単一の連からなる Run-Based Trie による ルール探索の高速化

原田 崇司[1]　田中 賢[1]　三河 賢治[2]

[1] 神奈川大学大学院 理学研究科 理学専攻 情報科学領域

[2] 新潟大学学術情報機構情報基盤センター

2017 年 3 月 13 日

AL・FPAI, 湯布院公民館

本日の内容

研究背景

Run-Based Trie とその探索法

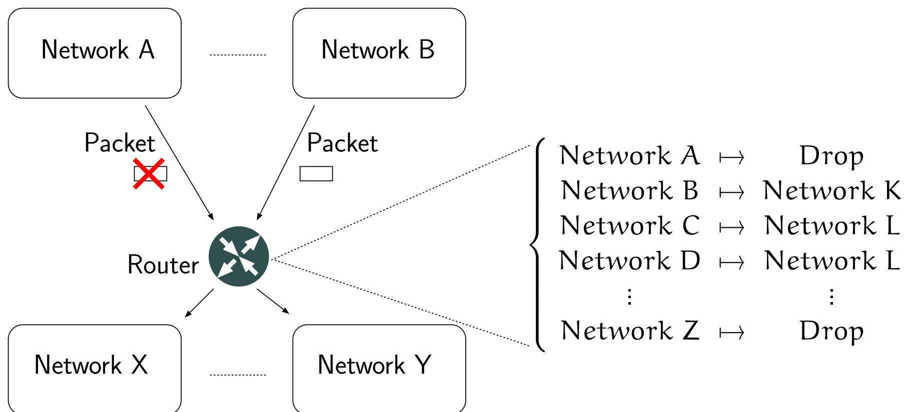
単一の連からなる RBT

ルールリスト列置換

計算機実験

まとめと課題

パケット分類

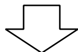


ポリシに従ってパケットを分類

ポリシとルールリスト

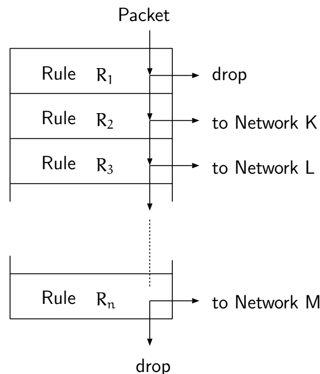
ポリシ : プログラムの仕様
ルールリスト : プログラムの実装

ポリシを満たすルールリストを作成



このルールリストによってパケットを分類

$(R_1, R_2, \ldots, R_{n-1}$ の順でパケットと照合$)$



パケット分類問題

パケット分類問題は右図のようなルールのリストで定義される関数

$$f : \{0,1\}^w \to \{1,2,\ldots,n+1\}$$

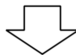
を計算する問題

R_1	$0 * 1 * \cdots 0 *$
R_2	$* 0 0 0 \cdots 1 1$
R_3	$1 * 0 1 \cdots * 1$
R_4	$1 1 * * \cdots 0 1$
$\vdots$	$\vdots$
R_{n-1}	$0 * * 1 \cdots * *$
R_n	$* 1 * * \cdots * *$

'$*$' は don't care，即ち'0' でも'1' でも良い
n はリストの長さ，w はルールの長さ
関数 f は長さ w の $0,1$ の系列をもらって，最初に合致するルールの番号を返す関数

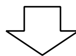
研究の目的

ルールリストのルール数 n は増える一方
線型探索によるパケット分類の探索はルール数 n に依存するので遅い



ルール数 n に依存せずパケット分類を高速に行うためのデータ構造とアルゴリズムが必要

探索時間がルール数 n に依存しないアルゴリズムは少ない



ある制約を課したルールリストに対して，探索時間が $O(w)$ とルール数 n に依存しないアルゴリズムを提案

Run-Based Trie とは

- Run-Based Trie はルールリストで定義された函数

$$f : \{0, 1\}^w \to \{1, \dots, n+1\}$$

  を計算するためのデータ構造

- Run-Based Trie は二分木から成る森,

- Trie といっているが現在は二分木，将来的には函数

$$g : \{a_1, a_2, \dots, a_m\}^w \to \{1, \dots, n+1\}$$

  を計算することを想定

Run-Based Trie

Run-Based Trie は函数 $f : \{0,1\}^w \to \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f の計算には，'$*$' 以外の 0, 1 の部分 (赤色の部分) で充分

0, 1 の部分にどうやって注目しようか？

0, 1 が連続している部分を一塊（連）と見做し，その塊（連）に注目しよう！

R_1	$*\ 0\ *\ 1$
R_2	$0\ 0\ 0\ 0$
R_3	$0\ *\ 0\ 0$
R_4	$0\ *\ 1\ *$
R_5	$*\ 1\ *\ 1$
R_6	$*\ *\ *\ 1$

0, 1 の一塊（連）の例

- R_2 の 1 ビット目から 4 ビット目の 0000
- R_3 の 1 ビット目からの 0 と 3 ビット目からの 00

Run-Based Trie

Run-Based Trie は函数 $f : \{0,1\}^w \to \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f の計算には，'$*$' 以外の 0, 1 の部分 (赤色の部分) で充分

0, 1 の部分にどうやって注目しようか？

0, 1 が連続している部分を一塊（連）と見做し，その塊（連）に注目しよう！

R_1	$* \ 0 \ * \ 1$
R_2	$0 \ 0 \ 0 \ 0$
R_3	$0 \ * \ 0 \ 0$
R_4	$0 \ * \ 1 \ *$
R_5	$* \ 1 \ * \ 1$
R_6	$* \ * \ * \ 1$

0, 1 の一塊（連）の例

- R_2 の 1 ビット目から 4 ビット目の 0000
- R_3 の 1 ビット目からの 0 と 3 ビット目からの 00

Run-Based Trie

Run-Based Trie は函数 $f:\{0,1\}^w \rightarrow \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f の計算には，'$*$' 以外の 0, 1 の部分 (赤色の部分) で充分

0, 1 の部分にどうやって注目しようか？

0, 1 が連続している部分を一塊（連）と見做し，その塊（連）に注目しよう！

R_1	$*\ 0\ *\ 1$
R_2	$0\ 0\ 0\ 0$
R_3	$0\ *\ 0\ 0$
R_4	$0\ *\ 1\ *$
R_5	$*\ 1\ *\ 1$
R_6	$*\ *\ *\ 1$

0, 1 の一塊（連）の例

- R_2 の 1 ビット目から 4 ビット目の 0000
- R_3 の 1 ビット目からの 0 と 3 ビット目からの 00

Run-Based Trie

Run-Based Trie は関数 $f : \{0,1\}^w \rightarrow \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f の計算には，'∗' 以外の 0, 1 の部分 (赤色の部分) で充分

0, 1 の部分にどうやって注目しようか？

0, 1 が連続している部分を一塊（連）と見做し，その塊（連）に注目しよう！

R_1	∗ 0 ∗ 1
R_2	0 0 0 0
R_3	0 ∗ 0 0
R_4	0 ∗ 1 ∗
R_5	∗ 1 ∗ 1
R_6	∗ ∗ ∗ 1

0, 1 の一塊（連）の例

- R_2 の 1 ビット目から 4 ビット目の 0000
- R_3 の 1 ビット目からの 0 と 3 ビット目からの 00

連に注目して探索

Run-Based Trie は関数 $f : \{0,1\}^w \to \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f を計算するために，入力系列 α がどの連に合致するかを探索

α がどの連に合致するかをどうやって調べるか...

それぞれの連 R_i^j の開始位置 s ビット毎に連 R_i^j で Trie 木を w 本構成

w 本のトライを探索

R_1	$* \; 0 \; * \; 1$
R_2	$0 \; 0 \; 0 \; 0$
R_3	$0 \; * \; 0 \; 0$
R_4	$0 \; * \; 1 \; *$
R_5	$* \; 1 \; * \; 1$
R_6	$* \; * \; * \; 1$

連に注目して探索

Run-Based Trie は函数 $f : \{0,1\}^w \to \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f を計算するために，入力系列 α がどの連に合致するかを探索

α がどの連に合致するかをどうやって調べるか...

それぞれの連 R_i^j の開始位置 s ビット毎に連 R_i^j で Trie 木を w 本構成

w 本のトライを探索

R_1	$* \; 0 \; * \; 1$
R_2	$0 \; 0 \; 0 \; 0$
R_3	$0 \; * \; 0 \; 0$
R_4	$0 \; * \; 1 \; *$
R_5	$* \; 1 \; * \; 1$
R_6	$* \; * \; * \; 1$

連に注目して探索

Run-Based Trie は函数 $f : \{0,1\}^w \to \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f を計算するために，入力系列 α がどの連に合致するかを探索

α がどの連に合致するかをどうやって調べるか…

それぞれの連 R_i^j の開始位置 s ビット毎に連 R_i^j で Trie 木を w 本構成

w 本のトライを探索

R_1	$*$ 0 $*$ 1
R_2	0 0 0 0
R_3	0 $*$ 0 0
R_4	0 $*$ 1 $*$
R_5	$*$ 1 $*$ 1
R_6	$*$ $*$ $*$ 1

連に注目して探索

Run-Based Trie は関数 $f : \{0,1\}^w \rightarrow \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f を計算するために，入力系列 α がどの連に合致するかを探索

α がどの連に合致するかをどうやって調べるか...

それぞれの連 R_i^j の開始位置 s ビット毎に連 R_i^j で Trie 木を w 本構成

w 本のトライを探索

R_1	* 0 * 1
R_2	0 0 0 0
R_3	0 * 0 0
R_4	0 * 1 *
R_5	* 1 * 1
R_6	* * * 1

連に注目して探索

Run-Based Trie は函数 $f : \{0, 1\}^w \to \{1, 2, \ldots, n + 1\}$ を計算するためのデータ構造

函数 f を計算するために，入力系列 α がどの連に合致するかを探索

α がどの連に合致するかをどうやって調べるか…

それぞれの連 R_i^j の開始位置 s ビット毎に連 R_i^j で Trie 木を w 本構成

w 本のトライを探索

R_1	$* \; 0 \; * \; 1$
R_2	$0 \; 0 \; 0 \; 0$
R_3	$0 \; * \; 0 \; 0$
R_4	$0 \; * \; 1 \; *$
R_5	$* \; 1 \; * \; 1$
R_6	$* \; * \; * \; 1$

連に注目して探索

Run-Based Trie は関数 $f : \{0,1\}^w \to \{1,2,\ldots,n+1\}$ を計算するためのデータ構造

函数 f を計算するために，入力系列 α がどの連に合致するかを探索

α がどの連に合致するかをどうやって調べるか…

それぞれの連 R_i^j の開始位置 s ビット毎に連 R_i^j で Trie 木を w 本構成

w 本のトライを探索

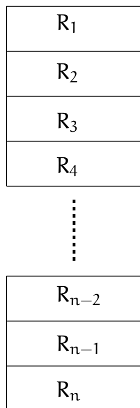
R_1	$* \ 0 * 1$
R_2	$0\ 0\ 0\ 0$
R_3	$0 * 0\ 0$
R_4	$0 * 1 *$
R_5	$* \ 1 * 1$
R_6	$* * * \ 1$

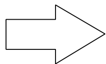
この w 本のトライのことを **Run-Based Trie** と呼ぶ

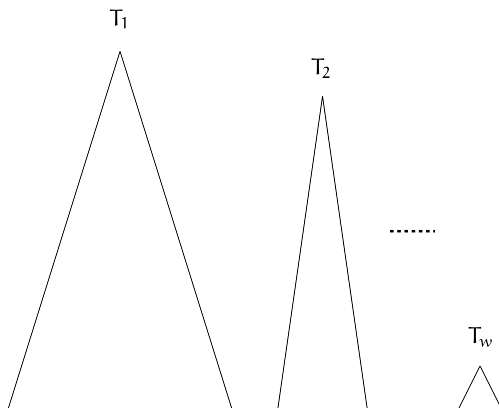
ルールリスト → Run-Based Trie

Rule List

Run-Based Trie







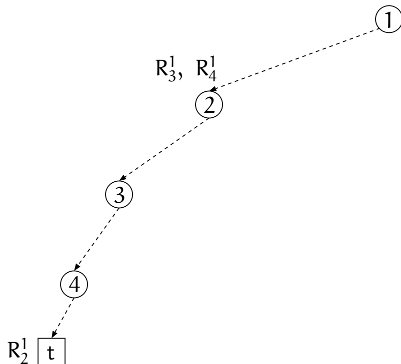
ルールリストから Run-Based Trie（w 本のトライ）を構成

Run-Based Trie 構築

R_1	$* \ 0 * 1$
R_2	$0 \ 0 \ 0 \ 0$
R_3	$0 * 0 \ 0$
R_4	$0 * 1 *$
R_5	$* \ 1 * 1$
R_6	$* * * 1$

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成



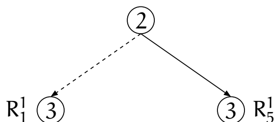
R_1	$*$ 0 $*$ 1
R_2	0 0 0 0
R_3	0 $*$ 0 0
R_4	0 $*$ 1 $*$
R_5	$*$ 1 $*$ 1
R_6	$*$ $*$ $*$ 1

1 ビット目の連で T_1 を構成

R_i^j はルール R_i の先頭から j 番目の連を表現

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成



R_1	∗ 0 ∗ 1
R_2	0 0 0 0
R_3	0 ∗ 0 0
R_4	0 ∗ 1 ∗
R_5	∗ 1 ∗ 1
R_6	∗ ∗ ∗ 1

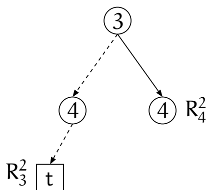
2 ビット目の連で T_2 を構成

R_i^j はルール R_i の先頭から j 番目の連を表現

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成

R_1	* 0 * 1
R_2	0 0 0 0
R_3	0 * 0 0
R_4	0 * 1 *
R_5	* 1 * 1
R_6	* * * 1



3 ビット目の連で T_3 を構成

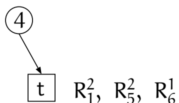
R_i^j はルール R_i の先頭から j 番目の連を表現

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成

R_1	$* \; 0 \; * \; 1$
R_2	$0 \; 0 \; 0 \; 0$
R_3	$0 \; * \; 0 \; 0$
R_4	$0 \; * \; 1 \; *$
R_5	$* \; 1 \; * \; 1$
R_6	$* \; * \; * \; 1$

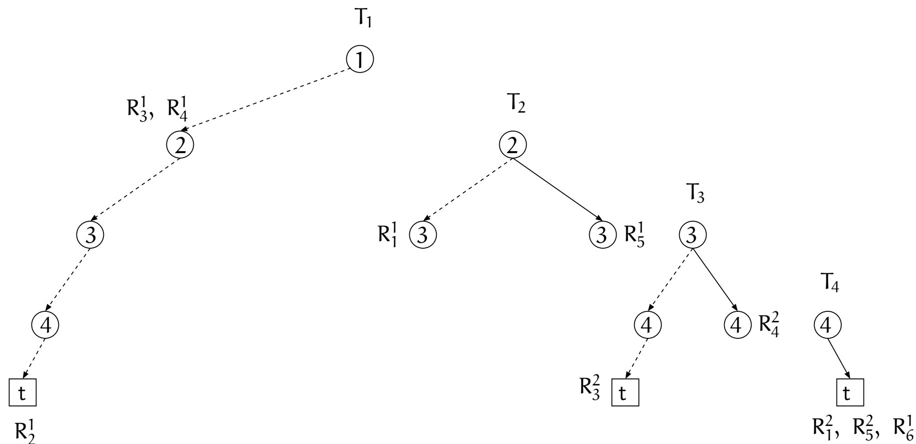
4 ビット目の連で T_4 を構成



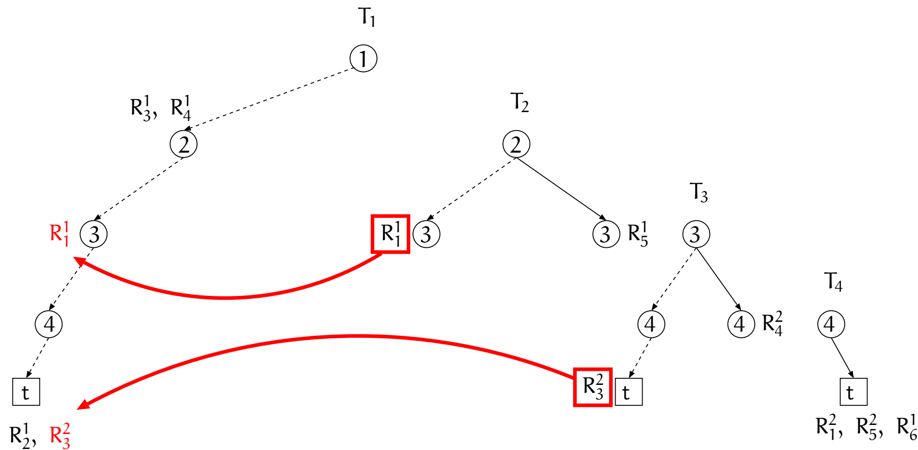
R_i^j はルール R_i の先頭から j 番目の連を表現

Run-Based Trie 構築

前スライドのルールリストから構成される Run-Based Trie は以下

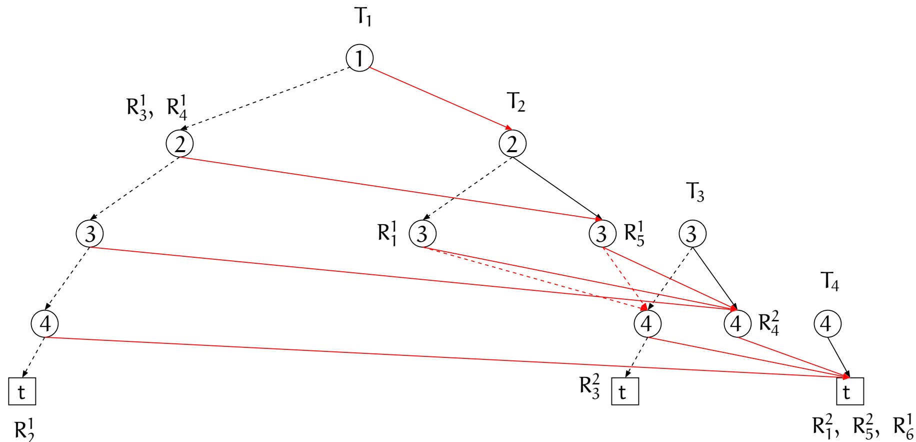


下位トライの連を上位トライへ付与



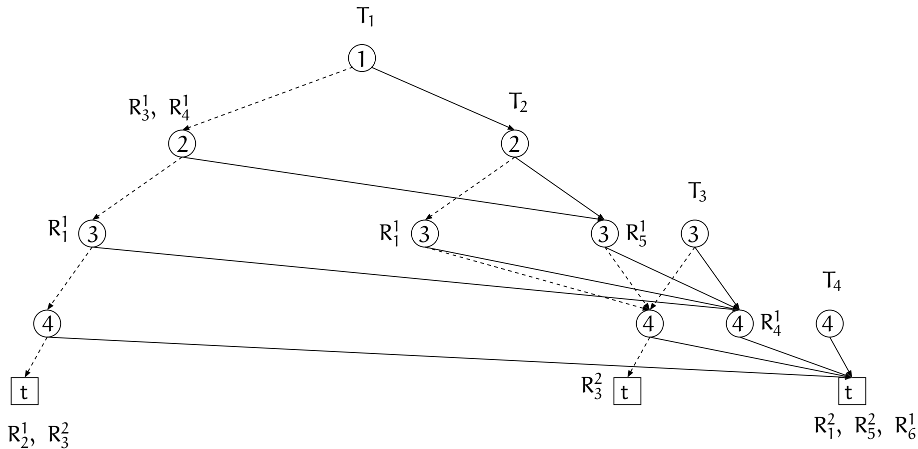
T_1 の 0 節点と T_2 の根，T_1 の 00 節点と T_3 の根を重ねる

上位トライから下位トライへポインタを付与



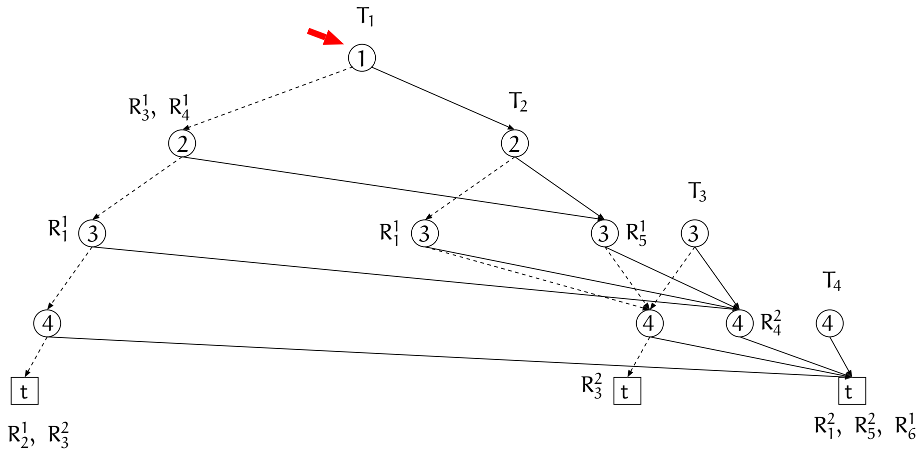
例：T_1 の 00，T_2 の 0, 1 節点から T_3 の 1 節点へポインタ

連とポインタを追加した Run-Based Trie



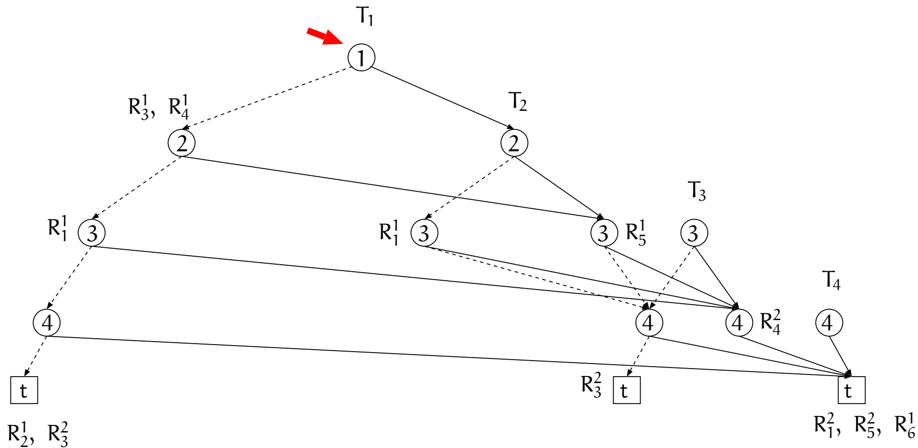
本シート以降では RBT とは，連とポインタを追加したものを指す

Run-Based Trie 探索

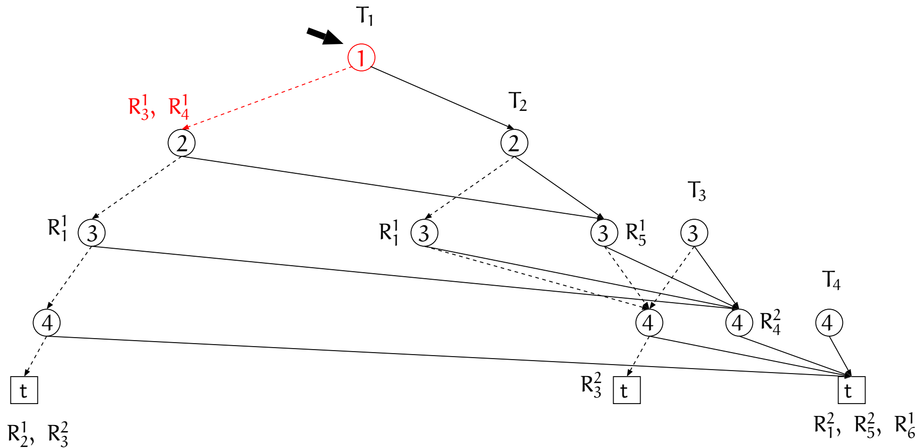


長さ n の配列 A と変数 B を用意し，T_1 の根から探索

Run-Based Trie 探索の例：0011 を探索

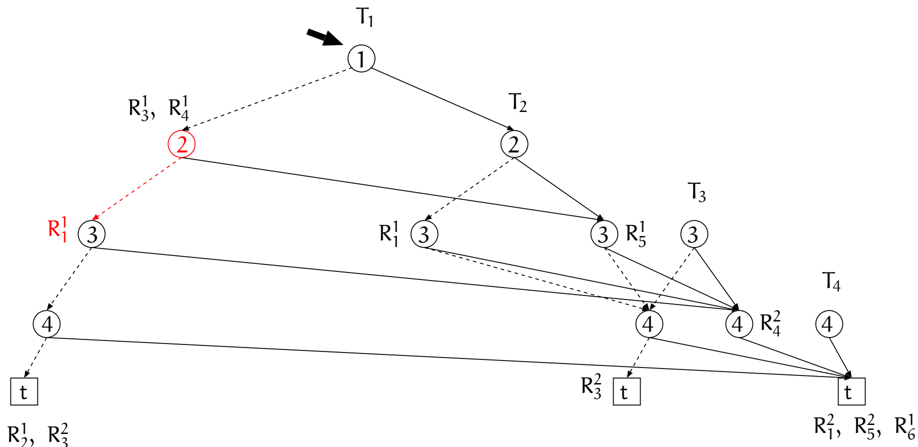


Run-Based Trie 探索の例：0011 を探索

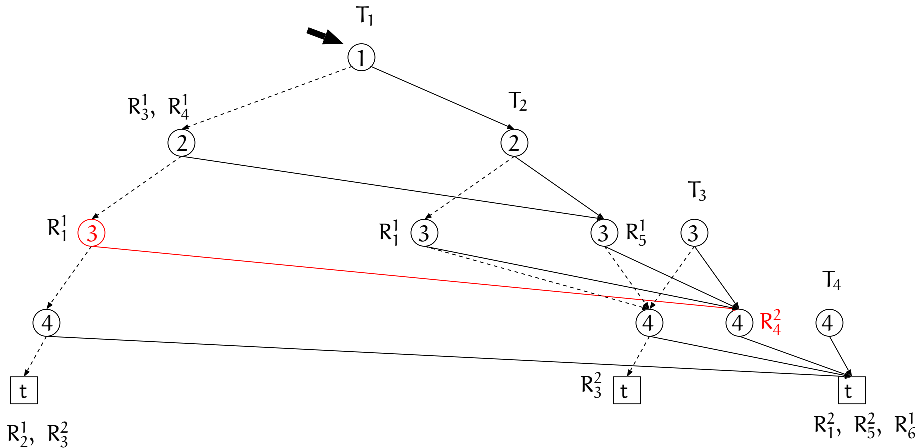


B = 7 A | 0 | 0 | 1 | 1 | 0 | 0 | α = 0011

Run-Based Trie 探索の例：0011 を探索



Run-Based Trie 探索の例：0011 を探索

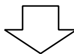


Run-Based Trie 探索の例：0011 を探索

$B = 1$ A | 2 | 0 | 1 | 2 | 1 | 1 | $\alpha = 0011$

RBT 探索の計算量

- 入力系列 α で RBT を辿る計算量 $O(w)$
- α に合致した連の照合，最優先ルールとの比較の計算量 $O(nw)$
  (連 R_i^j の数は高々 $nw/2$)

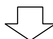


RBT 探索の時間計算量は $O(nw)$

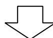
探索時間が n に依存しないアルゴリズムが欲しい

RBT 探索の計算量

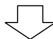
一つのルールが複数の連を持つ可能性がある



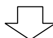
連に合致する度に照合を行わなければならないので探索時間 n に依存



各ルールの連が一つだけならば，連に合致する度に照合せずに済む？



RBT 探索の時間計算量が n に依存せず $O(w)$!



ルールリストを行列と見做し，列を交換して各ルールの連を一つに！

単一の連からなる Run-Based Trie

1. 与えられたルールリストを単一の連だけから成るように列を置換
(置換できない場合等については後で)
2. RBT を構築
必要ならば T_4 に終端節点を追加
3. 単一の連という特徴を生かして簡約

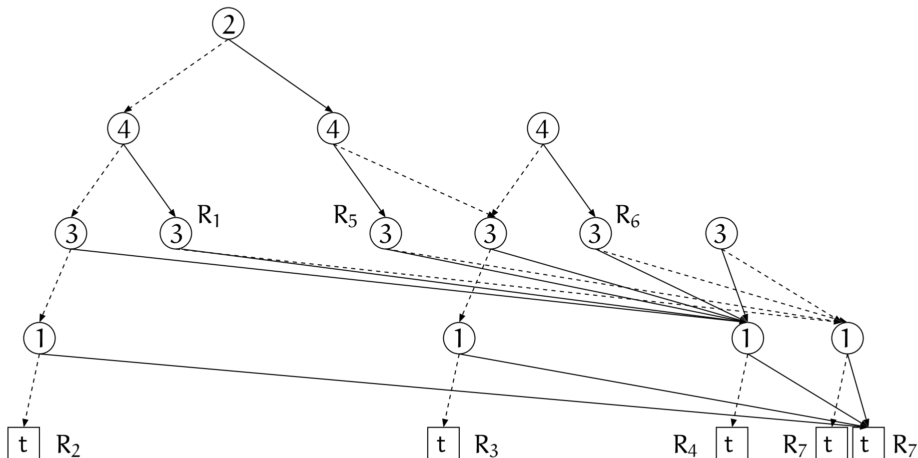
1. は提案手法に含まれておらず，今後の課題

R_1	$* \ 0 \ * \ 1$
R_2	$0 \ 0 \ 0 \ 0$
R_3	$0 \ * \ 0 \ 0$
R_4	$0 \ * \ 1 \ *$
R_5	$* \ 1 \ * \ 1$
R_6	$* \ * \ * \ 1$

$\downarrow \quad (4\ 1\ 3\ 2)$

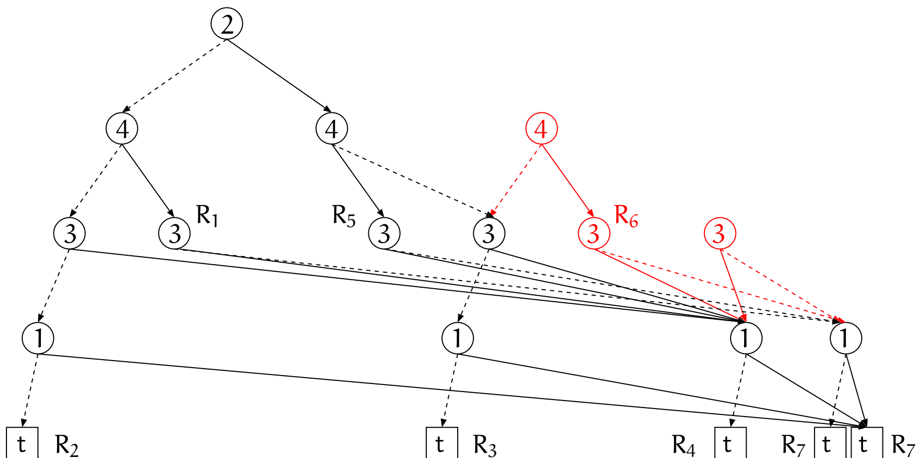
R_1	$0 \ 1 \ * \ *$
R_2	$0 \ 0 \ 0 \ 0$
R_3	$* \ 0 \ 0 \ 0$
R_4	$* \ * \ 1 \ 0$
R_5	$1 \ 1 \ * \ *$
R_6	$* \ 1 \ * \ *$

単一の連から成る RBT の簡約



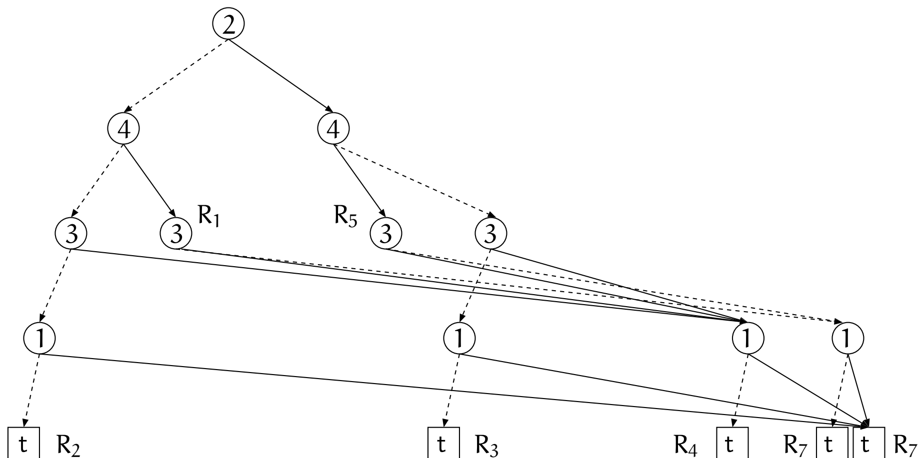
T_4 に終端節点を追加

単一の連から成る RBT の簡約



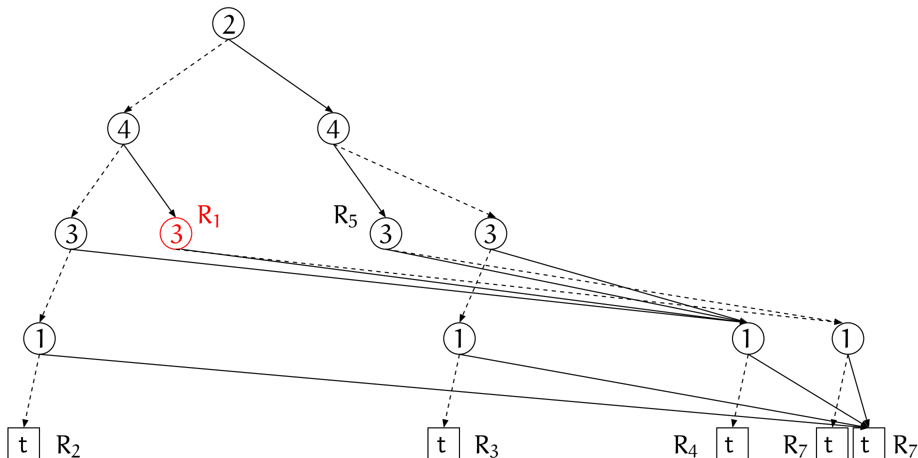
T_1 の根節点から到達不可能な節点・枝を削除

単一の連から成る RBT の簡約



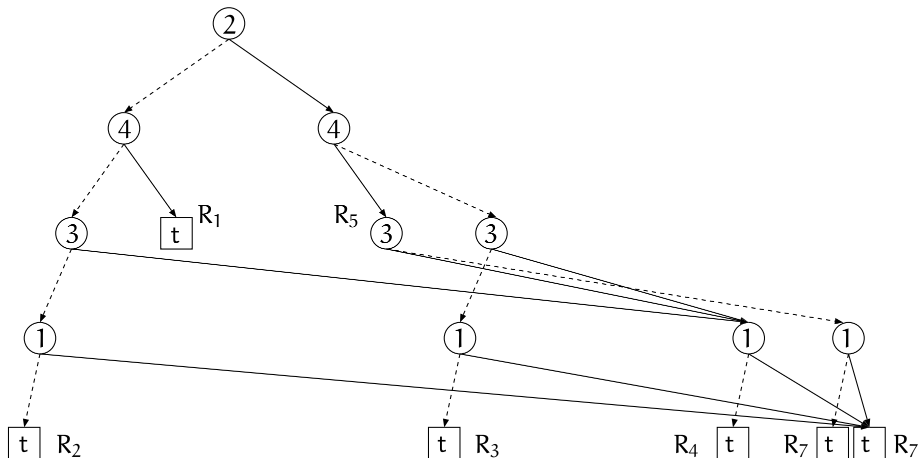
T_1 の根節点から到達不可能な節点・枝を削除

単一の連から成る RBT の簡約



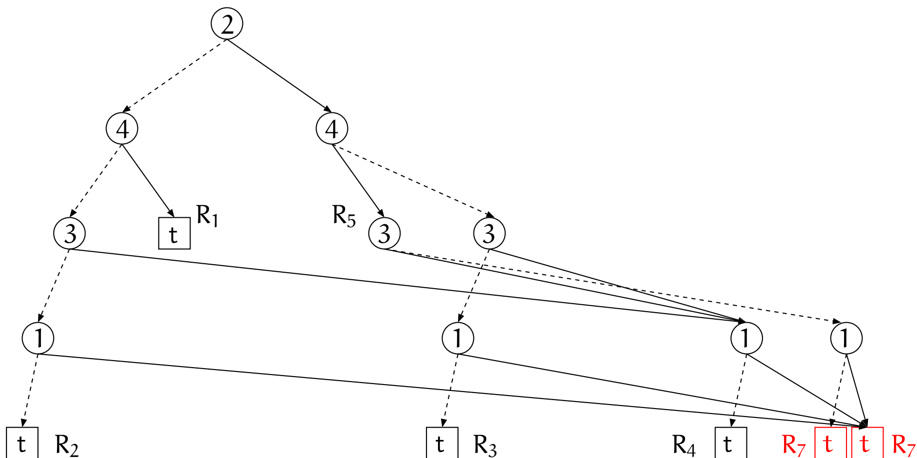
R_i を持ち，子孫に R_j $(j < i)$ を持たない節点を終端節点に

単一の連から成る RBT の簡約



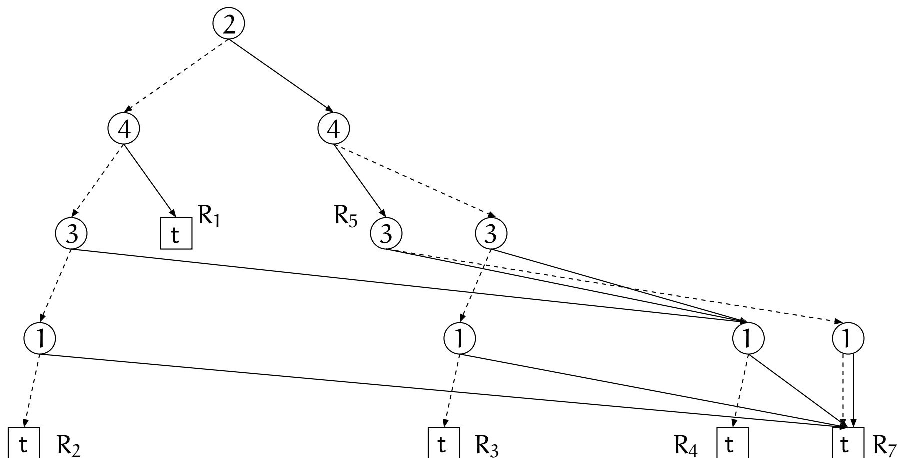
R_i を持ち，子孫に R_j $(j < i)$ を持たない節点を終端節点に

単一の連から成る RBT の簡約



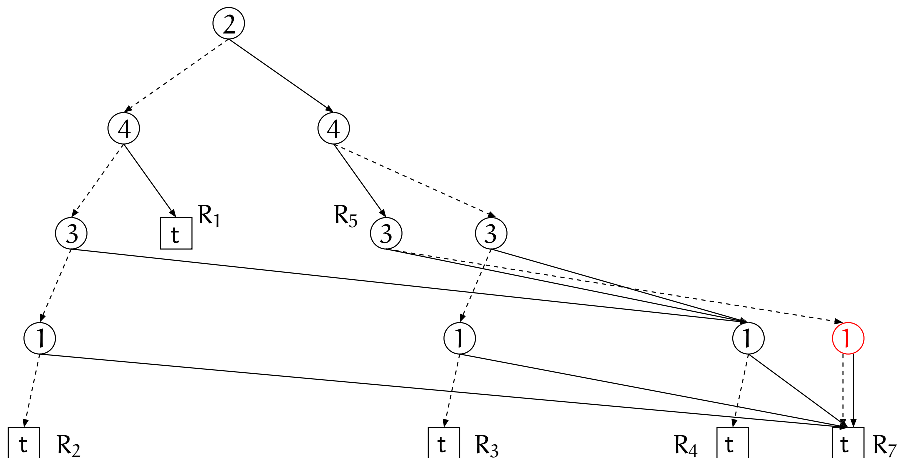
0 枝，1 枝の先が同じ節点 u と v を共有

単一の連から成る RBT の簡約



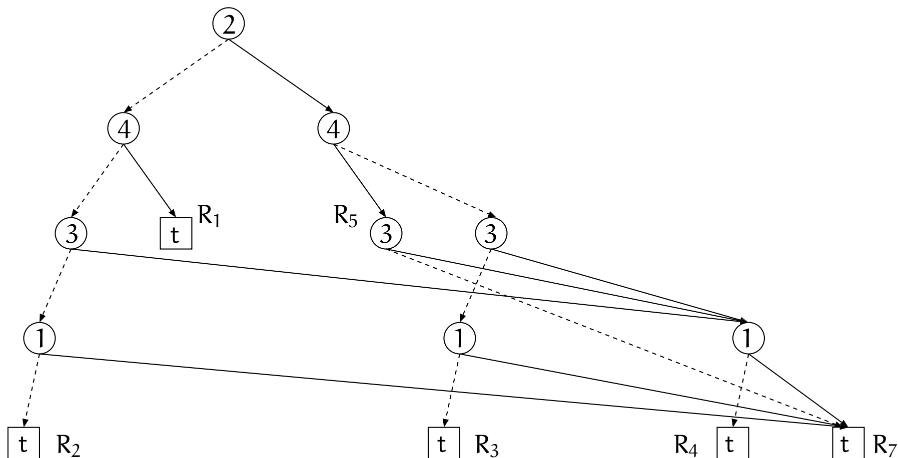
0 枝，1 枝の先が同じ節点 u と v を共有

単一の連から成る RBT の簡約



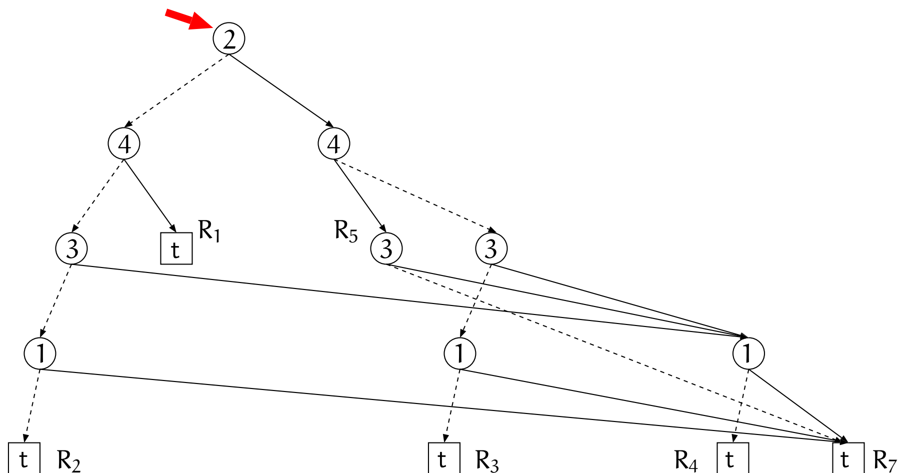
0 枝，1 枝の先が同じ冗長節点を削除

単一の連から成る RBT の簡約



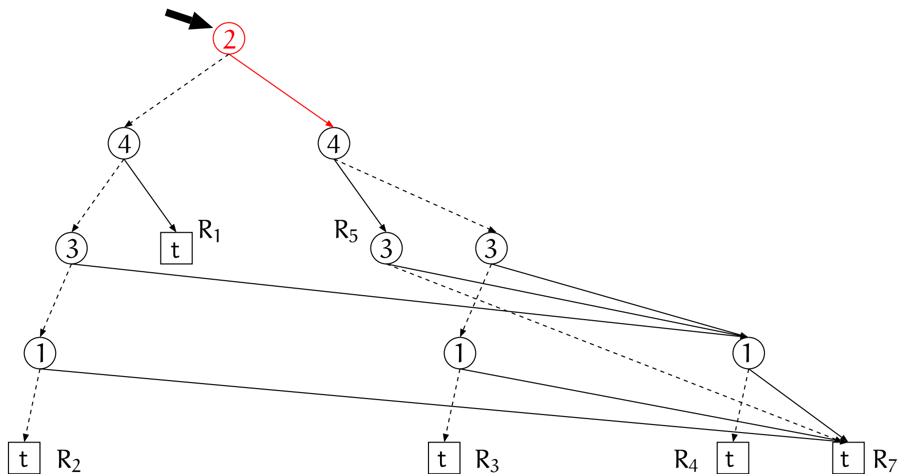
簡約終了

単一の連からなる RBT の探索



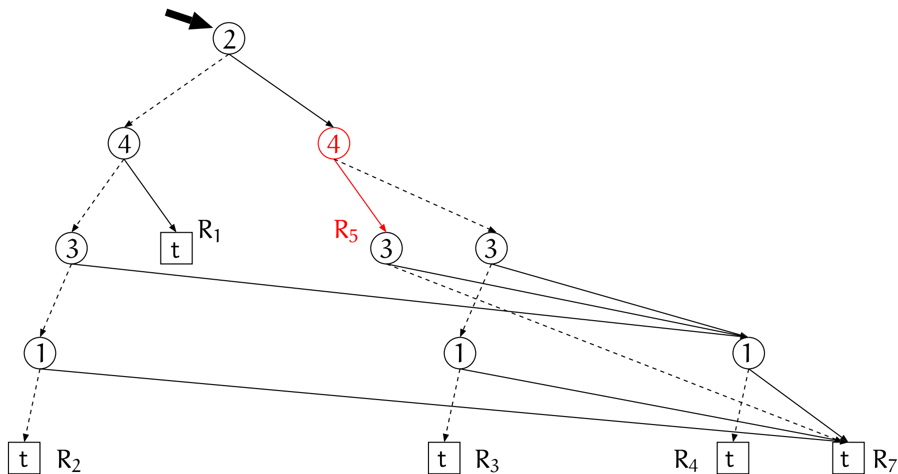
変数 B を用意して T_1 の根から終端節点に到達するまで探索

単一の連からなる RBT の探索



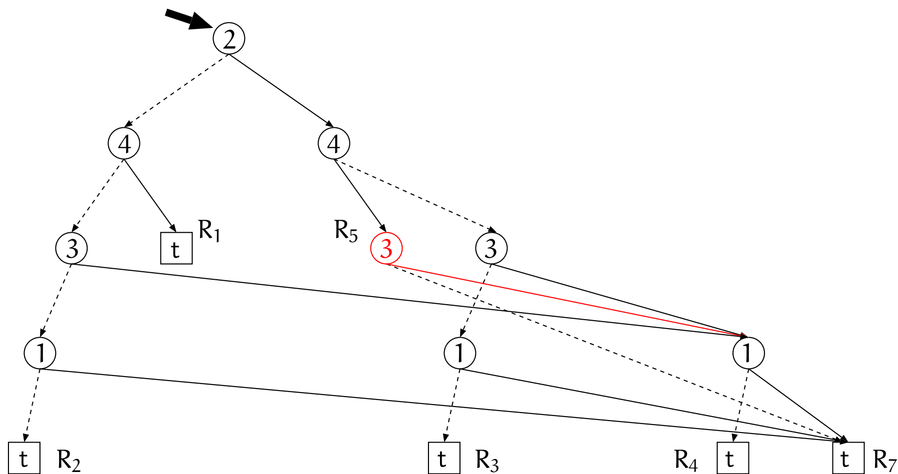
$B = 7 \qquad \alpha = 0111$

単一の連からなる RBT の探索



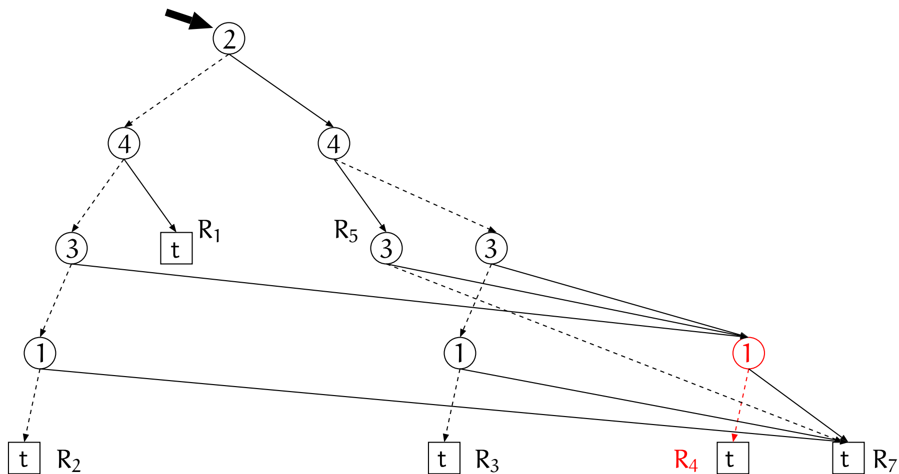
$B = 5 \qquad \alpha = 0111$

単一の連からなる RBT の探索



$B = 5 \qquad \alpha = 0111$

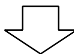
単一の連からなる RBT の探索



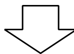
$B = 4 \qquad \alpha = 0111$

単一の連からなる RBT 探索の時間計算量

- T_1 の根から終端節点までの長さは高々w
- 各節点にルールは高々一つ



照合回数は $O(w)$ で，系列で RBT を辿る計算量も $O(w)$

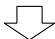


単一の連からなる RBT 探索の時間計算量は n に依存せず $O(w)$

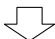
ルールリスト分解・列置換

提案手法は単一の連からなるルールにのみ適用可

右ルールリストのように，単一の連からなるルールへ置換不可能なものが...



与えられたルールリスト $\mathbf{R}$ を複数のルールリスト $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_k$ へ分割，各ルールリストに列置換，RBT の構築



$O(kw)$ でパケット分類可能

	1	2	3	4	5	6
R_1	0	0	*	0	*	0
R_2	*	0	0	*	0	*
R_3	0	*	*	0	*	0
R_4	0	0	0	*	*	*
R_5	*	*	0	*	0	*
R_6	*	0	*	*	0	*
R_7	0	*	*	*	*	0
R_8	0	*	0	0	0	*

ルールリスト分解・列置換の例

RL	1 2 3 4 5 6
R_1	0 0 * 0 * 0
R_2	* 0 0 * 0 *
R_3	0 * * 0 * 0
R_4	0 0 0 * * *
R_5	* * 0 * 0 *
R_6	* 0 * * 0 *
R_7	0 * * * * 0
R_8	0 * 0 0 0 *

RL_1	1 2 3 4 5 6
R_1	0 0 * 0 * 0
R_3	0 * * 0 * 0
R_4	0 0 0 * * *
R_5	* * 0 * 0 *

RL_1'	5 3 2 1 4 6
R_1	* * 0 0 0 0
R_3	* * * 0 0 0
R_4	* 0 0 0 * *
R_5	0 0 * * * *

RL_2	1 2 3 4 5 6
R_2	* 0 0 * 0 *
R_6	* 0 * * 0 *
R_7	0 * * * * 0
R_8	0 * 0 0 0 *

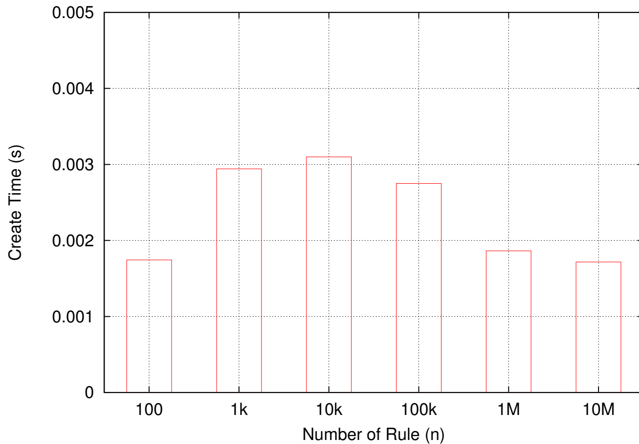
RL_2'	2 5 3 4 1 6
R_2	0 0 0 * * *
R_6	0 0 * * * *
R_7	* * * * 0 0
R_8	* 0 0 0 0 *

実験環境

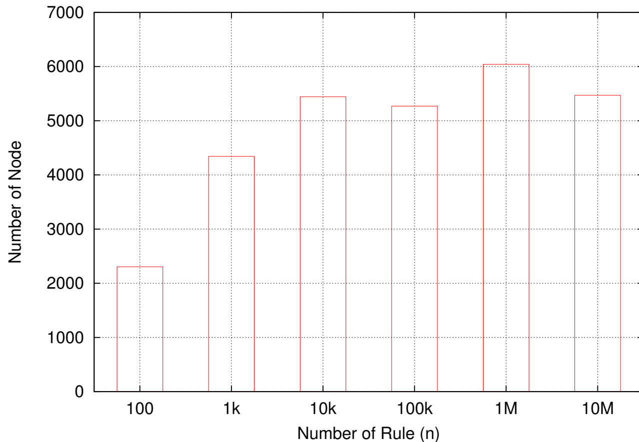
OS	:	CentOS Release 6.2
CPU	:	Intel Core i7-980X 3.33 GHz
主記憶容量	:	24GB
実装言語	:	C++
コンパイラ	:	gcc version 4.8.2

- ルール長 120，ルール数が百 ~ 一千万の単一の連からなるルールリストをランダムに生成
- ヘッダ数 1000 のヘッダリストを各ルールリストに対して作成
- 構築時間 (s)，節点数，探索時間 (s)，照合回数を計測

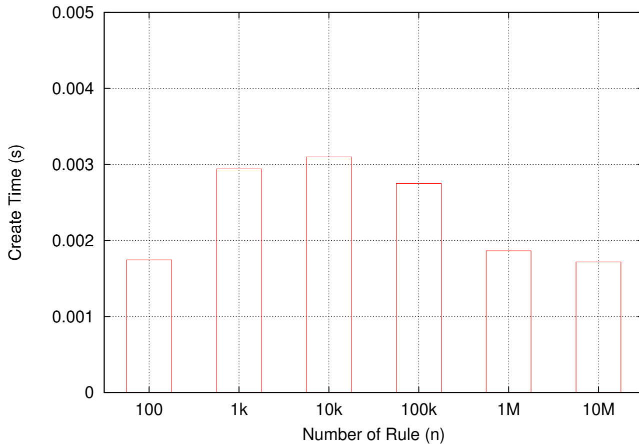
実験結果：　構築時間（秒）



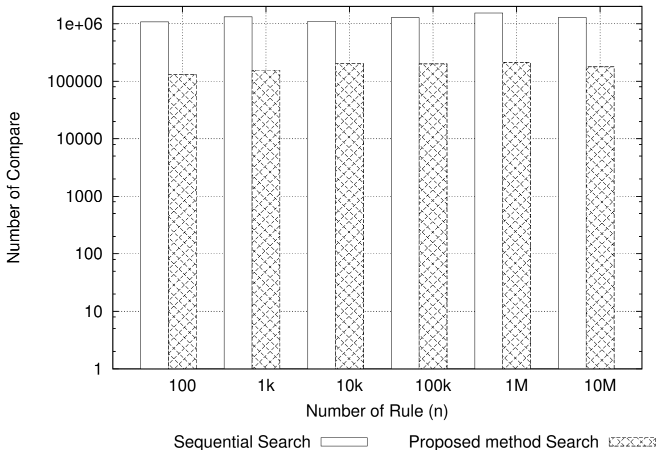
実験結果： 節点数



実験結果： 探索時間（秒）



実験結果： 照合回数



まとめと今後の課題

- 与えられたルールリストの各ルールの連の数が一つだけならば，探索時間が $O(w)$ となるデータ構造と探索法を提案
- ルールリスト分解・列交換問題に対するアルゴリズムが必要