ポインタ付与による Run-Based Trie 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

2016 年 11 月 24 日
CAS・MSS・AL, 神戸情報大学院大学

本日の内容

問題背景

Run-Based Trie とその探索法

Pointed Run-Based Trie（提案手法）

計算機実験

まとめと今後の課題

パケット分類問題

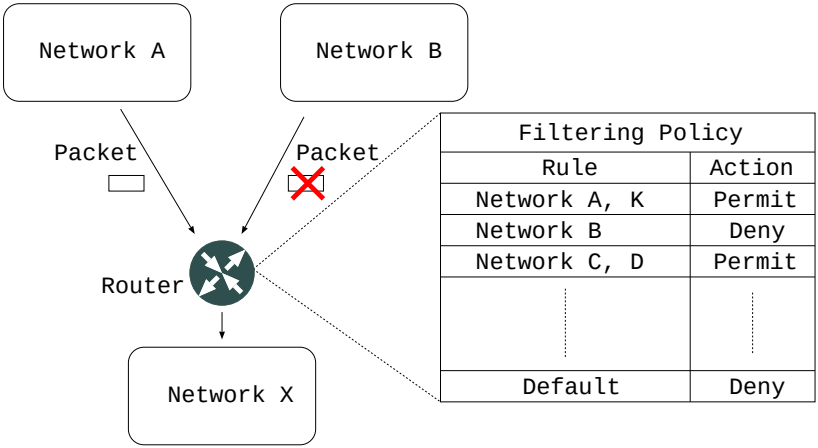


Figure : 入ってくるパケットをポリシーに従ってルータで分類

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た関数

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

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 でも良いことを表す.
n はリストの長さ，w はルールの長さを表す.
函数 f は長さ w の 0,1 の系列をもらって，初めて合致するルールの番号を返す函数.

研究の目的

ルールリストを線型探索してパケット分類するのは遅い

パケット分類問題を高速に解く (f を高速に計算する) ためのデータ構造とアルゴリズムを研究

発表者は，その中で特に Run-Based Trie を用いたアルゴリズムを研究

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 を計算するためには，'$*$' 以外の 0,1 の部分 (赤色の部分) にだけ注目すれば良い.

⇩

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	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$

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 を計算するためには，'$*$' 以外の 0,1 の部分 (赤色の部分) にだけ注目すれば良い.

⇩

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	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$

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 を計算するためには，'$*$' 以外の 0,1 の部分 (赤色の部分) にだけ注目すれば良い.

⇩

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	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$

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 を計算するためには，'$*$' 以外の 0,1 の部分 (赤色の部分) にだけ注目すれば良い.

⇩

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	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$

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 を計算するためには，入力系列 α がどの連に合致するかを調べれば良い．

⇩

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 α で探索

R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$

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 を計算するためには，入力系列 α がどの連に合致するかを調べれば良い．

⇩

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 α で探索

R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$

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 を計算するためには，入力系列 α がどの連に合致するかを調べれば良い．

⇩

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 α で探索

R_1	$*\ 0\ *\ 1$
R_2	$0\ 0\ 0\ 0$
R_3	$0\ *\ 0\ 0$
R_4	$0\ *\ 1\ *$
R_5	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$

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 を計算するためには，入力系列 α がどの連に合致するかを調べれば良い．

⇩

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 α で探索

R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$

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 を計算するためには，入力系列 α がどの連に合致するかを調べれば良い．

⇩

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 α で探索

R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$

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 を計算するためには，入力系列 α がどの連に合致するかを調べれば良い．

⇩

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 α で探索

R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$

この w 本のトライのことを **Run-Based Trie** と呼ぶ

Run-Based Trie 構築

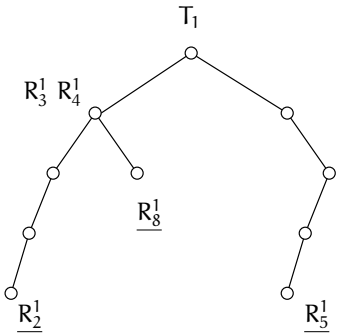
右のルールリストから Run-Based Trie を構成する．

R_1	$*\,0\,*\,1$
R_2	$0\,0\,0\,0$
R_3	$0\,*\,0\,0$
R_4	$0\,*\,1\,*$
R_5	$1\,1\,0\,0$
R_6	$*\,0\,1\,*$
R_7	$*\,*\,1\,0$
R_8	$0\,1\,*\,*$
R_9	$*\,1\,1\,*$

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成する.
1 ビット目からの連で T_1 を構成

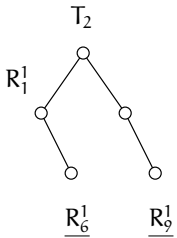
R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$



R_i^j はルール R_i の先頭から j 番目の連を表現.
下線が引いてある連 R_i^j はそのルール R_i の最後の連であることを示す.

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成する．
2 ビット目からの連で T_2 を構成

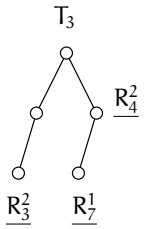


R_1	$*\ 0\ *\ 1$
R_2	$0\ 0\ 0\ 0$
R_3	$0\ *\ 0\ 0$
R_4	$0\ *\ 1\ *$
R_5	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$

R_i^j はルール R_i の先頭から j 番目の連を表現．
下線が引いてある連 R_i^j はそのルール R_i の最後の連であることを示す．

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成する．
3 ビット目からの連で T_3 を構成

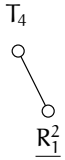


R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$

R_i^j はルール R_i の先頭から j 番目の連を表現．
下線が引いてある連 R_i^j はそのルール R_i の最後の連であることを示す．

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成する．
4 ビット目からの連で T_4 を構成

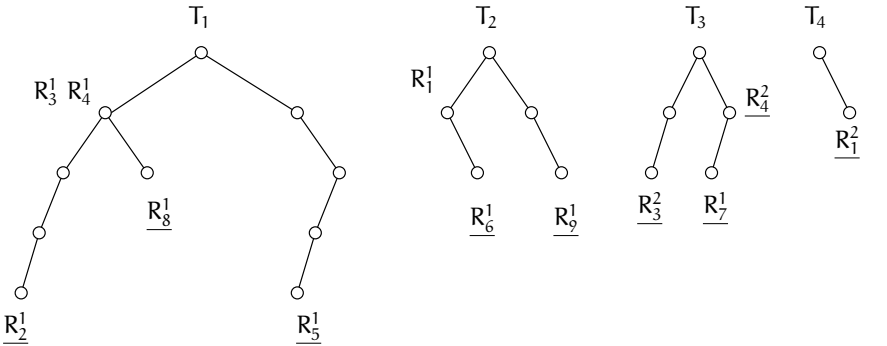


R_1	∗ 0 ∗ 1
R_2	0 0 0 0
R_3	0 ∗ 0 0
R_4	0 ∗ 1 ∗
R_5	1 1 0 0
R_6	∗ 0 1 ∗
R_7	∗ ∗ 1 0
R_8	0 1 ∗ ∗
R_9	∗ 1 1 ∗

R_i^j はルール R_i の先頭から j 番目の連を表現．
下線が引いてある連 R_i^j はそのルール R_i の最後の連であることを示す．

Run-Based Trie の例

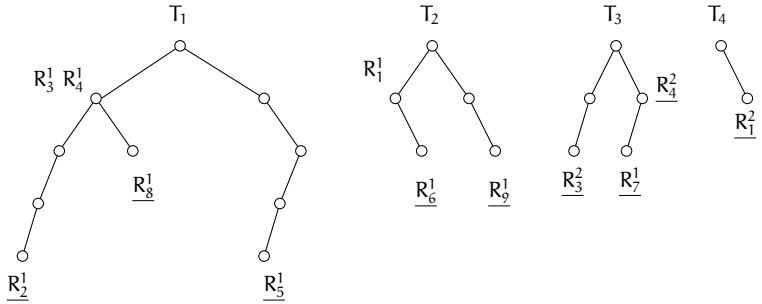
前スライドのルールリストから構成される Run-Based Trie は以下



R_i^j はルール R_i の先頭から j 番目の連を表現.
下線が引いてある連 $\underline{R_i^j}$ はそのルール R_i の最後の連であることを示す.

Run-Based Trie の探索 (Simple Search)

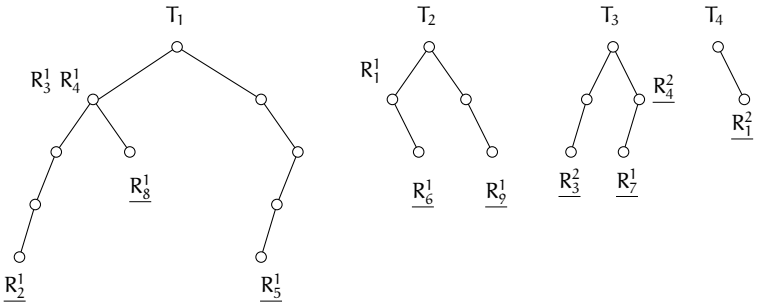
トライ $T_1, T_2, \ldots, T_w$ を探索し，全ての連が合致するルールを調査.



長さ n(ルール数) の配列 A[n] を用意：Run-Based Trie を探索中に各ルールの連の合致状況を保持
変数 B を用意：Run-Based Trie 探索途中での最優先ルール候補を保持

Run-Based Trie の探索 (Simple Search) の例

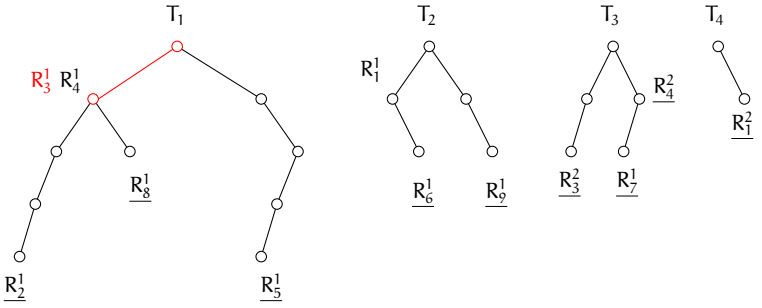
Run-Based Trie 探索の例として，f(0110) を求める．
$B := 10(= 9 + 1), \forall i\ A[i] := 0$ と初期化



B	A	1	2	3	4	5	6	7	8	9
10		0	0	0	0	0	0	0	0	0

Run-Based Trie の探索 (Simple Search) の例

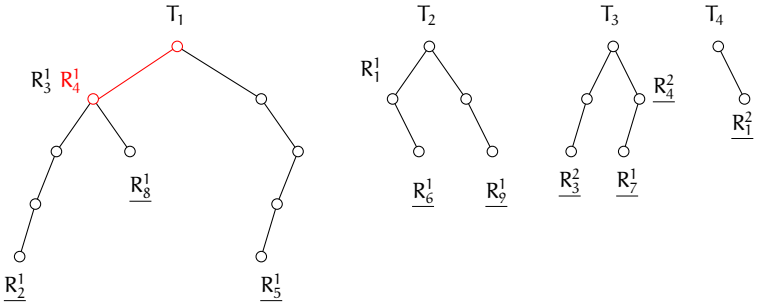
Run-Based Trie 探索の例として，f(0110) を求める．
0110 で T_1 を探索



B	A	1	2	3	4	5	6	7	8	9
10		0	0	1	0	0	0	0	0	0

Run-Based Trie の探索 (Simple Search) の例

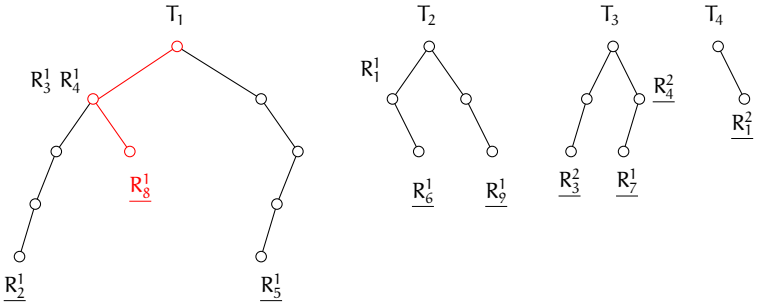
Run-Based Trie 探索の例として，f(0110) を求める．
0110 で T_1 を探索



B	A	1	2	3	4	5	6	7	8	9
10		0	0	1	1	0	0	0	0	0

Run-Based Trie の探索 (Simple Search) の例

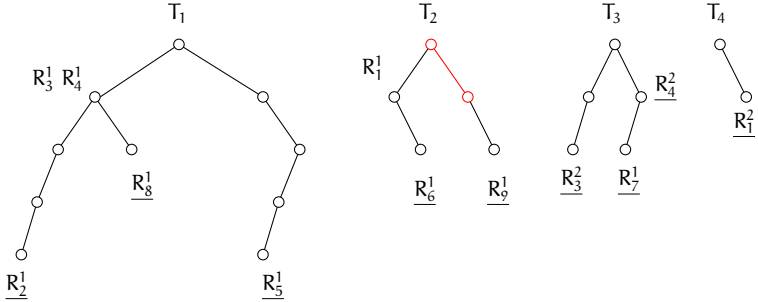
Run-Based Trie 探索の例として，f(0110) を求める．
0110 で T_1 を探索



B	A	1	2	3	4	5	6	7	8	9
8		0	0	1	1	0	0	0	1	0

Run-Based Trie の探索 (Simple Search) の例

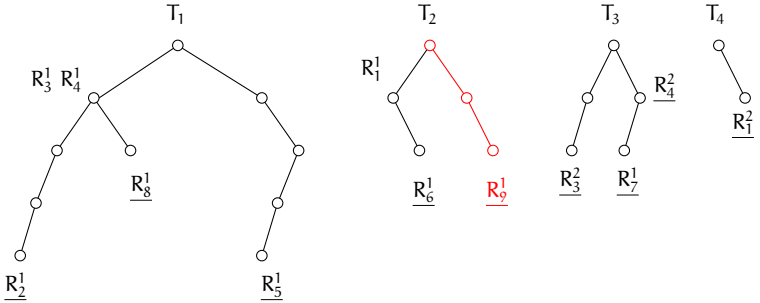
Run-Based Trie 探索の例として，f(0110) を求める．
110 で T_2 を探索



B	A	1	2	3	4	5	6	7	8	9
8		0	0	1	1	0	0	0	1	0

Run-Based Trie の探索 (Simple Search) の例

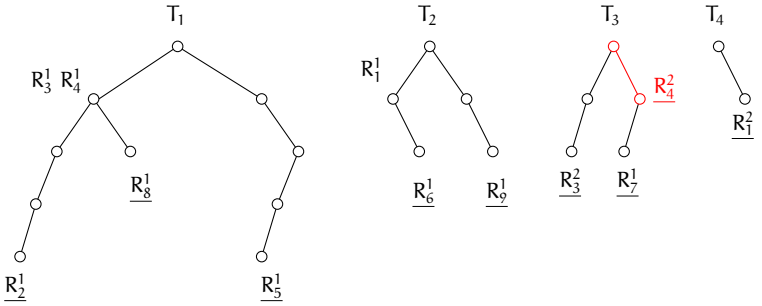
Run-Based Trie 探索の例として，f(0110) を求める．
110 で T_2 を探索



B	A	1	2	3	4	5	6	7	8	9
8		0	0	1	1	0	0	0	1	1

Run-Based Trie の探索 (Simple Search) の例

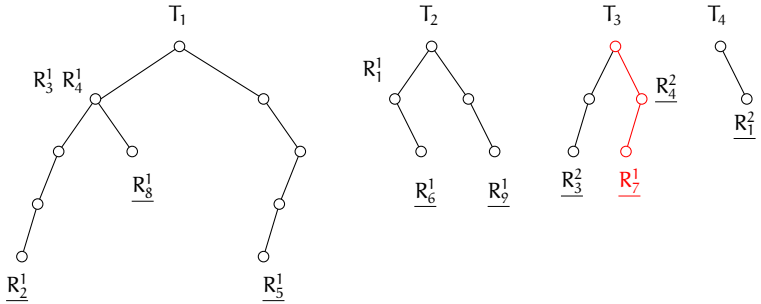
Run-Based Trie 探索の例として，f(0110) を求める．
10 で T_3 を探索



B	A	1	2	3	4	5	6	7	8	9
4		0	0	1	2	0	0	0	1	1

Run-Based Trie の探索 (Simple Search) の例

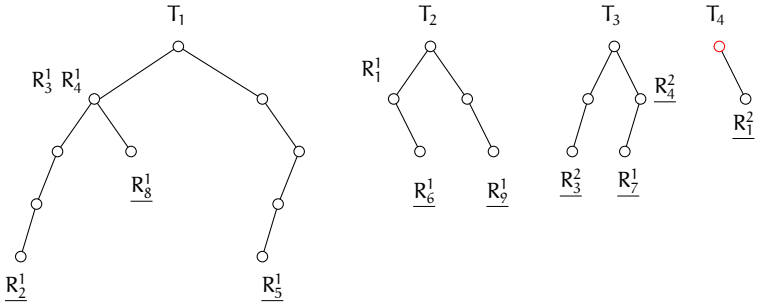
Run-Based Trie 探索の例として，f(0110) を求める．
10 で T_3 を探索



B	A	1	2	3	4	5	6	7	8	9
4		0	0	1	2	0	0	1	1	1

Run-Based Trie の探索 (Simple Search) の例

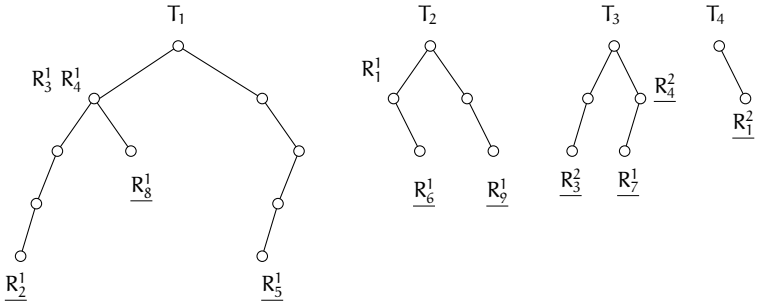
Run-Based Trie 探索の例として，f(0110) を求める．
0 で T_4 を探索



B	A	1	2	3	4	5	6	7	8	9
4		0	0	1	2	0	0	1	1	1

Run-Based Trie の探索 (Simple Search) の例

Run-Based Trie 探索の例として，f(0110) を求める．
探索終了：B = 4 なので f(0110) = 4



	B	A	1	2	3	4	5	6	7	8	9
最優先ルール →	4		0	0	1	2	0	0	1	1	1

Simple Search の問題点

- 入力系列を高々w^2 回参照する
  入力系列に対する参照回数は w 回にできるのでは？

- 探索時間計算量が $\mathrm{O}(\mathrm{n}w + w^2)$ とルール数 n に依存

Simple Search の問題点

- 入力系列を高々w^2 回参照する
  入力系列に対する参照回数は w 回にできるのでは？

  ⇨ 連を複製し，T_i の節点から T_j の節点へポインタを張る（本日提案する手法）

- 探索時間計算量が $O(nw + w^2)$ とルール数 n に依存

Simple Search の問題点

- 入力系列を高々w^2 回参照する
  入力系列に対する参照回数は w 回にできるのでは？

  ⟹ 連を複製し，T_i の節点から T_j の節点へポインタを張る（本日提案する手法）

  ⟹ 探索時間計算量が $O(nw + w^2)$ から $O(nw)$

- 探索時間計算量が $O(nw + w^2)$ とルール数 n に依存

Simple Search の問題点

- 入力系列を高々w^2 回参照する
  入力系列に対する参照回数は w 回にできるのでは？

  ⟹ 連を複製し，T_i の節点から T_j の節点へポインタを張る（本日提案する手法）

  ⟹ 探索時間計算量が $O(nw + w^2)$ から $O(nw)$

- 探索時間計算量が $O(nw + w^2)$ とルール数 n に依存

Simple Search の問題点

- 入力系列を高々w^2 回参照する
  入力系列に対する参照回数は w 回にできるのでは？

  ⟹ 連を複製し，T_i の節点から T_j の節点へポインタを張る（本日提案する手法）

  ⟹ 探索時間計算量が $O(nw + w^2)$ から $O(nw)$

- 探索時間計算量が $O(nw + w^2)$ とルール数 n に依存

  ⟹ Run-Based Trie から決定木を構築（提案済み・改良中）

Simple Search の問題点

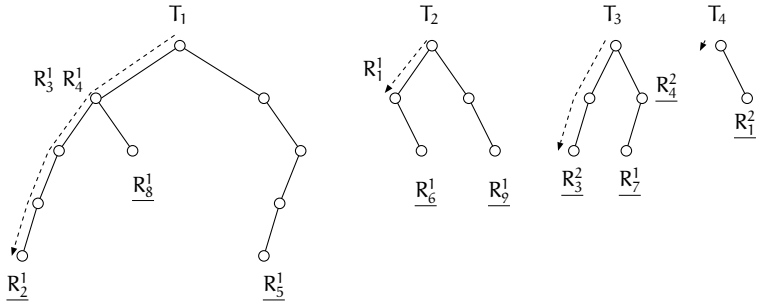
- 入力系列を高々w^2 回参照する
  入力系列に対する参照回数は w 回にできるのでは？

  ⟹ 連を複製し，T_i の節点から T_j の節点へポインタを張る（本日提案する手法）

  ⟹ 探索時間計算量が $O(nw + w^2)$ から $O(nw)$

- 探索時間計算量が $O(nw + w^2)$ とルール数 n に依存
  ⟹ Run-Based Trie から決定木を構築（提案済み・改良中）
  ⟹ 今後の最重要課題で提案中の決定木とは別手法を模索

Simple Search において $\frac{w(w+1)}{2}$ 回参照は参照しすぎ？の例

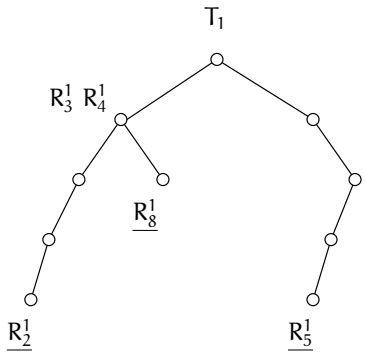


上図は 0000 を分類 ($f(0000) = 2$)．T_1, T_2, T_3 を $0000, 0, 00$ と探索．
T_1 を探索した時点で，T_2, T_3 の破線上の連 R^1_1, R^2_3 に合致することがわかる

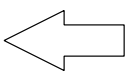
⇨ わかる部分を改めて探索する必要はない

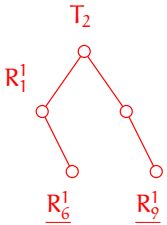
T_2 の節点上の連 R_i^j を T_1 の節点へ追加

T_2 の根節点を T_1 の 0, 1 節点へと重ね，重なっている箇所は連を追加



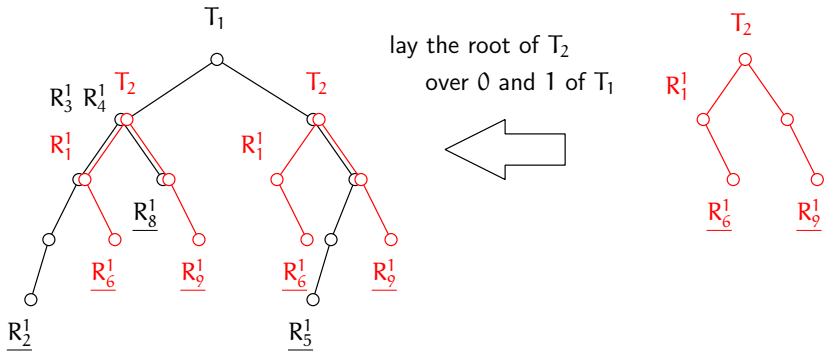
lay the root of T_2
over 0 and 1 of T_1





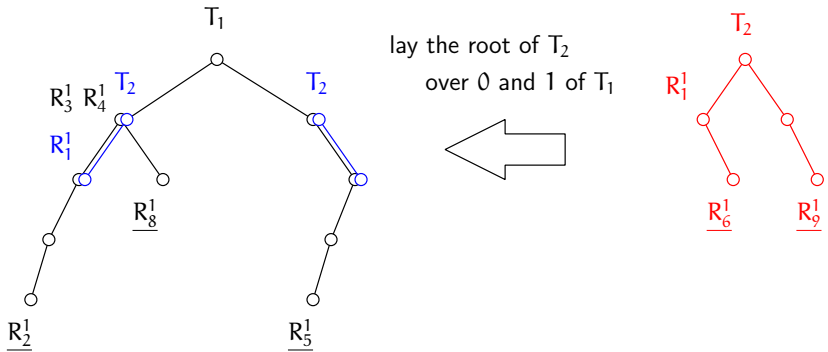
T_2 の節点上の連 R_i^j を T_1 の節点へ追加

T_2 の根節点を T_1 の 0, 1 節点へと重ね，重なっている箇所は連を追加



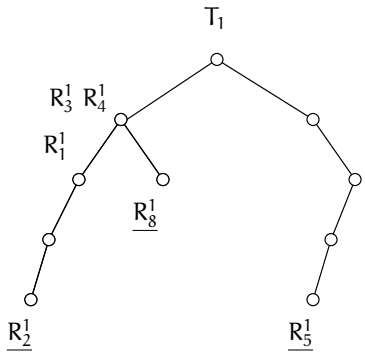
T_2 の節点上の連 R_i^j を T_1 の節点へ追加

T_2 の根節点を T_1 の 0, 1 節点へと重ね，重なっている箇所は連を追加

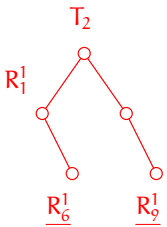


T_2 の節点上の連 R_i^j を T_1 の節点へ追加

T_2 の根節点を T_1 の 0, 1 節点へと重ね，重なっている箇所は連を追加

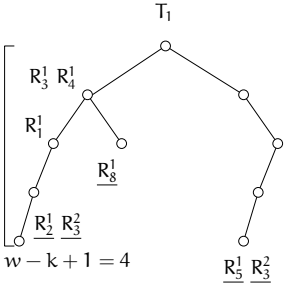


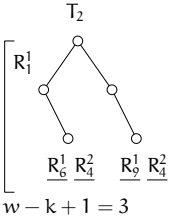
lay the root of T_2
over 0 and 1 of T_1

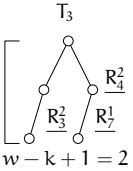


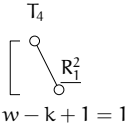
T_l の節点上の連 R_i^j を T_k の節点へ追加 $(1 \leq k < l \leq w)$

T_1, T_2, T_3 上全ての節点に関する経路でそれぞれ
$(T_2, T_3, T_4), (T_3, T_4), (T_4)$ を辿って連を追加すると下記のようになる．



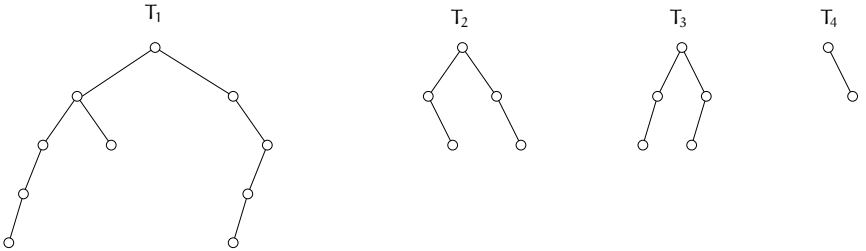




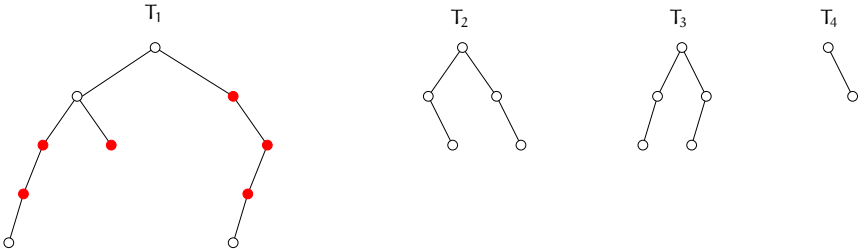


こうすることによって，入力系列 α でトライ T_k を深さ $w-k+1$ まで探索できたら $f(\alpha)$ を求められるようになった

T_k の節点から下位のトライへポインタを張る $(1 \leq k < w)$

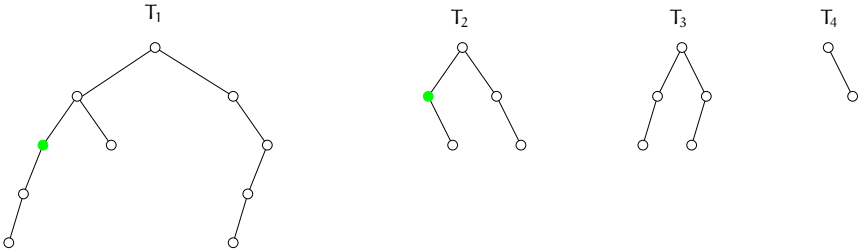


T_k の節点から下位のトライへポインタを張る $(1 \leq k < w)$



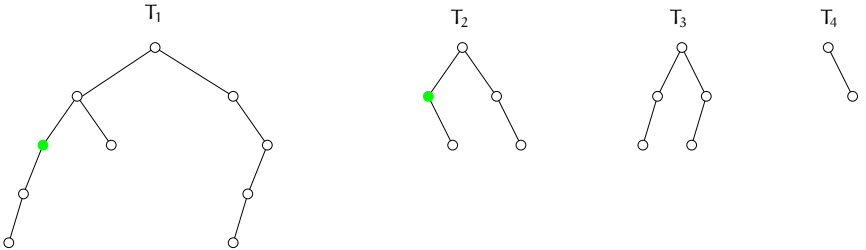
T_1 上の深さ 3 以下で 0, 1 枝，両方の枝が揃っていない節点に注目

T_k の節点から下位のトライへポインタを張る $(1 \leq k < w)$



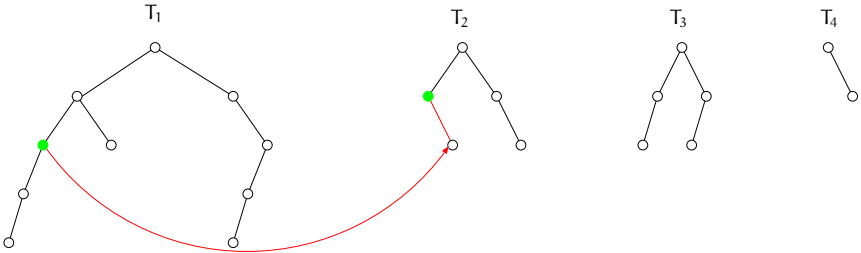
T_1 上の節点 00 に注目
T_1 を 00 と探索することは T_2 を 0 と探索することに対応.

T_k の節点から下位のトライへポインタを張る $(1 \leq k < w)$



T_1 上の節点 00 に注目
T_1 を 00 と探索することは T_2 を 0 と探索することに対応.
T_1 を 001 と辿ろうとするような系列 α は $001*$ という形
そのような系列 α は T_2 を 01 と辿ろうとする

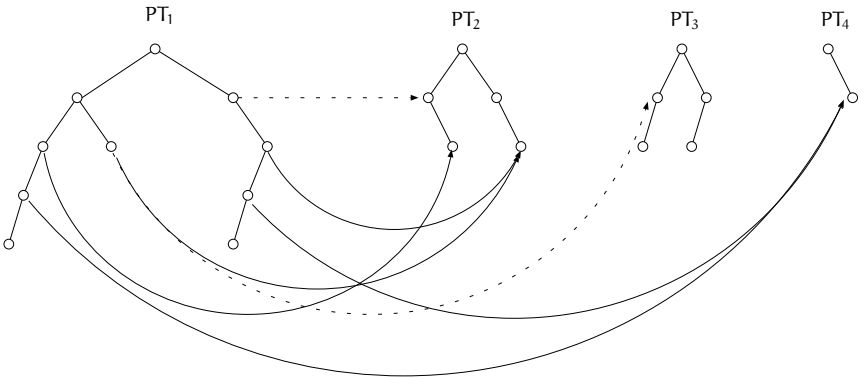
T_k の節点から下位のトライへポインタを張る $(1 \leq k < w)$



T_1 上の節点 00 に注目
T_1 を 00 と探索することは T_2 を 0 と探索することに対応.
T_1 を 001 と辿ろうとするような系列 α は 001$*$ という形
そのような系列 α は T_2 を 01 と辿ろうとする
これより，T_1 の 00 節点の 1 枝は T_2 の 0 節点の 1 枝と対応
よって，T_1 の 00 節点から，T_2 の 01 節点へとポインタを張る

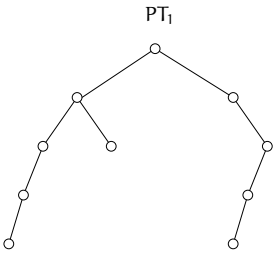
T_k の節点から下位のトライへポインタを張る $(1 \leq k < w)$

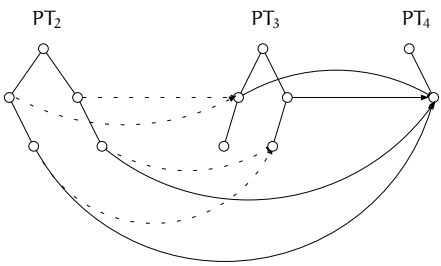
T_1 の深さ 3 以下で 0,1 枝，両方の枝が揃っていない節点にポインタを張ると



T_k の節点から下位のトライへポインタを張る $(1 \leq k < w)$

同様のことを T_2, T_3, T_4 にも行う





こうすることにより入力系列 α の参照回数が w 以下に

Pointed Run-Based Trie とその探索法

以上二つの操作

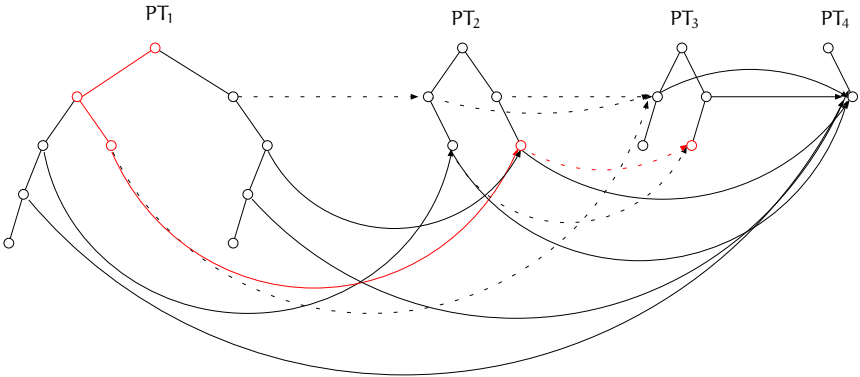
- 下位のトライから上位のトライへ連を追加
- 上位のトライから下位のトライへポインタを張る

を行った Run-Based Trie を Pointed Run-Based Trie と呼ぶ.

Pointed Run-Based Trie の探索の仕方は Run-Baesd Trie とほぼ同じ
ポインタが NULL を指したら探索終了

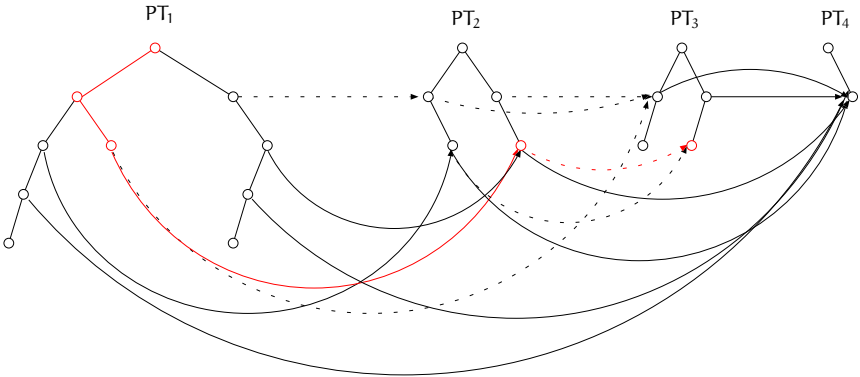
Pointed Run-Based Trie 探索の例

系列 0110 は，赤の経路を辿る



Pointed Run-Based Trie 探索の例

系列 0110 は，赤の経路を辿る



Pointed Run-Based 探索の時間計算量は $\mathrm{O}(nw)$

計算機実験

- ランダムに生成した $0, 1, *$ のルールと $0, 1$ のヘッダ
- パケット分類アルゴリズムのベンチマークである ClassBench によって生成したルールとヘッダ

を用い，

- 線型探索
- Simple Search (Run-Based Trie 探索)
- Pointed Run-Based Trie 探索

における入力系列 α とデータ構造との照合回数を数える実験を行った．流したヘッダの数は全て約 10 万

実験結果：Class Bench のルール・ヘッダ（単位は 10^7 回）

		acl ($w = 120$)	fw ($w = 120$)	ipc ($w = 120$)
2000	線型探索	104.849	101.334	181.525
	Simple Search	8.283	11.523	10.289
	提案手法	7.363	10.906	9.606
4000	線型探索	183.851	208.603	508.676
	Simple Search	13.129	19.997	20.466
	提案手法	11.991	19.397	19.800
6000	線型探索	291.203	321.354	775.077
	Simple Search	18.256	28.457	29.331
	提案手法	16.772	27.843	28.606
8000	線型探索	409.171	418.495	1038.860
	Simple Search	23.753	36.932	37.784
	提案手法	21.903	36.365	37.095
10000	線型探索	506.912	505.801	1304.940
	Simple Search	28.977	45.261	46.845
	提案手法	26.783	44.697	46.110

実験結果：ランダムなルール・ヘッダ（単位は 10^7 回）

灰色部分は，提案手法の照合回数が線型探索の照合回数よりも多い

他の部分は全て提案手法の照合回数が一番少ない

提案手法は Simple Search よりは照合回数が少ない

		$w = 16$	$w = 32$	$w = 64$
200	線型探索	5.087	5.982	6.037
	Simple Search	4.210	7.560	14.128
	提案手法	2.619	4.786	8.857
400	線型探索	8.924	12.020	11.940
	Simple Search	7.172	12.491	23.054
	提案手法	4.781	8.716	16.385
600	線型探索	11.663	17.966	18.038
	Simple Search	10.096	17.299	31.722
	提案手法	6.874	12.578	23.841
800	線型探索	13.361	24.030	23.969
	Simple Search	13.089	22.128	40.162
	提案手法	9.009	16.518	31.258
1000	線型探索	14.877	29.947	30.184
	Simple Search	15.794	26.750	48.665
	提案手法	10.913	20.237	38.775

まとめ

- Simple Search の時間計算量を $O(nw + w^2)$ から $O(nw)$ にする Pointed Run-Based Trie を提案
- ランダムに生成したルールリストだと，ルール数が多くなると（連の数が多くなると）線型探索よりも性能が悪化
- Simple Search よりはどんなルールリストでも照合回数が減少

今後の課題

1. Pointed Run-Based Trie 上には不要な連が存在（例えば PT_1 の 1100 節点上の $\overline{R_3^2}$ は不要）
   ⇨ 不要な連を追加しないようにアルゴリズムを改良

2. 上位トライへ下位の連を複製するのではなく，参照するようにプログラムを改良

3. Run-Based Trie・Pointed Run-Based Trie を用いた，探索時間がルール数 n に依存しないアルゴリズムの提案

課題 3 を解決するために次のことを考えている．

ルールリスト中の連の数を一つにしたい

ルールリスト中の全てのルールの連数が一つだと大変良い (理由省略)

⟹ ルールリストを行列と見做し，全てのルールの連の数が一つになるように列交換をしよう！

⟹ けれども，全てのルールの連の数が一つになるように列交換を行うことができないルールリストが存在

⟹ 幾つかの部分ルールリストに分け，それぞれの部分ルールリストで連の数を一つにすれば？

⟹ 部分ルールリストの数 k が w^2 くらいに収まらないだろうか？（$\lceil \mathrm{n} \rceil$ に分ければ，それぞれのルールリストを適切に列交換可能．ただ，$\mathrm{k} = \lceil \mathrm{n} \rceil$ と n に依存していて嬉しくないが...）

ルールリスト分解・ビット置換

以下のようにルールリストを複数のルールリストへ分割し，それぞれのルールリスト中のルールの連の数が一つになるように列交換を実行

original	
R_1	$*0*1$
R_2	0000
R_3	$0*00$
R_4	$0*1*$
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$
R_{10}	$*000$
R_{11}	$*1*1$
R_{12}	$***1$

$\mathbf{R}_1$	
R_2	0000
R_5	1100
R_6	$*01*$
R_7	$**10$
R_8	$01**$
R_9	$*11*$
R_{10}	$*000$
R_{12}	$***1$

サジェスチョン
お願い致します

$\mathbf{R}_2$	
R_1	$*0*1$
R_3	$0*00$
R_4	$0*1*$
R_{11}	$*1*1$

$\Downarrow$ 列交換

$\mathbf{R}_2'$	
R_1	$01**$
R_3	$*000$
R_4	$**10$
R_{11}	$11**$