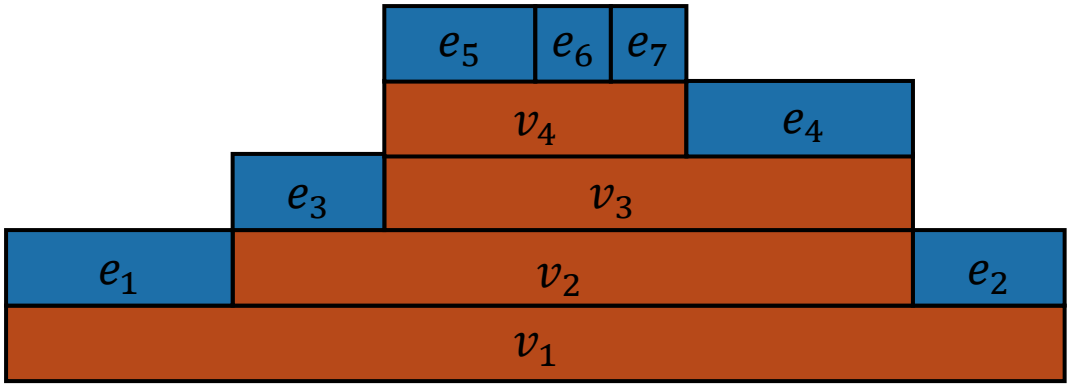
例: v_3 を始点に二分探索をして辺 e_6 を探す場合

- $O(\log n)$ の達成のために要求される計算量: $O\left(\log \frac{|I_V(v_3)|}{|I_e(e_6)|}\right)$
- 実際にかかる時間: $O\left(\log \frac{|I_V(v_1)|}{|I_e(e_6)|}\right)$



計算量の削減

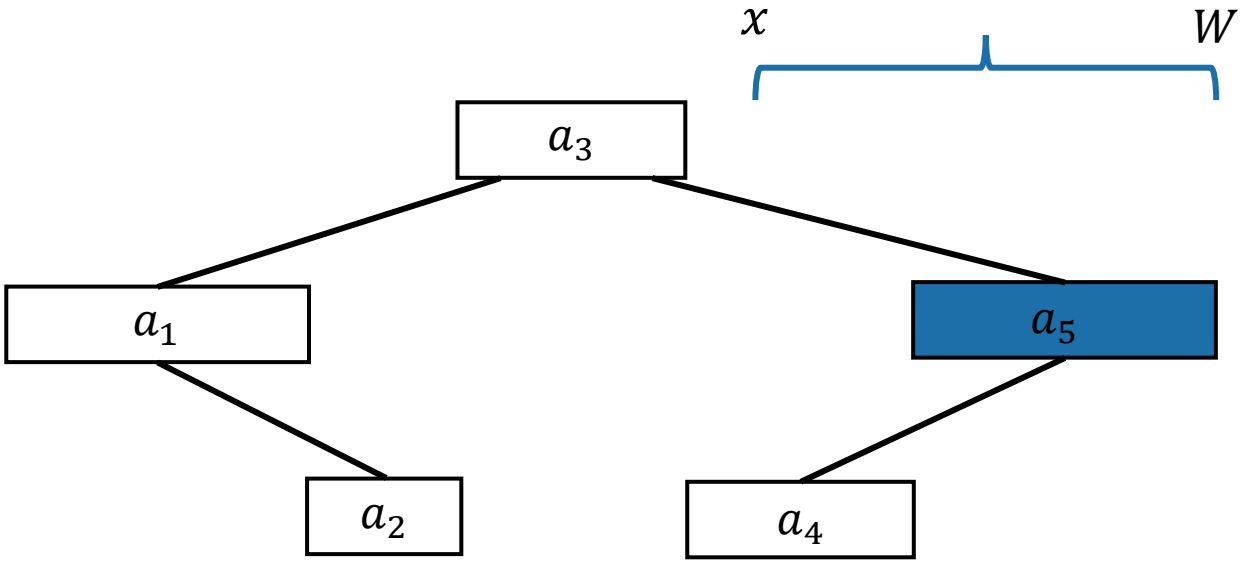
- 改善したいこと: Heavy Pathの途中から探索したい
- Heavy Pathの各頂点 v_i にjump pointerを保持して途中から探索することで、計算量を抑えられる
  - Jump pointer: 探索をショートカットできるようなIBSTのノードへのポインタ
  - v_3 にいる状態から e_1 や e_3 を探索しないようにして高速化



Jump pointer

定義: jump pointer

区間全体の和集合が $[1, W]$ であるようなIBSTにおいて、ノード p 以下の部分木の区間の和集合が区間 $[x, W]$ を包含するような最深のノード p_x を、位置 x のjump pointerと呼ぶ。



Jump pointerを使った場合の計算量

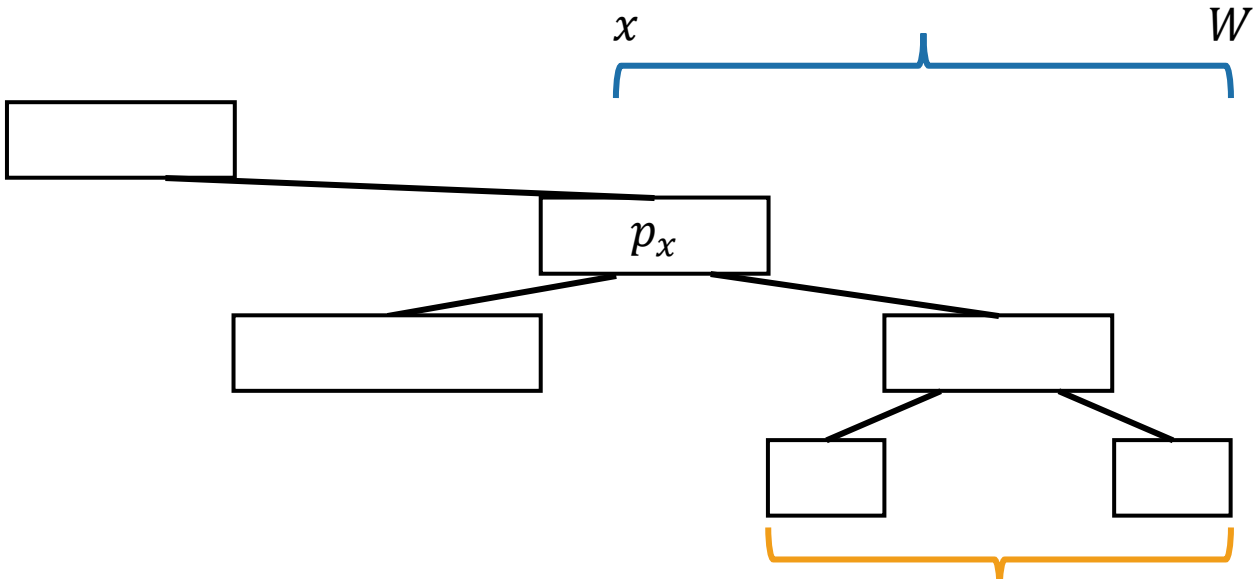
jump pointer: 部分木が $[x, W]$ を包含する最深ノード p_x へのポインタ

p_x から探索を始めて区間 $[l, r] \in [x, W]$を検索する計算量は $O\left(1 + \log \frac{W-x+1}{r-l+1}\right)$

- 証明:
  1. ノード p_x が表す区間は $[x, W]$ と共通部分を持っている
     - そうでないのに部分木が $[x, W]$ を包含するなら、その右の子孫がjump pointerになる
  2. p_x の右部分木の任意のノードが表す区間は $[x, W]$ に包含されている
     - p_x が表す区間の右端が x 以上なため、右部分木の任意の区間も x 以上
  3. p_x の右部分木の区間長総和が $W - x + 1$ 以下で、探索する区間長が $r - l + 1$ なため、探索のイテレーション回数は $O\left(1 + \log \frac{W-x+1}{r-l+1}\right)$

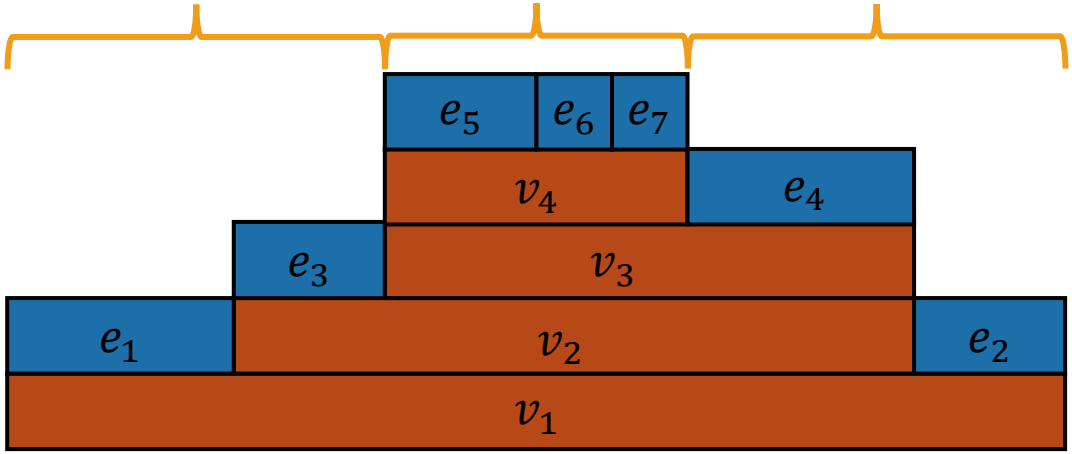
Jump pointerを使った場合の計算量

1. ノード p_x が表す区間は $[x, W]$ と共通部分を持っている
   - ▲ そうでないのに部分木が $[x, W]$ を包含するなら、その右の子孫がjump pointer
2. p_x の右部分木の任意のノードが表す区間は $[x, W]$ に包含されている
   - ▲ p_x が表す区間の右端が x 以上なため、右部分木の任意の区間も x 以上



Jump pointerを使った探索

- IBSTのjump pointerは区間の後半へのジャンプにしか対応していない
  - 今回は「区間の一部分にジャンプしたい」ため、そのまま使えない
- IBSTを3種類保持することで解決
  - Heavy Pathの始点 v_t に直接繋がるLight Edgeを保持するIBST（図中央）
  - $\max I_L(e) \leq \min I_H(v_t)$ であるようなLight Edge e を保持するIBST（図左側）
  - $\max I_H(v_t) \leq \min I_L(e)$ であるようなLight Edge e を保持するIBST（図右側）

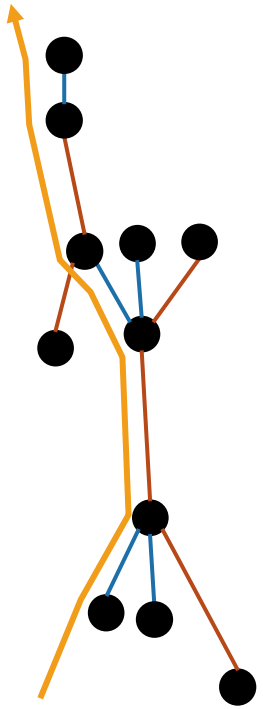


Jump pointerを使った探索

- IBSTを3種類保持する
  - Heavy Pathの始点 v_t に直接繋がるLight Edgeを保持するIBST
    - jump pointerは使わない（もともと探索区間が $I_H(v_t)$ の範囲に収まっている）
  - $\max I_L(e) \leq \min I_H(v_t)$ であるようなLight Edge e を保持するIBST
    - jump pointerで区間の後半にジャンプ
  - $\max I_H(v_t) \leq \min I_L(e)$ であるようなLight Edge e を保持するIBST
    - jump pointerで区間の前半にジャンプ
- 計算量は $O\left(\log \frac{R(\text{遷移元頂点})}{R(\text{遷移先頂点})}\right)$

Heavy Path + IBST手法のまとめ

- Heavy Path→Light Edgeの遷移をIBSTで行う
  - 一回の遷移の計算量は $O\left(\log \frac{R(\text{遷移元頂点})}{R(\text{遷移先頂点})}\right)$
- 上の操作を $O(\log n)$ 回繰り返してパスを得る
- 全体の計算量は $O(\log n)$
  - イテレーション回数は $O(\log n)$
  - 遷移の計算量合計も $O(\log n)$
- よって、$O(\log n)$ 時間でパスを得ることができた



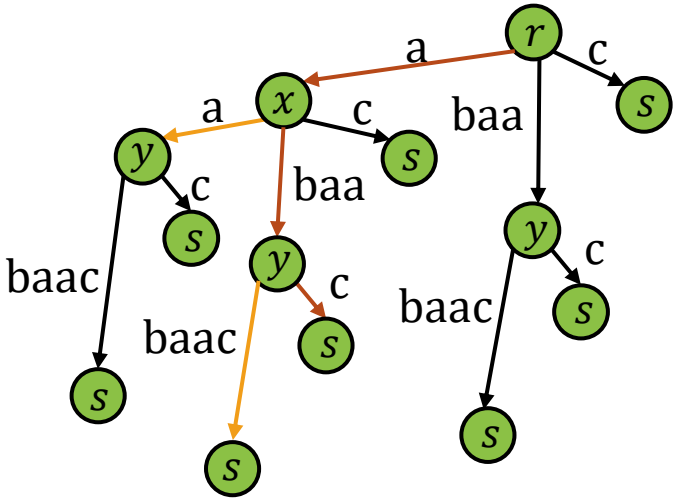
以降、パスから ISA などを求める方法を紹介

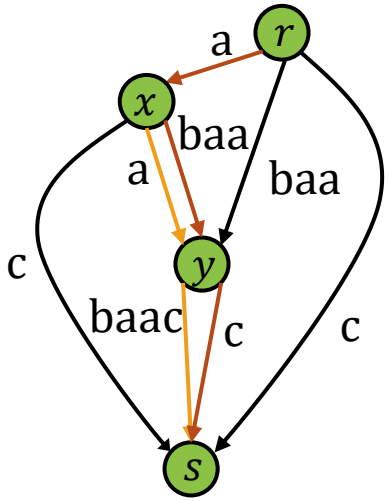
$T[l,n]$ を表すパスからの $\mathrm{ISA}[l]$ の計算（再掲）

$T[l,n]$ より辞書順で小さい接尾辞の数+1

辺 $e=(u,v,c,\ell)$ について、u を始点とするような、辺ラベルが c より小さい辺の集合を $L(e)$ とする

このとき、$\mathrm{ISA}[l]=1+\sum_{e\in P}\sum_{(u,v,c,\ell)\in L(e)} S(v)$ が成り立つ





$T[l,n]$ を表すパスからの $\mathrm{ISA}[l]$ の計算

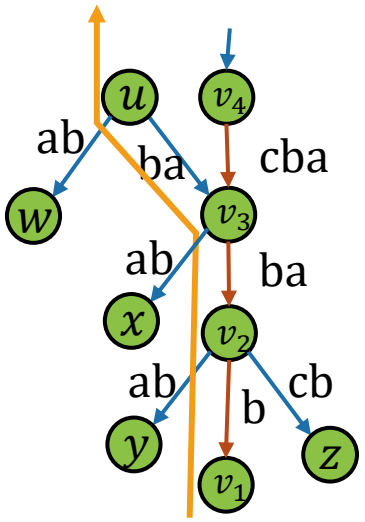
$T[l,n]$ より辞書順で小さい接尾辞の数+1

- $L(e)$: 辺 $e=(u,v,c,\ell)$ について、u を始点とするような、辺ラベルが c より小さい辺の集合

求めたいものは、パス上の全ての辺 e における $\sum_{(u,v,c,\ell)\in L(e)} S(v)$ の総和

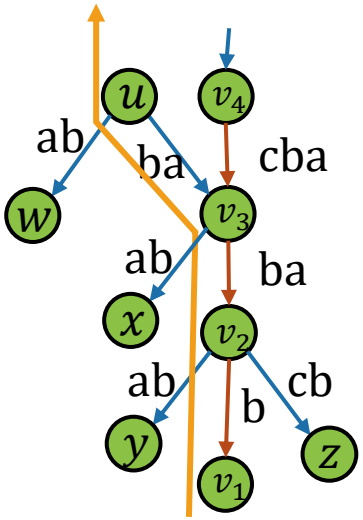
以下を新たに定義:

- $S_H(v_i)=\sum_{j=1}^{i-1}\sum_{(u,v,c,\ell)\in L(e_{j-1})} S(v)$
  - v_i はHeavy Pathに属する頂点
  - e_j は v_{j+1} と v_j を結ぶHeavy Edge
- $S_L(e)=\sum_{(u,v,c,\ell)\in L(e)} S(v)$
  - e はLight Edge



$T[l,n]$ を表すパスからの $\mathrm{ISA}[l]$ の計算

$T[l,n]$ より辞書順で小さい接尾辞の数+1



- $S_H(v_i) = \sum_{j=1}^{i-1} \sum_{(u,v,c,\ell) \in L(e_{j-1})} S(v)$
  - v_i はHeavy Pathに属する頂点
  - e_j は v_{j+1} と v_j を結ぶHeavy Edge
- $S_L(e) = \sum_{(u,v,c,\ell) \in L(e)} S(v)$
  - e はLight Edge

$S_H(v_i)$ と $S_L(e)$ を保存しておくことで、$\mathrm{ISA}[l]$ の暫定値を以下のように更新できる

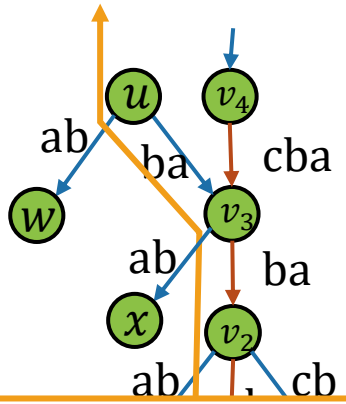
- Heavy Path $v_i - v_j$ をパスに含める場合: $S_H(v_i) - S_H(v_j)$を加算
- Light Edge e をパスに含める場合: $S_L(e)$ を加算

$\mathrm{ISA}[l]$ の暫定値の更新は定数時間で行えるため、$\mathrm{ISA}[l]$ の計算量は $O(\log n)$

$T[l,n]$ を表すパスからの $\mathrm{ISA}[l]$ の計算

$T[l,n]$ より辞書順で小さい接尾辞の数+1

- $S_H(v_i) = \sum_{j=1}^{i-1} \sum_{(u,v,c,\ell)\in L(e_{j-1})} S(v)$
  - v_i はHeavy Pathに属する頂点
  - e_j は v_{j+1} と v_j を結ぶHeavy Edge
- $S_L(e) = \sum_{(u,v,c,\ell)\in L(e)} S(v)$
  - e はLight Edge



$T[l,n]$ に対応する SA 上の区間の計算も同様の方法で高速に行える（詳細略）

$S_H(v_i)$ と $S_L(e)$ を保存しておくことで、$\mathrm{ISA}[l]$ の暫定値を以下のように更新できる

- Heavy Path $v_i - v_j$ をパスに含める場合: $S_H(v_i) - S_H(v_j)$を加算
- Light Edge e をパスに含める場合: $S_L(e)$ を加算

$\mathrm{ISA}[l]$ の暫定値の更新は定数時間で行えるため、$\mathrm{ISA}[l]$ の計算量は $O(\log n)$

補足のまとめ

以下のデータ構造について解説しました

定理（[Belazzougui and Cunial, CPM 2017]+α）

以下の処理を $O(\log n)$ 時間で行える

$O(e)$ 領域のデータ構造が存在する:

1. $\mathrm{SA}[i]$ の計算（次ページで簡単に説明）
2. $\mathrm{ISA}[i]$ の計算
3. $T[i]$ の計算
4. $T[l..r]$ に対応する SA 上の区間の取得

補足の補足: $\mathrm{SA}[i]$ の計算について

1. **root→sinkを結ぶ辞書順 i 番目のパスを得る**

   長さでなく辞書順を指定してのパス取得だが、以下の改変で対応できる

   - 逆向きではなく順方向（root→sink方向）に探索をするようにする
   - 辺をパス長で並べていたところを辞書順で並べるようにする

   Heavy Path+IBSTへの対応もでき、計算量は $O(\log n)$

2. **パスの長さが k のとき、 $n-k+1$ が答え**