疎なルールのもとでの RBT からの決定木構築法

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 5 月 12 日

COMP・AL, 長崎県建設総合会館

本日の内容

研究背景

Run-Based Trie とその探索法

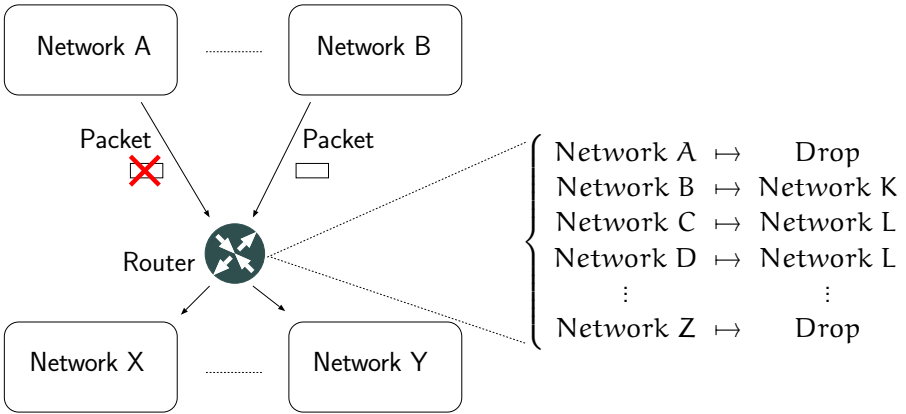
決定木

疎なルールに対する決定木構築

計算機実験

まとめと課題

パケット分類



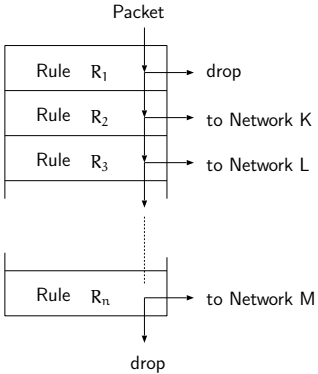
ポリシに従ってパケットを分類

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



($R_1, R_2, \ldots, R_n$ 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函数

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

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
函数 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函数

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

任意のビットマスクに対応しルール数 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

Run-Based Trie (RBT) というデータ構造と RBT から構築した決定木を用いた探索時間がルール数 n に依存しない探索法を提案

RBT から構成される決定木の領域計算量は $O(n^w)$...

決定木の節点のラベルに注目した決定木の枝刈り法を提案
($w = 16$ のルールリストに対して決定木構築可能)

密なルールリストに対しては無理だが，疎なルールリストに対しては決定木を構築できるのでは？

ClassBench で生成される疎なルールリストに対して決定木を構築できず...

疎なルールリストに対する決定木構築法を提案

Run-Based Trie とは

- Run-Based Trie はルールリストで定義された函数

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

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

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

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

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	0000
R_3	$0*00$
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	0000
R_3	$0*00$
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	0000
R_3	$0*00$
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

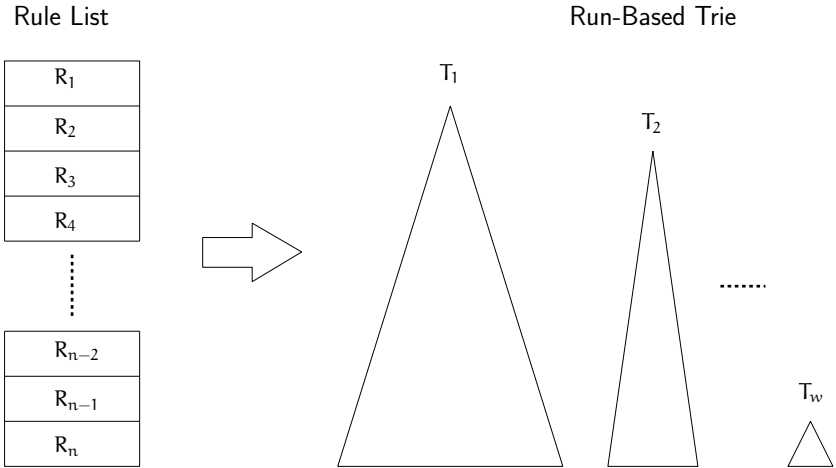
⇩

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	$*\,*\,*\,1$

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



ルールリストから Run-Based Trie（w 本のトライ）を構成

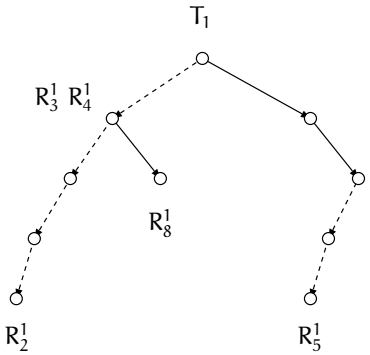
Run-Based Trie 構築

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	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$
R_{10}	$*\ 0\ 0\ 0$
R_{11}	$*\ 1\ *\ 1$
R_{12}	$*\ *\ *\ 1$

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成

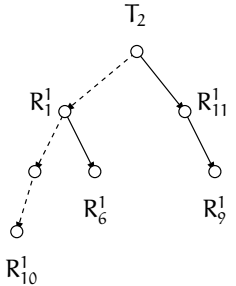


R_i^j はルール R_i の先頭から j 番目の連を表現

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}	$***1$

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成

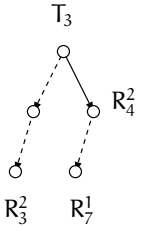


R_i^j はルール R_i の先頭から j 番目の連を表現

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_{10}	$*\,0\,0\,0$
R_{11}	$*\,1\,*\,1$
R_{12}	$*\,*\,*\,1$

Run-Based Trie 構築

右のルールリストから Run-Based Trie を構成



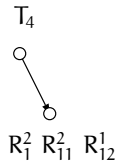
R_i^j はルール R_i の先頭から j 番目の連を表現

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_{10}	∗ 0 0 0
R_{11}	∗ 1 ∗ 1
R_{12}	∗ ∗ ∗ 1

Run-Based Trie 構築

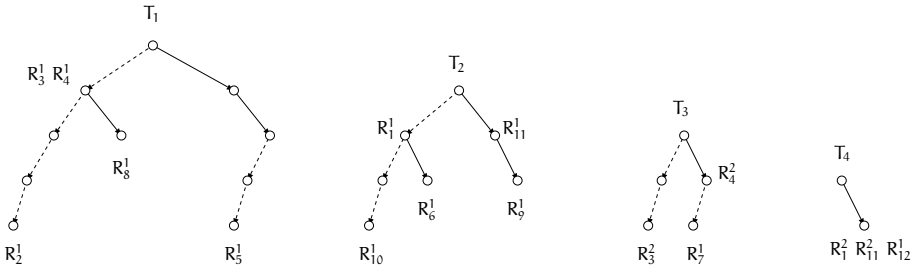
右のルールリストから Run-Based Trie を構成

R_1	$*\ 0\ *\ 1$
R_2	$0\ 0\ 0\ 0$
R_3	$0\ *\ 0\ 0$
R_4	$0\ *\ 1\ *$
R_5	$1\ 1\ 0\ 0$
R_6	$*\ 0\ 1\ *$
R_7	$*\ *\ 1\ 0$
R_8	$0\ 1\ *\ *$
R_9	$*\ 1\ 1\ *$
R_{10}	$*\ 0\ 0\ 0$
R_{11}	$*\ 1\ *\ 1$
R_{12}	$*\ *\ *\ 1$



R_i^j はルール R_i の先頭から j 番目の連を表現

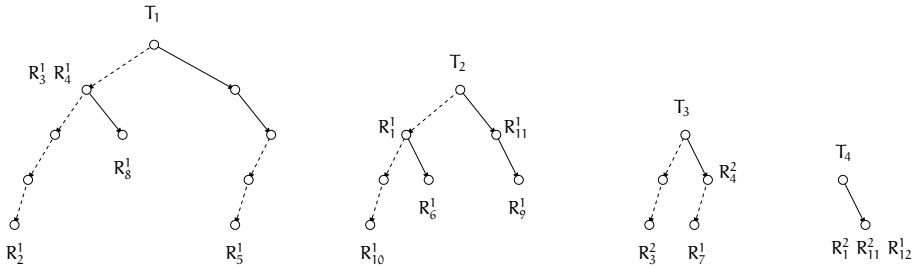
Run-Based Trie 構築



前スライドのルールリストから構成される Run-Based Trie

Run-Based Trie 探索

長さ n の配列 A, C と変数 B を用意し，T_1 の根から探索
B は $n+1$ で，A の要素は全て 0 で，C の i 番目は R_i の連の数で初期化

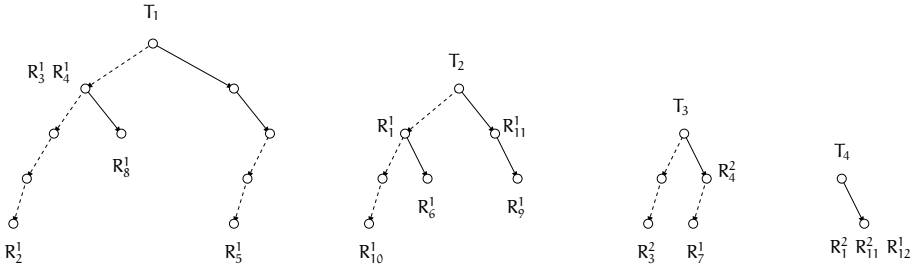


B = 13

A	0	0	0	0	0	0	0	0	0	0	0	0

C	2	1	2	2	1	1	1	1	1	1	2	1

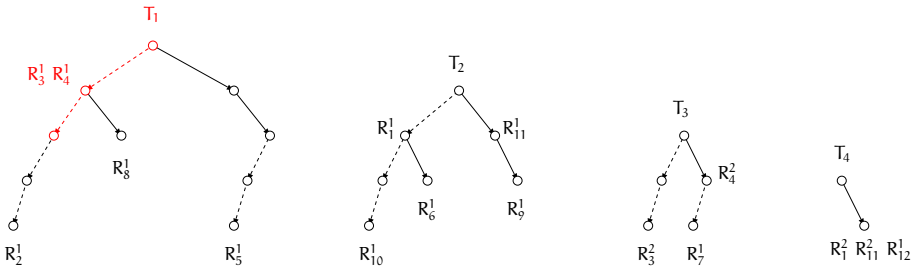
Run-Based Trie 探索の例：0011 を探索



$B = 13$

A	0	0	0	0	0	0	0	0	0	0	0	0
C	2	1	2	2	1	1	1	1	1	1	2	1

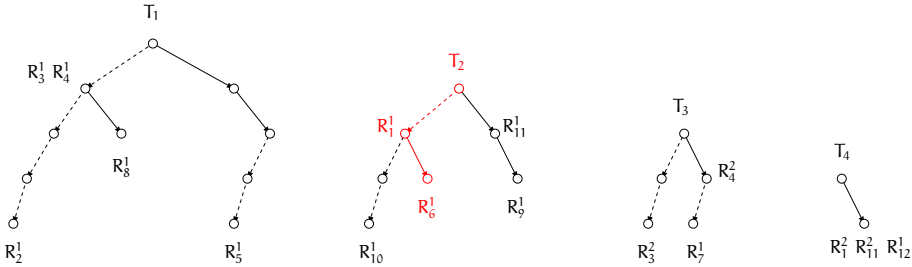
Run-Based Trie 探索の例：0011 を探索



$B = 13$

A	0	0	1	1	0	0	0	0	0	0	0	0
C	2	1	2	2	1	1	1	1	1	1	2	1

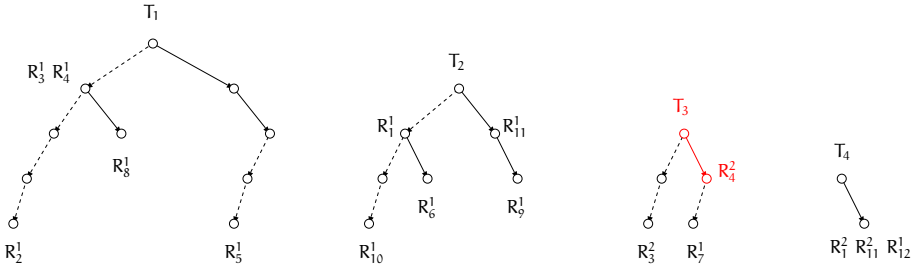
Run-Based Trie 探索の例：0011 を探索



B = 6

A	1	0	1	1	0	1	0	0	0	0	0	0
C	2	1	2	2	1	1	1	1	1	1	2	1

Run-Based Trie 探索の例：0011 を探索

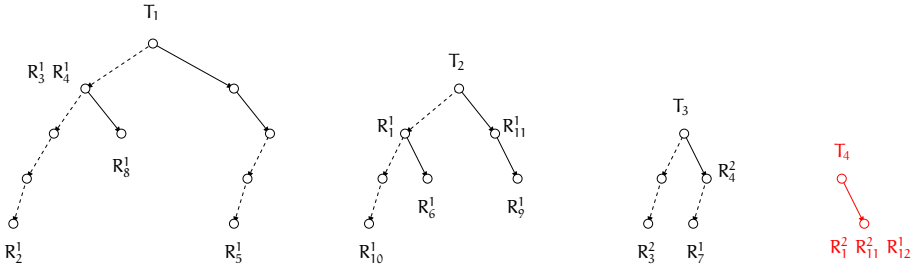


B = 4

A	1	0	1	2	0	1	0	0	0	0	0	0

C	2	1	2	2	1	1	1	1	1	1	2	1

Run-Based Trie 探索の例：0011 を探索



B = 1

A	2	0	1	2	0	1	0	0	0	0	1	1

C	2	1	2	2	1	1	1	1	1	1	2	1

RBT 探索の計算量

- 入力系列 α で RBT を辿る計算量 $O(w^2)$
- 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 $O(w^2 + nw)$

探索時間が n に依存しないアルゴリズムが欲しい

ルール数 n に依存しない探索法が欲しい

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が n に依存

パケットによる T_i の探索パタン S_i は有限

⇩

$S_1 \times S_2 \times \cdots \times S_w$ の各要素 s に最優先ルール R_{prior} を与える

⇩

T_i でパケットがどのパタン $s_i \in S_i$ に合致するかだけを調べればよい

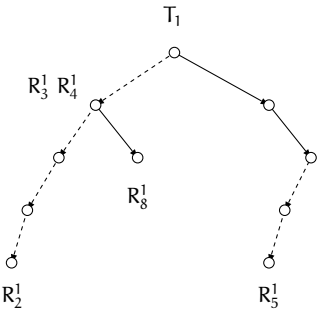
⇩

ルール数 n に依存せずにパケット分類可能

合致連集合

T_1 を探索する系列は全部で $0000, 0001, \ldots, 1111$ の 16 個
それぞれ (1) 式のように連に合致
ϕ はいずれの連にも合致しないことを表す

$$
\begin{array}{ll}
0000 : \{R_2^1, R_3^1, R_4^1\} & 1000 : \phi \\
0001 : \{R_3^1, R_4^1\} & 1001 : \phi \\
0010 : \{R_3^1, R_4^1\} & 1010 : \phi \\
0011 : \{R_3^1, R_4^1\} & 1011 : \phi \\
0100 : \{R_3^1, R_4^1, R_8^1\} & 1100 : \{R_5^1\} \\
0101 : \{R_3^1, R_4^1, R_8^1\} & 1101 : \phi \\
0110 : \{R_3^1, R_4^1, R_8^1\} & 1110 : \phi \\
0111 : \{R_3^1, R_4^1, R_8^1\} & 1111 : \phi
\end{array}
\qquad (1)
$$



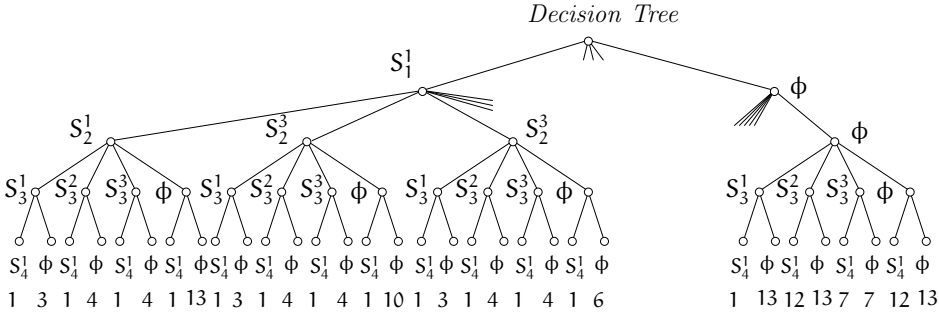
合致連集合

合致連集合：RBT の T_i を探索した際に合致する連の組合せ集合 S_i

先の RBT の各 T_i に対する合致連集合 S_i は以下の通り

$$
\begin{aligned}
S_1 &= \{\ \{R_3^1, R_4^1\},\ \{R_2^1, R_3^1, R_4^1\},\ \{R_3^1, R_4^1, R_8^1\},\ \{R_5^1\},\ \phi\ \} \\
S_2 &= \{\ \{R_1^1\},\ \{R_1^1, R_{10}^1\},\ \{R_1^1, R_6^1\},\ \{R_{11}^1\},\ \{R_9^1, R_{11}^1\}\ \} \\
S_3 &= \{\ \{R_3^2\},\ \{R_4^2\},\ \{R_4^2, R_7^1\},\ \phi\ \} \\
S_4 &= \{\ \{R_1^2, R_{11}^2, R_{12}^1\},\ \phi\ \}
\end{aligned}
$$

決定木 ($S_1 \times S_2 \times \cdots \times S_w \times \{1, 2, \ldots, n+1\}$)



RBT を用いて決定木を降りるだけで探索終了
探索時間計算量は $O(w^2)$ とルール数 n に依存しない

決定木探索の問題点

探索時間は $O(w^2)$ だが領域計算量が $O(n^w)$

⇩

領域計算量が膨大で実用的でない

⇩

決定木の枝刈りアルゴリズムが必要

⇩

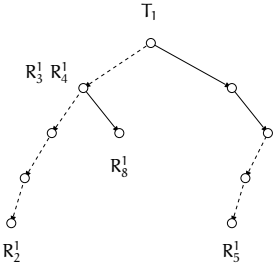
S_i に合致するビットパタンに注目した枝刈り手法を提案（論文投稿中）

合致連集合 S_i の要素にラベル付け

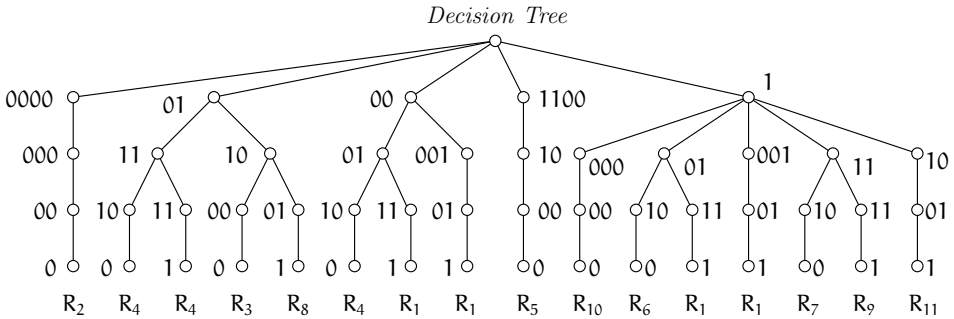
合致連集合 S_i の元 s に，s を得るパケットのビットパタンを付与

$$S_1 = \{ \{R_3^1, R_4^1\}, \{R_2^1, R_3^1, R_4^1\}, \{R_3^1, R_4^1, R_8^1\}, \{R_5^1\}, \phi \}$$
$$S_2 = \{ \{R_1^1\}, \{R_1^1, R_{10}^1\}, \{R_1^1, R_6^1\}, \{R_{11}^1\}, \{R_9^1, R_{11}^1\} \}$$
$$S_3 = \{ \{R_3^2\}, \{R_4^2\}, \{R_4^2, R_7^1\}, \phi \}$$
$$S_4 = \{ \{R_1^2, R_{11}^2, R_{12}^1\}, \phi \}$$

$$\bar{S}_1 = \{ 00, 0000, 01, 1100, 1 \}$$
$$\bar{S}_2 = \{ 001, 000, 01, 10, 11 \}$$
$$\bar{S}_3 = \{ 00, 11, 10, 01 \}$$
$$\bar{S}_4 = \{ 1, 0 \}$$

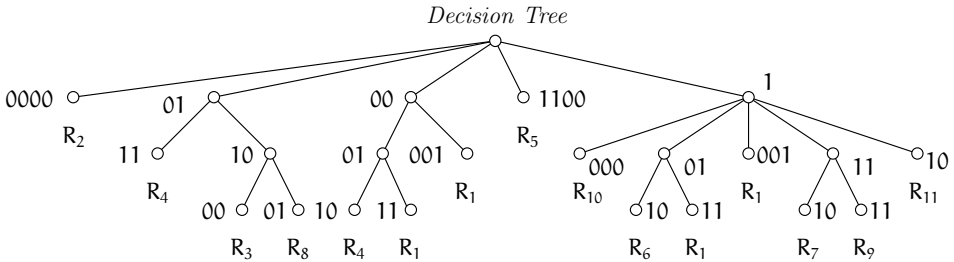


枝刈り手法を適用した決定木



ノード数: 330 → 48

枝刈り手法を適用した決定木



ノード数: 48 → 23

疎なルールリストに対する決定木構築

ビット長が短いルールに対しては決定木を構築可能
ビット長が長くなると合致連集合の要素が増えるので決定木構築不可

⇩

合致連集合 S_i の要素数が増えることが問題ならば，$|S_i|$ が小さくなる（と予想される）疎なルールリストに対しては決定木構築可能？

そうは問屋がおろさない...

⇩

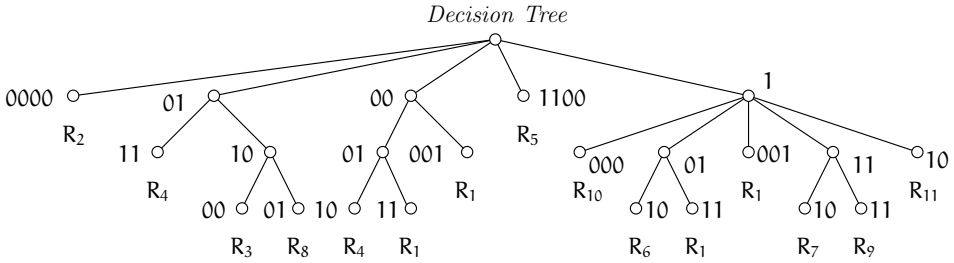
疎なルールリストに対する決定木構築法を提案

決定木を構築できない理由

疎なルールリストに対して何故従来の手法では決定木が構築できないのか？

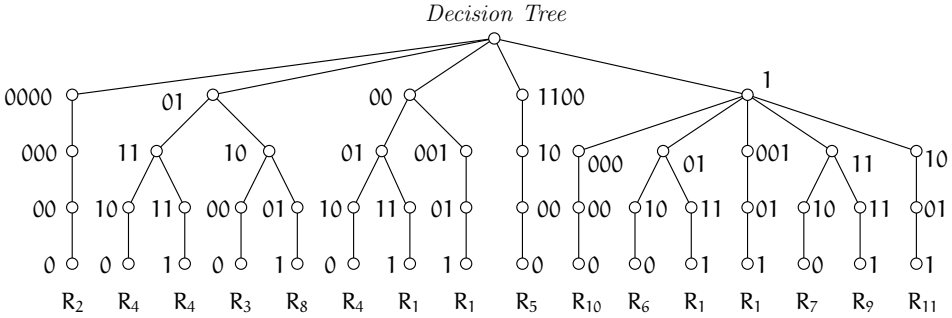
1. 最終的に枝刈りされる場合も，深さ w の節点まで構築
2. 合致連集合の要素より不要と判断できる節点をも生成

合致連集合の更新による枝刈り



深さ w の節点まで構築して T_w 上の連 R_i^j まで得ないと，そのパスに対する最優先ルールが決定しない．よって，深さ k まで枝刈りされるパスも深さ w の節点まで一旦生成．けれども...

合致連集合の更新による枝刈り



深さ w の節点まで構築して T_w 上の連 R_i^j まで得ないと，そのパスに対する最優先ルールが決定しない．よって，深さ k まで枝刈りされるパスも深さ w の節点まで一旦生成．けれども...

合致連集合の更新による枝刈り

S_i^j に合致するパケットは，i ビット以降から始まり $|\bar{S}_i^j|$ ビットまでに終わる連 R_i^j の中で $\bar{S}_i^j$ に合致する連とも合致する．

$$S_1 = \{ \{R_3^1, R_4^1\}, \{R_2^1, R_3^1, R_4^1\}, \{R_3^1, R_4^1, R_8^1\}, \{R_5^1\}, \phi \}$$
$$S_2 = \{ \{R_1^1\}, \{R_1^1, R_{10}^1\}, \{R_1^1, R_6^1\}, \{R_{11}^1\}, \{R_9^1, R_{11}^1\} \}$$
$$S_3 = \{ \{R_3^2\}, \{R_4^2\}, \{R_4^2, R_7^1\}, \phi \}$$
$$S_4 = \{ \{R_1^2, R_{11}^2, R_{12}^1\}, \phi \}$$

$$\bar{S}_1 = \{ 00, 0000, 01, 1100, 1 \}$$
$$\bar{S}_2 = \{ 001, 000, 01, 10, 11 \}$$
$$\bar{S}_3 = \{ 00, 11, 10, 01 \}$$
$$\bar{S}_4 = \{ 1, 0 \}$$

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}	$***1$

合致連集合の更新による枝刈り

S_i^j に合致するパケットは，i ビット以降から始まり $|\bar{S}_i^j|$ ビットまでに終わる連 R_i^j の中で $\bar{S}_i^j$ に合致する連とも合致する．

$$S_1 = \{ \{R_3^1, R_4^1\}, \{R_2^1, R_3^1, R_4^1\}, \{R_3^1, R_4^1, R_8^1\}, \{R_5^1\}, \phi \}$$
$$S_2 = \{ \{R_1^1\}, \{R_1^1, R_{10}^1\}, \{R_1^1, R_6^1\}, \{R_{11}^1\}, \{R_9^1, R_{11}^1\} \}$$
$$S_3 = \{ \{R_3^2\}, \{R_4^2\}, \{R_4^2, R_7^1\}, \phi \}$$
$$S_4 = \{ \{R_1^2, R_{11}^2, R_{12}^1\}, \phi \}$$

$$\bar{S}_1 = \{ 00, 0000, 01, 1100, 1 \}$$
$$\bar{S}_2 = \{ 001, 000, 01, 10, 11 \}$$
$$\bar{S}_3 = \{ 00, 11, 10, 01 \}$$
$$\bar{S}_4 = \{ 1, 0 \}$$

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}	$***1$

合致連集合の更新による枝刈り

S_i^j に合致するパケットは，i ビット以降から始まり $|\bar{S}_i^j|$ ビットまでに終わる連 R_i^j の中で $\bar{S}_i^j$ に合致する連とも合致する．

$$S_1 = \{\ \{R_3^1, R_4^1\},\ \{R_2^1, R_3^1, R_4^1\},\ \{R_3^1, R_4^1, R_8^1\},\ \{R_5^1\},\ \phi\ \}$$
$$S_2 = \{\ \{R_1^1, R_1^2, R_{11}^2, R_{12}^1\}, \ldots\}$$
$$S_3 = \{\ \{R_3^2\},\ \{R_4^2\},\ \{R_4^2, R_7^1\},\ \phi\ \}$$
$$S_4 = \{\ \{R_1^2, R_{11}^2, R_{12}^1\},\ \phi\ \}$$

$$\bar{S}_1 = \{\ 00,\ 0000,\ 01,\ 1100,\ 1\ \}$$
$$\bar{S}_2 = \{\ 001,\ 000,\ 01,\ 10,\ 11\ \}$$
$$\bar{S}_3 = \{\ 00,\ 11,\ 10,\ 01\ \}$$
$$\bar{S}_4 = \{\ 1,\ 0\ \}$$

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_{10}	$*\ 0\ 0\ 0$
R_{11}	$*\ 1\ *\ 1$
R_{12}	$*\ *\ *\ 1$

合致連集合の更新による枝刈り

合致連集合 S_i をそれぞれ下記のように $\mathfrak{S}_i$ へと更新

これより，深さ w ではなく $i + |\bar{S}_i^j| - 1 = w$ で最優先ルールがわかる

⇒ 深さ w まで必ずしも節点を構築しなくてよい

$$S_1 = \{\ \{R_3^1, R_4^1\},\ \{R_2^1, R_3^1, R_4^1\},\ \{R_3^1, R_4^1, R_8^1\},\ \{R_5^1\},\ \phi\ \}$$

$$S_2 = \{\ \{R_1^1\},\ \{R_1^1, R_{10}^1\},\ \{R_1^1, R_6^1\},\ \{R_{11}^1\},\ \{R_9^1, R_{11}^1\}\ \}$$

$$S_3 = \{\ \{R_3^2\},\ \{R_4^2\},\ \{R_4^2, R_7^1\},\ \phi\ \}$$

$$S_4 = \{\ \{R_1^2, R_{11}^2, R_{12}^1\},\ \phi\ \}$$

$$\mathfrak{S}_1 = \{\ \{R_3^1, R_4^1\},\ \{R_3^1, R_4^1, R_1^1, R_2^1, R_{10}^1, R_3^2\},\ \{R_3^1, R_4^1, R_8^1, R_{11}^1\}, \{R_{11}^1, R_5^1, R_3^2\},\ \phi\ \}$$

$$\mathfrak{S}_2 = \{\ \{R_1^1, R_1^2, R_{11}^2, R_{12}^1\},\ \{R_1^1, R_{10}^1, R_3^2\},\ \{R_1^1, R_6^1, R_4^2\},\ \{R_{11}^1\}, \{R_9^1, R_{11}^1, R_4^2\}\ \}$$

$$\mathfrak{S}_3 = \{\ \{R_3^2\},\ \{R_4^2, R_1^2, R_{11}^2, R_{12}^1\},\ \{R_4^2, R_7^1\},\ \{R_1^2, R_{11}^2, R_{12}^1\}\ \}$$

$$\mathfrak{S}_4 = \{\ \{R_1^2, R_{11}^2, R_{12}^1\},\ \phi\ \}\ (= S_4)$$

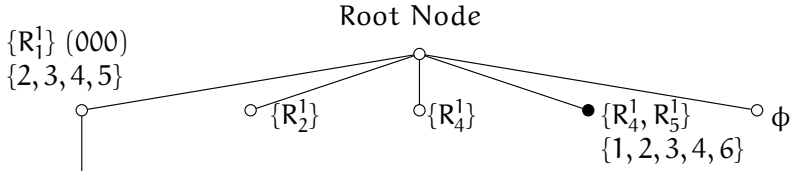
不一致ルール集合による枝刈り

S_i^j に合致するパケットが合致しないルールの集合 N_i^j を生成

$$
\begin{aligned}
S_1 &= \{\{R_1^1\}, \{R_2^1\}, \{R_4^1\}, \{R_4^1, R_5^1\}, \phi\} & N_1 &= \{\{2,3,4,5\}, \{1,3,4,5,6\}, \{1,2\}, \{1,2,3,4,6\}, \phi\} \\
S_2 &= \{\{R_6^1\}, \phi\} & N_2 &= \{\{2\}, \{1,3,4,5,6\}\} \\
S_3 &= \{\phi\} & N_3 &= \{\phi\} \\
S_4 &= \{\{R_2^2\}, \{R_4^2\}, \phi\} & N_4 &= \{\{1,4,6\}, \{2,3,5\}, \{4\}\} \\
S_5 &= \{\phi\} & N_5 &= \{\phi\} \\
S_6 &= \{\{R_4^3\}, \{R_6^2\}, \phi\} & N_6 &= \{\{1,2,5,6\}, \{1,2,4,5\}, \phi\} \\
S_7 &= \{\phi\} & N_7 &= \{\phi\} \\
S_8 &= \{\phi\} & N_8 &= \{\phi\}
\end{aligned}
$$

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

不一致ルール集合による枝刈り



$S_2 =$
$\{\{R_6^1\}, \phi\}$ (1, 0)

$S_4 =$
$\{\{R_2^2\}, \{R_4^2\}, \phi\}$

矢印の節点に到達するパケットは，ルール 2, 3, 4, 5, 6 には合致しないので，
S_4 の $\{R_2^2\}, \{R_4^2\}$ に対応する節点は生成しない

実験環境

OS	:	Mac OS X 10.9.5
CPU	:	Intel Core i5 1.4 GHz
主記憶容量	:	4GB
実装言語	:	C++
コンパイラ	:	gcc version 4.9.3

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リスト（疎なルールリスト）を用い構築時間 (s), 節点数を計測

実験結果

Table : 決定木の構築時間（秒）

n	acl	fw	ipc
100	2.395	71.475	4.336
500	37.449	†	977.950
1000	400.870	†	†
1500	1679.874	†	†
2000	2809.696	†	†

Table : 決定木の節点数

n	acl	fw	ipc
100	5005	65665	15004
500	8718	†	498689
1000	25030	†	†
1500	62403	†	†
2000	43516	†	†

† は 1 時間で決定木を構築できなかったことを表す

- fw が $n = 100$ までしか生成できないのは，(1-3) のようなレンジルールを上手く扱えなかったことが原因
- 従来の手法では acl の 100 ルールのルールリストで決定木構築不可
  ⇒ acl の 2000 ルールまで決定木構築可能

まとめと今後の課題

まとめ

ClassBench で生成されるような疎なルールに対する決定木構築手法を提案

今後の課題

- より大きなサイズのルールリストに対する決定木構築法の考案
  - 終端節点等の子孫の部分グラフが等価な節点の共有
  - 合致連集合が同じ節点の共有
- 合致連集合以外の組合せ集合による決定木の考案