文法圧縮を用いた類似度計算の大規模データベースへの適用

Application to a big database of similarity calculation using the grammar compression

大西　孝典 [1*]　久保山　哲二 [2]　坂本　比呂志 [1]
Takanori Onishi[1]　Tetsuji Kuboyama[2]　Hiroshi Sakamoto[2]

[1] 九州工業大学 情報工学府
[1] Graduate Shool of Computer Systems Engineering,Kyushu Institute of Technology,Japan
[2] 学習院大学計算機センター
[2] Computer Center,Gakushuin University

Abstract: We implement application to a big database of similarity calculation using the grammar compression,and confirm the performance.

1　はじめに

本研究では，大規模データベースにおいて，文章の剽窃検出や DNA の解析，データサイズの大きなリポジトリにおけるソースコードの剽窃検出を可能にする．今回はオンライン文法圧縮法を用いて頻出パターンを発見し，さらに領域を削減するために，定数領域で頻度計算を行うアルゴリズムを用いることによって，高速かつ領域を削減した上でデータサイズの大きなリポジトリにおける剽窃検出を可能にしている．

文法圧縮とは入力された文字列からデータの共通する規則性を生成規則として記憶し，冗長な部分を排除して文字列を導出する CFG を構築する圧縮法である．

繰り返しの多いテキストにおいては，この文法圧縮法は有効である事ことが示されているが，今回は大規模なソースコードのリポジトリにおいて文法圧縮法を用いることによって，実際にソースコードの剽窃検出が可能であるのか，またその剽窃のとれる範囲に着目をし実験を行った．

本論文は次のような構成になっている．第２章で ESP[1] について述べ，次に，今回用いる FOLCA[2] というオンライン文法圧縮法と Simple Algorithm[3] について述べる．また，第３章で実験に用いたアルゴリズムについて，第４章で実際に大規模データベースを用いて提案手法を行った際の実行手順と結果について述べる．

*連絡先：九州工業大学　情報工学府　先端情報工学専攻
〒 820-0067　福岡県飯塚市川津 409-1
E-mail: t_onishi@donald.ai.kyutech.ac.jp

2　準備

2.1　ESP について

ESP(Edit Sensitive Parsing)[1] とは文字列分割法兼構文木構築法のことで，文字列を長さ２または３の部分文字列に分解して２，３分木を構築する．図１に実際の構築を示した．同じ文字列をそれより短い $O(\lg^* n \log n)$ の同じ部分木の列によって符号化して，バランスした同じ高さの２，３分木を構築することによって，同じ文字列をその出現位置に関わらず同じノード番号で表すことを可能にする．

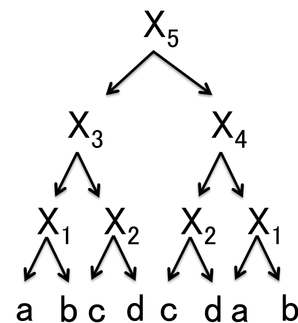


図 1: ESP を用いて構築した構文木の例

2.2　FOLCA について

次に，オンライン文法圧縮法である FOLCA(Fully-OnlineLCA)[2] について述べる．FOLCA は，同じ文

字列をそれより短い $O(\lg^* n \log n)$ の同じ部分木の列によって符号化して，バランスした同じ高さの二分木を構築する文字列分割法兼構文木構築法である ESP[1] と同等の性質を持ち，その上でオンラインで構文木を構築することを可能としている．FOLCA は同じ文字列に対して，その出現位置に関わらず同じ非終端記号で表すことができるアルゴリズムで，共通する非終端記号の一番大きいものを見つけることにより，パターンの近似発見を行う．今回，共通する部分文字列における展開長最大の非終端記号をコア (core) と呼び，そのコアを探すことによってパターンの近似発見を行う．その例を図 2 に示す．図 2 のように，全ての非終端記号はコアとなりうる可能性を持っており，図 2 においては，X_2，X_4，X_5，X_6 がコアとなる．

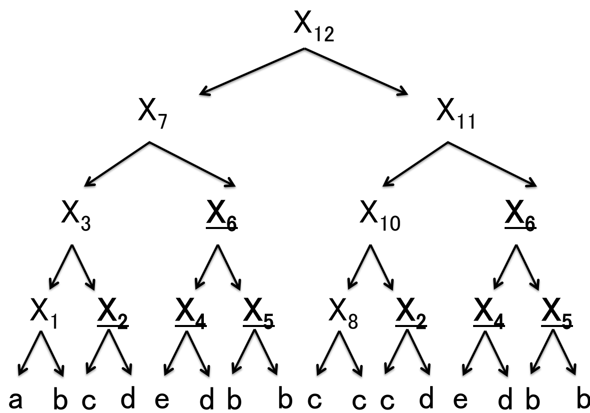


図 2: コアとなりうるノードの例 (X2，X4，X5，X6)

2.3 Simple Algorithm について

次に，定数領域で頻出パターンの近似発見を行うための Simple Algorithm について説明する．Simple Algorithm とは，定数領域で非終端記号の出現回数のカウントを行うアルゴリズムで，出現した非終端記号の長さを N，閾値を $\theta(0 < \theta < 1)$ とした際に，$\frac{1}{\theta}$ 個の非終端記号を最大で保持でき，最終的に θN より大きいものを必ず含んでいるということが保証されている．保持している非終端記号の集合を X とすると，入力される非終端記号のカウントのアルゴリズムは次のようになる．図 3 と図 4 に具体的なカウントの様子を示す．

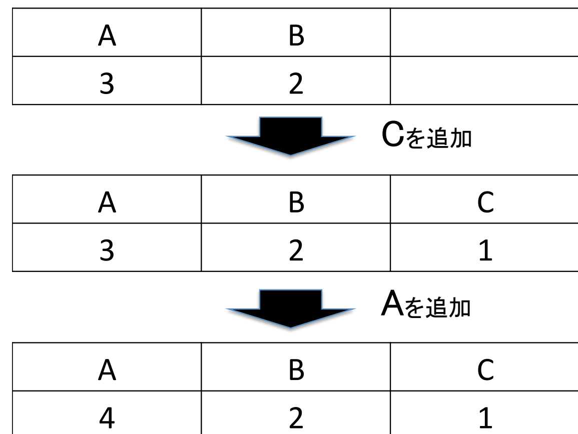
---

if(入力された非終端記号が保持されている)
その非終端記号の頻度を 1 増加させる

else if(入力された非終端記号が保持されていない)

if($|X| = \frac{1}{\theta}$)
存在しないアイテムを入力すべてのアイテムの頻度を 1 減少させて，頻度 が 0 のアイテムを全て削除した後その非終端記号の出現頻度を 1 として新しく X に追加する．

else
X に 出現頻度を 1 として新しく追加する



A	B	
3	2	

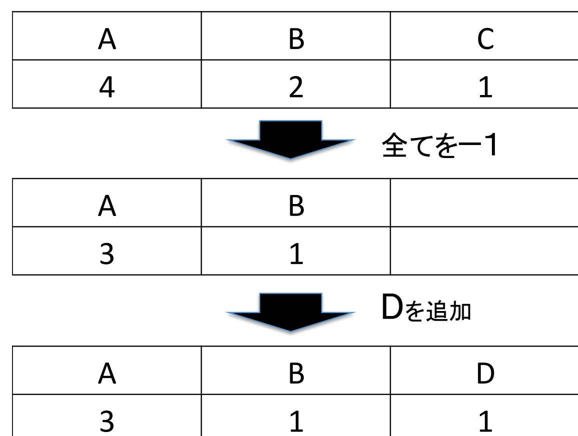
Cを追加

A	B	C
3	2	1

Aを追加

A	B	C
4	2	1

図 3: 非終端記号の追加とカウントの追加



A	B	C
4	2	1

全てを−1

A	B	
3	1	

Dを追加

A	B	D
3	1	1

図 4: 非終端記号の列が埋まっている場合

3 提案手法

今回は，FOLCA を用いて探索を行い，ノード番号と出現頻度を Simple Algorithm を用いて保存する手法によって，大規模データベースにおいて，ソースコードの剽窃の近似発見をするためのアルゴリズムを提案する．今回の実験では，どの非終端記号も core になる可能性があるため，この非終端記号のカウントを行うことによって近似発見を行う．

3.1 アルゴリズム

STEP1

データサイズの大きいソースコードのリポジトリをデータ $Data$ とする．その $Data$ と Simple Algorithm で用いるカウントのための箱 A とある閾値 S を用意しておく．

STEP2

FOLCA を用いて探索を行い，非終端記号のノード番号と出現頻度を Simple Algorithm を用いて A に追加する．

STEP3

A の領域が満たされている場合に新しいノード番号が出てきた場合，全てのノード番号の出現頻度を-1 して，出現頻度が 0 になったものは A から削除し，新しいノード番号と出現頻度を A に追加する．

STEP4

全ての巡回を終えた時，ある閾値以上の文字列で出現頻度が 2 以上のものを表示して，頻出パターンがとれているかを確認する．

4 実験

4.1 実験方法と入力データ

提案手法を用いて実際に実験を行い，ある大規模なデータベースから徐々にデータサイズを増やしたものを用意し，出現した比較的長いパターンの頻度計算を行い，どのようなパターンが出力されているのかを確認した．また，出現回数とパターンにどのような関連性があるのかについても，比較を行った．実験の指標は，比較的長いパターンがとれるかという点とする．ソースコードのリポジトリのファイルサイズを 132MB，540MB,1GB,2.4GB とする．simpleAlgorithm で用いるカウントのための箱を 100000 とし，閾値を $\frac{1}{\theta} = 100 or 1000$ として行う．

4.2 実験結果

結果として，どのファイルサイズにおいてもソースコードの剽窃を見つけ出すことができたが，閾値が $\frac{1}{\theta} = 100$ の場合は，パターンの発見数が莫大となり，ソースコードの剽窃を素早く発見することには不向きであることが分かった．$\frac{1}{\theta} = 1000$ とした場合，パターンの総数が減ったため，ソースコードの剽窃を発見しやすく，例としてデータサイズが 540MB の場合においては，剽窃の文字数が 1000 文字から最大 4500 文字を超えるソースコードの剽窃を発見することができた．その一部分を図 5 に示す．図 5 は一部分であるため，実際はこれより長いパターンを多く検出できた．ソースコードの剽窃の発見と出現回数との関係については，出現回数が 2 or 3 回のノード番号ほどこのように長いソースコードの剽窃がとれているという結果が多く，出現回数が増えるにつれて，とれるパターンも綺麗なソースコードではないものが増える．結果として，繰り返しの少ない大規模なデータにおいても，ソースコードのリポジトリにおける剽窃を表示することができた．

```
nn, err := o.DecodeVarint()
    if err != nil {
        return err
    }
    nb := int(nn) // number of bytes of encoded int64s

    fin := o.index + nb
    if fin < o.index {
        return errOverflow
    }
    for o.index < fin {
        u, err := p.valDec(o)
        if err != nil {
            return err
        }
        v.Append(u)
    }
    return nil
}

// Decode a slice of strings ([]string).
func (o *Buffer) dec_slice_string(p *Properties, base
structPointer) error {
    s, err := o.DecodeStringBytes()
    if err != nil {
        return err
    }
    v := structPointer_StringSlice(base, p.field)
    *v = append(*v, s)
    return nil
}

// Decode a slice of slice of bytes ([][]byte).
func (o *Buffer) dec_slice_slice
```

図 5: 出現パターンの１例

4.3　考察

今回の提案手法においては，大きなデータサイズのリポジトリにおけるソースコードの剽窃検出における様々なパターンを表示することができた．しかし，必要ではない部分のパターンの表示も多く，無駄な出力結果が出る場面が多々見られた．また，実行速度の面でも，データサイズに比例して多くの時間を要するため，実行時間の面から見ても改良が必要であると感じた．これからの課題として，その状況に応じたデータの検出を可能にするために改良することと今回は最大で 2.4GB までの大規模データで実験を行ったが，さらに大きいデータでも剽窃がとれるようにし，実行時間の短縮の面でも改良を加えて研究を進めていきたい．

参考文献

[1] Cormode, G.; Muthukrishnan, S. The String Edit Distance Matching Problem With Moves. ACM Trans. Algor. 2007, 3, 119.

[2] S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane. Fully-Online Grammar Compression. In SPIRE, pages 218-229, 2013.

[3] Richard M. Karp，Scott Shenker and Christos H. Papadimitriou. : A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions on Database System(TODS)* Vol. 28 No. 1， pp. 51-55 (2003).