Vol.2017-AL-162 No.2
2017/3/13

# 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,a)]　田中 賢[1,b)]　三河 賢治[2,c)]

**概要**：Run-Based Trie (RBT) は，アルファベット $\{0,1,*\}$ 上の長さ $w$ の系列のリストによって定義された函数 $f:\{0,1\}^w \to \mathbb{N}$ を計算することができるデータ構造である．パケット分類問題は，ネットワーク機器を通過するパケットと与えられているフィルタリングルールリストの各ルールとを照合して，パケットに合致する優先度の最も高いルールを求める問題である．本稿では，単一の連からなる RBT を提案する．単一の連からなる RBT は，各ルールの連は一つだけ，RBT 上の各節点が持つルールは一つのみという特徴を持つ．本稿では，単一の連からなる RBT の構成法と探索時間計算量が $O(w)$ とルール数 $n$ に依存しない探索法を示す．また，計算機実験により提案手法の有効性を確認する．

# A Fast Search Method for Run-Based Trie Consisting of A Single Run

Harada Takashi[1,a)]　Tanaka Ken[1,b)]　Mikawa Kenji[2,c)]

## 1. はじめに

Run-Based Trie (RBT) は，アルファベット $\{0,1,*\}$ 上の長さ $w$ の系列のリストによって定義された函数 $f:\{0,1\}^w \to \{1,2,\ldots,n+1\}$ を計算することができる，三河らが提案したデータ構造である [1].

函数 $f:\{0,1\}^w \to \{1,2,\ldots,n+1\}$ として表されるような問題にパケット分類問題がある．$f$ をパケット分類のポリシとみると $w$ はパケット長で $n$ はルール数である．パケット分類では，パケットヘッダの送信元 IP，宛先 IP，送信元ポート番号，宛先ポート番号，プロトコルの五つの性質に基づいてパケットを分類することが多い．今までに提案されてきたパケット分類手法 [2] が対象にしているルールの多くは，前記のような幾つかのフィールドに分かれており，各フィールドは CIDR や `0-65535` のような範囲指定で表される．

文献 [2] にあるように今までに数多のパケット分類が提案されてきた．前述のような CIDR や範囲指定のルールを対象にしている手法の中で最先端の手法に HybridCuts[3] や井上らが提案した手法 [4] がある．HybridCuts は与えられたルールリストをその特徴に従って複数の部分ルールリストに分割し，それぞれの部分ルールリストに対して Cutting アルゴリズムを適用する手法である．井上らの手法は，Binary Decision Diagram (BDD) と Multi-valued Decision Diagram (MDD) を用いたパケット分類手法である．なお井上らの手法が扱う問題は，一つのスイッチによるパケット分類問題ではなく，複数のスイッチ間でパケットがどのうように転送されるかを特定する問題であり本稿で扱う問題とは異なる．

今後 Software Defined Network (SDN) 等の普及によってより複雑な条件に従ったパケット分類が必要となり，プレフィックスルールやレンジルールよりも表現力の強い任意のビットマスクルールに適したアルゴリズムが求められる．さらに，一般に管理者は外部の脅威が排除されたことを確かめられないので，ルールリストに一旦追加されたルールは削除されない．これよりフィルタリングルールリ

[1] 神奈川大学大学大学院理学研究科情報科学領域
Kanagawa University, Hiratsuka, Kanagawa 259–1293, Japan
[2] 新潟大学学術情報基盤機構情報基盤センター
Niigata University, Niigata, Niigata 950–2181, Japan
[a)] takaharada@jindai.jp
[b)] ktanaka@info.kanagawa-u.ac.jp
[c)] mikawa@cais.niigata-u.ac.jp

ストのルール数は増える一方なので，探索時間もそれに応じて増大していく．よって今後増大するセキュリティルールに対応するためには，分類時間がルール数に依存しないフィルタリングアルゴリズムが不可欠である．

このような状況に対して我々は，任意のビットマスクルールに適したデータ構造 RBT とこれを用いた探索法を提案し，分類時間がルール数に依存しない手法を提案した [1]．この手法 [1] に対しては，RBT から構成される決定木の領域計算量が $O(n^w)$ と膨大になるという問題点 [1] と，RBT の探索には入力系列を余分に参照する問題点があった．そこで我々は，RBT から構成される決定木の領域計算量を $O(n^w)$ から $O(3^w)$ へと減少させる決定木の枝刈り法 [5] と，ポインタと連を RBT に付与することにより RBT 探索の探索時間計算量を $O(w^2+nw)$ から $O(nw)$ へと減少させる手法とを提案した [6]．

けれども，枝刈り法を適用した決定木の空間計算量は $O(3^w)$ と指数オーダで現実的ではなく，ポインタとを付与した RBT を用いた探索法の時間計算量は $O(nw)$ とルール数 $n$ に依存している．これより，領域計算量が多項式オーダになるように決定木の更なる枝刈りと，分類時間がルール数に依存しないように RBT の探索法の改良とが求められる．

本稿では，十分小さな領域計算量で探索時間計算量がルール数に依存しない手法を今後提案するために，単一の連からなる RBT による領域計算量が $O(nw)$ で探索時間計算量が $O(w)$ である探索法を示す．また，計算機実験により提案手法の有効性を確認する．

## 2. Run-Based Trie

任意の位置にビットマスク'$*$' を含むビットマスクルール $R \in \{0,1,*\}^w$ とパケットのビット列 $p \in \{0,1\}^w$ との照合を行う際には，ルールのビットマスク $*$ 部分は省略できるので，ビットマスクルール R の $*$ 以外の 0,1 の部分が重要である．任意の位置にビットマスクを含むビットマスクルールで構成されたルールリストの例を表 1 に示す．我々の手法は，連とよばれるルール中で 0,1 が連続する部分に注目する．

**定義 2.1** $R \in \{0,1,*\}^w$ を長さ $w$ のビットマスクルールとする．下記の二つの条件を満たす R の部分列 $r_i r_{i+1} \dots r_j$ $(1 \le i \le j \le w)$ を R の連と呼ぶ．

- $r_k = 0 \vee r_k = 1 \quad (i \le k \le j)$
- $(i \ge 2 \Rightarrow r_{i-1} = *) \wedge (j \le w-1 \Rightarrow r_{j+1} = *)$

例えば，長さ 16 のビットマスクルール

$$*101**001*1**010$$

は，2 番目，7 番目，11 番目，14 番目から始まる四つの連 101，001，1，010 を含む．ルール $R_i$ の k 個の連を $R_i^1, R_i^2, \dots, R_i^k$ $(1 \le k \le \lceil \frac{w}{2} \rceil)$ と表す．我々の探索法の基本的な考えは，パケットのビット列がルールリスト中のどの連と合致するかを調べ，すべての連が合致するルールを調べ，合致するルールの中で一番優先度の高いルールを返す，というものである．パケットのビット列がどの連と合致するかを調べるために，ルールリスト中の連の開始位置毎に $w$ 個のトライを構成する．

**表 1** ビットマスクルールのルールリスト

| | |
|---|---|
| $R_1$ | $*0*1$ |
| $R_2$ | 0000 |
| $R_3$ | $0*00$ |
| $R_4$ | $0*1*$ |
| $R_5$ | $*1*1$ |
| $R_6$ | $***1$ |

長さ $w$ のルールを $n$ 個含むルールリスト $\mathbf{R} = [R_1, R_2, \dots, R_n]$ から構成した $w$ 個のトライを $T_1, T_2, \dots, T_w$ とする．トライ $T_k$ はルールリストに含まれる連の中で k ビット目から始まる連で構成される．この $w$ 個のトライ $T_1, T_2, \dots, T_w$ の集まりを Run-Based Trie (RBT) という．表 1 から構成した RBT を図 1 に示す．図 1 の破線は 0 枝を実線は 1 枝を表す．

RBT は長さ $w$，ルール数 $n$ のルールリストに含まれる連から構成され，かつ RBT においてそれぞれの連は一カ所にのみ出現するので，RBT の領域計算量は $O(nw)$ である

### 2.1 Simple Search

我々が Simple Search と呼んでいる RBT の探索法は，$T_1$ から $T_w$ の順にトライ $T_i$ を入力パケットのビット列で辿り，入力パケットに合致する連を集めて合致するルールを探し，合致したルールの中で最優先のルールを返す，という探索法である．探索に先立って各ルールの連の添字を格納する配列 A[n] と最優先ルールを格納する変数 B を用意し，それぞれ $A[i] := 0$ $(1 \le i \le n)$，$B = n+1$ と初期化する．そして，各トライ $T_i (1 \le i \le w)$ を入力パケット p の i 番目からのビット列 $p[i]p[i+1] \dots p[w]$ で辿る．各トライを辿るとき，トライの節点に連の印 $R_i^j$ が付いていたら，連の添字 j の値と配列 A[i] の値とを比較して，$j = A[i-1]+1$ ならば，$A[i] := j$ とする．さらに，連の印に下線が引いてあり $i < B$ ならば，$B := i$ とする．全てのトライを辿り終えたときの B の値が，入力パケットに対する最優先のルールとなる．

Simple Search の探索時間計算量を考える．各トライ $T_1, T_2, \dots, T_w$ を探索するのに $w+(w-1)+\dots+1$ ステップ必要なので，各トライ $T_i$ の探索時間は $O(w^2)$ である．そして，ビット長 $w$ のルールを $n$ 個含むルールリストの連の総数は高々 $nw$ なので，合致した連と配列との比較に必要な時間計算量は $O(nw)$ である．また，合致したルールと最優先ルールとの比較回数は高々ルール数分の $n$ 回

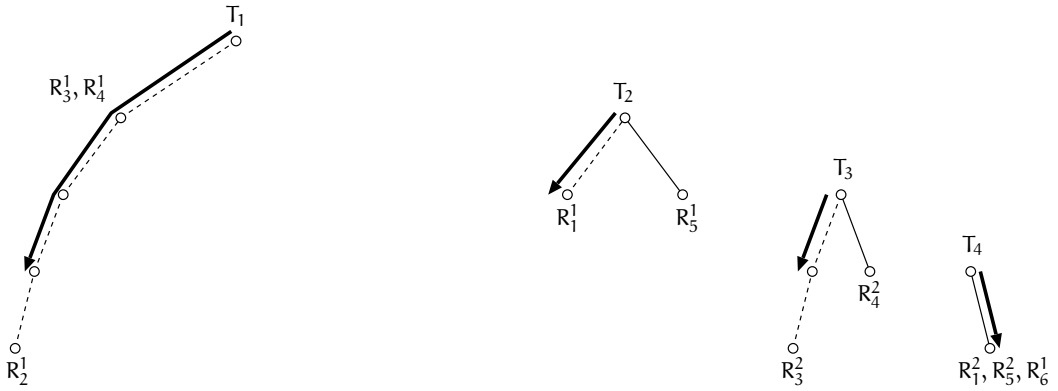


**図 1** 表 1 のルールリストから構成される Run-Based Trie

なので，最優先ルールを見つけるのに必要な時間計算量は $O(n)$ である．よって，Simple Search の時間計算量は $O(w^2 + nw)$ となる．

### 2.2 RBT へ連とポインタを付与

前節に示した Simple Search には入力系列を何度も参照するという冗長な点がある．例えば，図 1 の RBT を用いて 0001 を分類することを考える．図 1 の太実線は各トライを入力系列が辿る経路を表す．初めに $T_1$ を 0001 で辿り $R^1_3, R^1_4$ に合致する．次に $T_2$ を 001 で辿り，$T_3$ を 01 で辿り，$T_4$ を 1 で辿り $T_2, T_3, T_4$ で $R^1_1, R^2_1, R^2_5, R^1_6$ に合致する．しかし $T_1$ を探索した時点でパケット 0001 の全てのビットを参照していて $T_2, T_3, T_4$ を如何に探索するのかわかるので，この時点で $T_2, T_3, T_4$ を同時に探索することが本来可能である．この点に注目して，我々は探索する際に入力系列を一度だけ参照する手法を提案した [6].

入力系列を一度だけ参照するためには，RBT の上位のトライから下位のトライへとポインタを張り，下位のトライから上位のトライへと連を付与すればよい．図 1 の RBT に連とポインタを付与した RBT を図 2 に示す．

この連とポインタを付与した RBT の探索法は，Simple Search と同様である．Simple Search は $T_k$ の探索を終了したら $T_{k+1}$ の根節点から探索を続けたのに対して，連とポインタを付与した RBT の探索法は，$T_1$ の根節点から辺またはポインタを辿れなくなるまで高々長 $w$ 分だけ探索する．図 2 の太実線は系列 0001 が連とポインタを付与した RBT を辿る経路を表す．連とポインタを付与した RBT 探索は RBT 探索に対してトライ全体をビット長 $w$ 分だけ探索するようにした手法なので，RBT 探索の時間計算量 $O(w^2 + nw)$ に対して $O(nw)$ となる．

本節以降で RBT といった場合は，この連とポインタを付与された RBT のことをいう．

## 3. 単一の連からなる RBT

前節のように RBT へ連とポインタを追加しても，RBT の探索時間はルール数 $n$ に依存している．その原因は，RBT の節点に連が高々 $n\lceil \frac{w}{2} \rceil$ 個，散らばって存在し，RBT の探索を行う際には節点に連が存在する度に照合を行わなければならないからである．そこで，ルールリスト中の各ルールが一つの連だけから成る場合を考える．単一の連からなるルールだけを含むルールリストから構成される RBT は，(1) 各節点が持つ連の数は一つだけ，(2) 節点 $v$ が連 $R^1_i$ を持ち，節点 $v$ から到達可能な節点に存在する連 $R^1_j$ のルール番号が小さい場合は，節点 $v$ からポインタを張る必要がない，という二つの特徴を持つ．一つ目の特徴により，単一の連からなる RBT の探索は高々 $w$ 回節点を辿り，連の照合を高々 $w$ 回行うだけで最優先のルールを求められる．

一般にルールリストの各ルールは単一の連からなるルールではなく，単一の連からなるルールのルールリストへと変換できるわけではない．また，与えられたルールリストを単一の連からなるルールのルールリストへ変換する方法も自明ではない．この問題については 3.3 節で述べる．

けれども，もし与えられたルールリストが単一の連からなるルールのルールリストであるか，そのようなルールリストへと変換できれば RBT を用いることにより探索時間計算量が $O(w)$ の探索を行える．例えば，表 2 の単一の連からなるルールのみのルールリストは，表 1 のルールリストを，各ルールの 1, 2, 3 および 4 ビット目を 4, 1, 3 および 2 ビット目へと置換することにより得られ，表 1 のルールリストに対しては探索時間が $O(w)$ の探索を行える．

### 3.1 単一の連からなる RBT の構成法

単一の連からなる RBT の構成法は前章の RBT の構成法と同様であるが，新たに各節点を終端節点と非終端節点に分け，何れのルールにも合致しない入力系列のための，ルール $R_{n+1}$ を持つ終端節点 $t$ を追加する．そして，各節点には変数番号を持たせる．表 2 から構成される RBT を図 4 に示す．図中の白丸は非終端節点を黒矩形は終端節点を表す．RBT の左側にある丸で囲まれた数が，その高さの節点の変数番号を表す．前章の RBT において，0 若しく

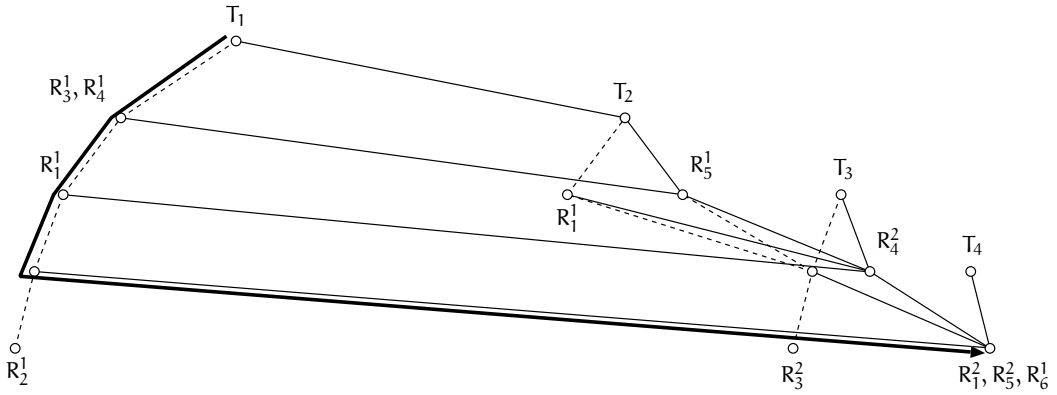


**図 2** 図 1 の RBT に連とポインタを付与

**表 2** 表 1 のルールのビット列を (4 1 3 2) と置換したルールリスト

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

**表 3** 単一の連から成るルールのリストへ変換できないルールリスト

| | 1 2 3 |
|---|---|
| $R_1$ | * 0 0 |
| $R_2$ | 0 * 0 |
| $R_3$ | 0 0 * |

は 1 枝が欠けていた節点の 0 または 1 枝の先として終端節点 t をもってくる．例えば図 4 について前章までの RBT では，$T_3$ の 1 節点の 1 枝は存在しないが新たに提案する RBT では 1 枝の先を終端節点 t とする．

さらに 0 枝 1 枝両方の行き先が同じ節点 v がある場合は，その節点 v の親節点 u と節点 v の 0,1 枝の行き先の節点 w とを図 3 のように直接繋げる．例えば，図 4 の $T_4$ の

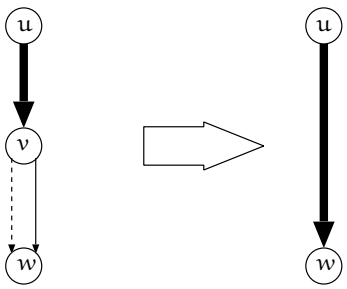


**図 3** 冗長節点の削除

根節点の 0,1 枝の行き先は終端節点 t と同じなので，$T_4$ の根節点を指している $T_3$ の根節点，$T_2$ の 1 節点，$T_1$ の 11 節点と終端節点 t を図 5 のように直接繋げる．また，$T_1$ の根節点から到達不可能な節点を削除する．

### 3.2 単一の連からなる RBT の探索法

単一の連からなる RBT の探索法は前章までの RBT の探索法と同様だが違いが二つある．一つは，$T_1$ の根節点から辿れなくなるまで探索を続けるのではなくて，終端節点に到達するまで探索を続けることである．二つ目は，入力系列の 1 ビット目からの系列で RBT を探索するのではなく，各節点の変数番号目のビットで RBT を探索することである．例えば系列 0001 で図 5 の RBT を探索すると，初めに系列 2 ビット目の 0 で探索するので 0 枝を辿り，次に 4 ビット目の 1 で探索するので 1 枝を辿る．そして，終端節点に到達したので探索を止める．

### 3.3 ルールリストの列置換による連の統合

本章で提案した手法は，各ルールが連を一つだけ持つルールリストを対象にしているので，提案手法を適用したければ各ルールの連の数が一つになるように与えられたルールリストを変換しなければならない．そこで我々はルールリストを n 個の長さ w のルールが成す $n \times w$ の行列と見做し，行列の列の置換によって各ルールの連の数を一つにすることを考えている．

与えられるルールリストには，$n = 3, w = 3$ と小さなルールリストでも表 3 のように単一の連から成るルールのリストへ変換できないルールリストが存在する．そこで，与えられたルールリストを単一の連から成るルールのリストへ変換できる k 個のルールリストへ分割することを考える．例えば表 4 のルールリスト RL は，単一の連から成るルールのリストへ変換できないルールリストなので，$R_1, R_3, R_4, R_5$ から成るルールリスト $RL_1$ と $R_2, R_6, R_7, R_8$ から成るルールリスト $RL_2$ の二つのルールリストへと分割し，RL ではなく $RL_1$ と $RL_2$ を変換する．$RL_1$ は各 i $(1 \leq i \leq 6)$ 列を

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 3 & 2 & 5 & 1 & 6 \end{pmatrix}$$

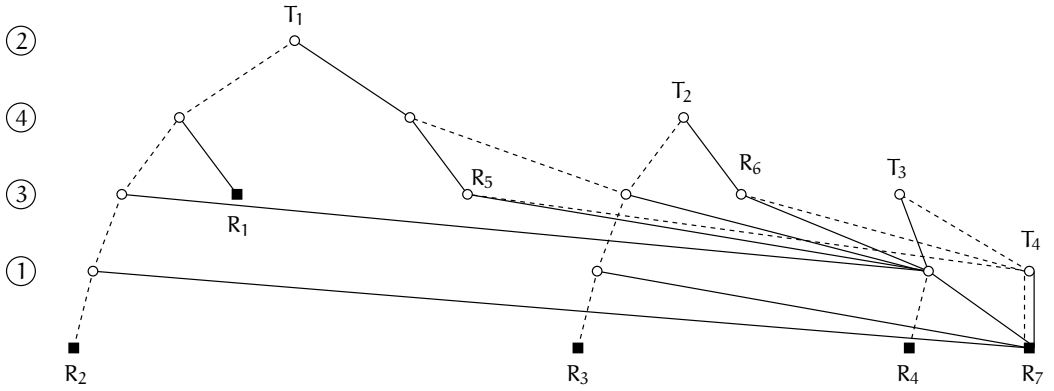


**図 4** 表 2 のルールリストから構成される単一の連から成る RBT

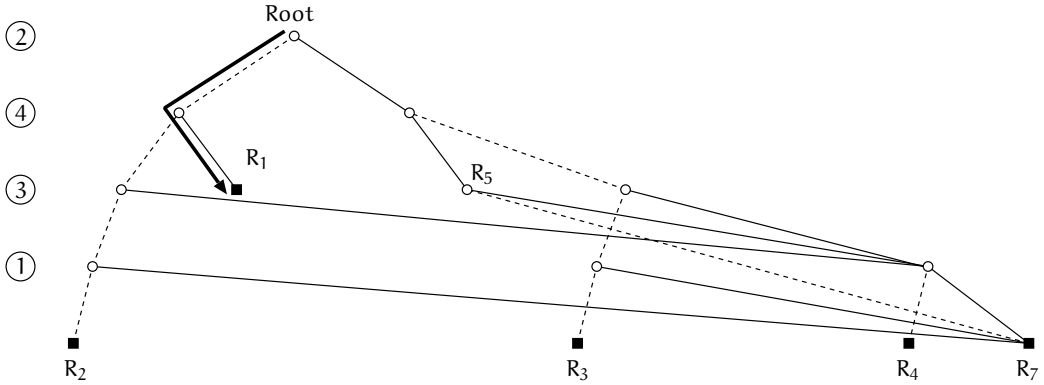


**図 5** 図 4 の不要な節点を削除して構成した RBT

**表 4** ルールリスト 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 ∗ |

と置換すると表 7 のルールリスト $RL_1'$ となり，各ルールの連の数が一つとなる．$RL_2$ は

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 1 & 3 & 4 & 2 & 6 \end{pmatrix}$$

と置換すると表 8 のルールリスト $RL_2'$ となり，各ルールの連の数が一つとなる．

これより列置換を行うためには，与えられたルールリストが表 3,4 のように単一の連から成るルールのリストへ変換できないルールリストなのかを判断するアルゴリズムが求められる．

また表 4 の例のように，与えられたルールリストが単一の連から成るルールのリストへと置換できないと判断できた場合は，k を最小とするような，単一の連から成るルールのリストへと置換できるルールリスト $RL_1, RL_2, \ldots, RL_k$ へと分割するアルゴリズムが求められる．

そして，与えられたルールリストが単一の連から成るルールのリストへと置換できる場合は，単一の連から成るルールのリストへとルールリストを置換するアルゴリズムが求められる．

以上三つのアルゴリズムを開発することが，単一の連か

**表 5** $RL_1$

| | 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 ∗ |

**表 6** $RL_2$

| | 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 ∗ |

**表 7** $RL_1'$

| | 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 ∗ ∗ ∗ ∗ |

**表 8** $RL_2'$

| | 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 ∗ |

らなる RBT を用いるための現時点での問題である．

## 4. 計算機実験

提案した手法の効率を確かめるために C++言語を用いて計算機実験を行った．実験環境は，主記憶容量が 24GB，CPU が Intel Core i7-980X 3.33GHz，OS のカーネルが CentOS Release6.2 である．ルール数が百から一千万までのルールリストとヘッダ数 1000 のヘッダリストをランダムに作成して実験を行った．ルール長は 120 である．

図 6 に示すように，ルール数が一千万でも構築時間は 0.01 秒未満なので構築に時間がかかりすぎるということはない．

図 7 に示すように，ルール数を一千万まで増やしても RBT の節点数は高々 7000 なので提案手法の領域計算量は充分に小さい．

図 8,9 に示すように，提案手法はルール数が増えても探索時間はルール数に依存していない．線型探索の時間はルール数に依存するので線型探索のグラフは右肩上がりになるはずである．実際にグラフが平坦に成ってしまっている原因は，ルールをランダムに生成しているため，ほとんどが * のルールがルールリストの途中に入ってしまって，ルール数を増やしても途中でそのようなルールに入力系列が合致してしまうからだと考えられる．それでも図 8,9 に示すように，線型探索よりも 10 倍速く，比較回数も $\frac{1}{10}$ と提案手法は有効である．

## 5. まとめ

本稿では，単一の連からなる RBT の構成法とその探索法がルール数に依存しないことを示し，提案手法の有効性を計算機実験によりルール数が一千万までのルールリストで確かめた．よって，与えられたルールリストを単一の連からなるルールリストへと変換することができれば，ルール数に依存しない探索が行えることを示した．

本稿で提案した手法は，単一の連からなるルールだけで構成されるルールリストを対象にしている．しかし，本来は任意のビットマスクルールのルールリストで定義された分類問題を扱いたいので，与えられた任意のビットマスクルールのルールリストを単一の連からなるルールのルールリストへと変換する手法，若しくは単一の連からなるルールリスト複数へと分割する手法を提案することが今後の課題である．

### 参考文献

[1] MIKAWA, K. and TANAKA, K.: Run-Based Trie Involving the Structure of Arbitrary Bitmask Rules, *IEICE Transactions on Information and Systems*, Vol. E98.D, No. 6, pp. 1206–1212 (2015).
[2] Taylor, D. E.: Survey and Taxonomy of Packet Classification Techniques, *ACM Comput. Surv.*, Vol. 37, No. 3, pp. 238–275 (2005).
[3] Li, W. and Li, X.: HybridCuts: A Scheme Combining Decomposition and Cutting for Packet Classification, *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pp. 41–48 (2013).
[4] Inoue, T., Mano, T., Mizutani, K., Minato, S. I. and Akashi, O.: Rethinking Packet Classification for Global Network View of Software-Defined Networking, *2014 IEEE 22nd International Conference on Network Protocols*, pp. 296–307 (2014).
[5] 原田崇司，　田中賢，三河賢治：Run-Based Trie から構成される決定木の枝刈り法 (技術と社会・倫理)，電子情報通信学会技術研究報告 = IEICE technical report : 信学技報，Vol. 115, No. 294, pp. 11–17 (2015).
[6] 原田崇司，　田中賢，三河賢治：ポインタ付与による Run-Based Trie 探索の高速化 (回路とシステム)，電子情報通信学会技術研究報告 = IEICE technical report : 信学技報，Vol. 116, No. 315, pp. 13–18 (2016).

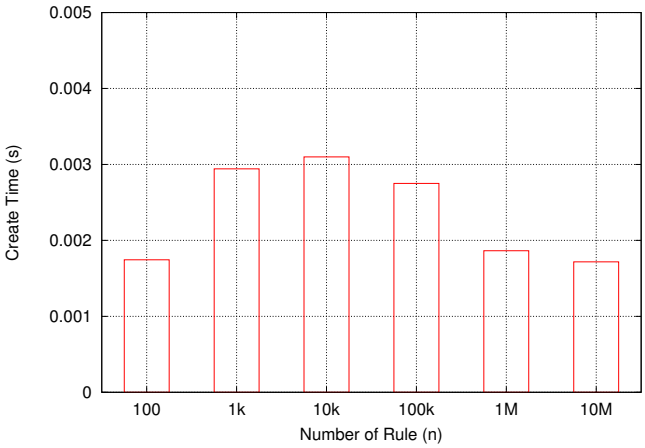


**図 6** 単一の連から成る RBT の構築時間（秒）

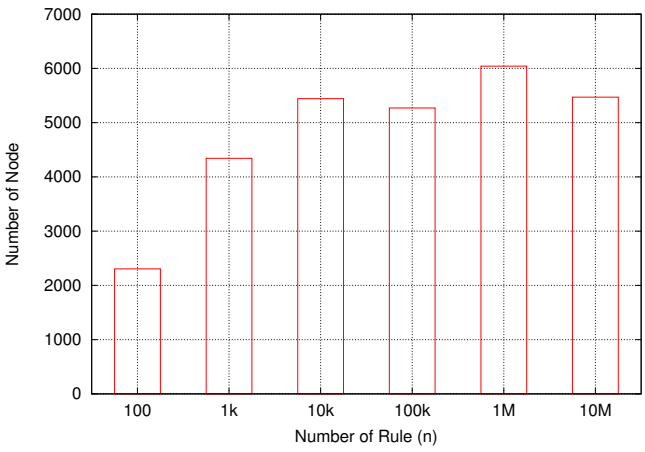


**図 7** 単一の連から成る RBT の節点数

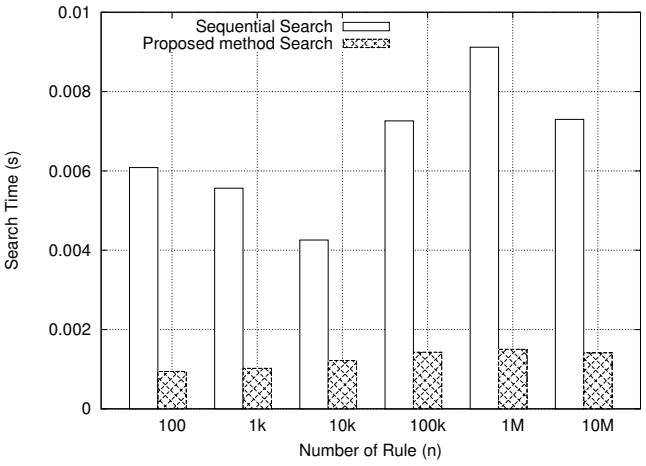


**図 8** 探索時間（秒）

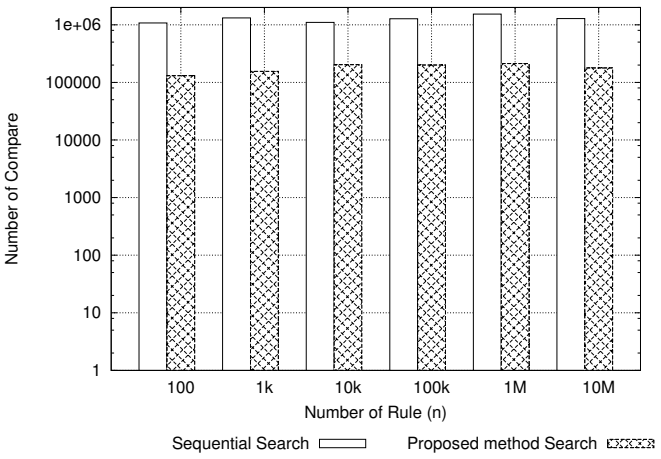


**図 9** 照合回数