言語特徴量を用いた正規表現同一判定の効率化

呉 晟董† 趙 文峰† 成 凱†

†九州産業大学 〒813-8503 福岡県福岡市東区 2-3-1

E-mail: †chengk@is.kyusan-u.ac.jp

あらまし 正規表現は文字列のパターンを定義する言語としてデータ統合やテキスト処理等に広く応用されている．しかし，同じパターンを記述するには異なる正規表現が使えるため，理解しやすくて最適な正規表現を求める際に正規表現の同一判定が必要である．正規表現同一判定の従来手法として，正規表現を有限オートマトンに変換し，オートマトンの同一性に基づき正規表現の同一性を判定することが一般的である．しかし，正規表現によって複雑な有限オートマトンになり，同一判定の効率が悪いという問題が指摘されている．本研究では，正規表現の言語特徴量（要素数，文字列の長さ）やマッチングテストを利用し，同一判定の効率化を図ることを提案する．正規表現の言語特徴量はオートマトンの状態数や記号列のサイズに依存せず，非同一判定の効率化に大きく寄与できる．評価実験により，提案手法の有効性を検証した．

キーワード 正規表現，正規言語，有限オートマトン，テキスト処理，リファクタリング，同一判定

1. はじめに

正規表現は，文字列のパターンを記述する言語の一つであり，文字列の検索や置換，抽出などを行う際の対象の指定などのために用いられる[16][17][18][19]．正規表現は有限オートマトンによって受理できる言語と対応することができる[14][20]．一方，正規表現の自由度が高く，同じパターンを表現するには複数の異なる正規表現を用いることができる．初心者にとっては，ある正規表現が正解と一致するかを判断することが難しい．ソフトウェアにおける正規表現はソースコードの一部とみなすことができるため，リファクタリングの対象になる[10][15]．プログラミング言語によって，サポートする正規表現の書き方が異なり，ソースコードマイグレーションが必要となる際には，異なる正規表現の同一性を保証することが重要である．機能や挙動の同じ正規表現であっても，わかりづらくて保守性を持たない「臭いの悪い」もの(Bad Code Smells)が存在する．それらを特定し，より理解しやすく保守しやすいコードへ変換する，所謂リファクタリングを施す必要がある[8][9]．例えば，数字 3 文字を表すには¥d{3}よりも[0-9]{3}のほうが読みやすく保守しやすい．

正規表現の同一判定に関して様々な先行研究が多く存在するが，処理時間がΣ（アルファベット）の大きさ及び有限オートマトンの状態数に依存しているため，効率が悪い問題が指摘されている[3][7]．

本研究では，正規表現の言語特徴量（要素数，文字列の長さ）やマッチングテストを利用して同一判定の効率化を提案する．正規表現の言語特徴量がオートマトンの状態数や記号列のサイズに依存せず，非同一判定の効率化に寄与できる．提案手法を評価するために，任意の正規表現に対応する言語の最小長さ，最大長さ，要素数を求めるアルゴリズムを開発し，特徴量，マッチングテストによる同一判定を Perl で実装し，評価実験を行う．

2. 関連研究

正規表現の同一判定は計算理論の基本的問題として研究されている．まず，正規表現と有限オートマトンの関係に基づき，同一判定を行う手法が最もよく利用される．中では，最小確定有限オートマトン（DFA）の唯一性により，正規表現に対応する最小 DFA を求め，同一判定を行う方法が提案された[2]．しかし，k=|Σ|，状態数 n のオートマトンの最小化に最悪計算量が$O(kn\log n)$であり，計算コストが高いと知られている．これに対して，Hopcroft と Karp らは状態の同値性に基づき，最小化を必要としない同一判定アルゴリズム（HK アルゴリズム）が提案された[3]．HK アルゴリズムは同じΣ上で状態数がそれぞれ n と m の DFA の同一判定に必要な計算量が最悪$O(k(n+m))$であり，Σの大きさと状態数に依存している．

オートマトンを使わない同一判定アルゴリズムについて研究されていた．M. Almeid ら[7]は正規表現の代数的性質を利用した同一判定を行う手法を提案した．この手法は Rewrite 公理[4] [5][6]に基づいているが，Σの大きさに依存するため，性能面では上記の有限オートマトン法と変わらない．

C. Chapman ら[8][9]は，正規表現の分かりやすさがソフトウェアの保守性にどのような影響を与えるかについて調べた．Github 上のプロジェクトにおいて正規表現の望ましい (least smelly) 形式を調べた．[9]では類似する正規表現をクラスタリングするため，本研究でのマッチングテストに近い方法で，正規表現の類似度を計算した．例えば，正規表現 A と B の類似度を計算するため，パターン A の文字列 100 個のうち，B に

マッチするものが 90 個とする場合は，B が A に 90% 類似する．逆にパターン B の文字列 100 個のうち，A にマッチするのが 50 個しかない場合は，A が B に 50% 類似という．類似度スコア行列を用いてクラスタリングを行い，正規表現クラスタの特徴を分析することにより，よく使われる正規表現を特定でき，プログラマに推奨することができる．

3. 正規表現の同一判定

本節は，正規表現とその同一判定の基本的事項について述べる．

3.1. 正規表現とその同一性

正規表現は，ある記号集合（アルファベット）上で定義される文字列のパターンを定義する言語である．文字列のパターンマッチングによく使われる．アルファベットΣ上の正規表現はメタ文字と通常文字に分けている．表 1 はよく使われるメタ文字を列挙している．

表 1 正規表現のメタ文字

記号	説明	記号	説明
.	任意 1 文字	*	0 回以上繰り返し
[ ]	[]中任意 1 文字	+	1 回以上繰り返し
[^]	[]外任意 1 文字	?	0 回か 1 回出現
¥d	数字 1 文字	{n}	n 回繰り返し
¥w	英数 1 文字	{n,m}	n 回以上 m 回以下繰り返し

正規表現で表すことができる言語は正則言語とも呼ばれ，有限オートマトン（DFA，NFA）によって受理できる言語と等しい[1][14]．正規表現 α に対応する正規言語は$L(\alpha)$と表す．正規表現の同一性は以下のように定義する．

定義：$L(\alpha) = L(\beta)$であれば，正規表現 α と β が同一正規表現という．

α と β が形式上で違っても同じ正規言語を表すものであれば，お互いに同一である．例えば，言語$L = \{aaa, aab, aba, abb, bbb, bba, bab, baa\}$に対応する正規表現$[ab]\{3\}$と$(a|b)\{3\}$は同一の正規表現である．

3.2. オートマトンに基づく同一判定

正規表現 α と β に対する正則言語$L(\alpha)$と$L(\beta)$を受理する有限オートマトンの同一性によって，α と β の同一性を判定する手法について述べる[14]．まず，有限オートマトンの状態の同値性を説明する．

ある有限オートマトンにおいて，状態pとqが全ての入力列wに対して，$\hat{\delta}(p,w)$が受理状態で$\hat{\delta}(q,w)$も受理状態あるときにpとqは**同値**である．ここの$\hat{\delta}(p,w)$は状態 p で入力列 w を受けてその遷移先の状態を求める状態遷移関数である．つまり，状態pとqが同値であるとは，これらのうちの一方を初期状態として与えられた入力列を読み進むとき受理状態に到達するかどうかを調べるだけでは，両者を区別することは不可能である．二つの状態が同値でないとき，その二つの状態は区別可能であるという．すなわち，状態pが状態qから区別可能であるのは$\hat{\delta}(p,w)$と$\hat{\delta}(q,w)$のうち一方は受理状態であるが，もう一方は受理状態ではない，となるような文字列wが少なくとも一つ存在することである．

同値な状態を全て見つけるには，区別可能な状態の対を可能な限り見つけ出せば良い．状態の同値を調べるには穴埋めアルゴリズムと呼ばれるアルゴリズムが提案されている[3][14]．これは DFA $A = (Q, \Sigma, \delta, q_0, F)$における区別可能な対を再帰的に発見していくアルゴリズムである．

1. **基礎**：状態pが受理状態であり，状態qが受理状態でないならば対$\{p,q\}$は区別可能である．
2. **再帰**：状態pとqは，ある入力文字aに対し$r = \delta(p,a)$，$s = \delta(q,a)$であり，しかも$\{r,s\}$が区別可能であれば，p,qも区別可能である．

この判定が正しいことは以下の理由による．rとsが区別可能であることから，$\hat{\delta}(r,w)$と$\hat{\delta}(s,w)$のうちの一方のみが受理状態である文字列 w が存在する．そうすると，文字列awはpとqを区別する．$\{\hat{\delta}(p,aw), \hat{\delta}(q,aw)\}$は$\{\hat{\delta}(r,w), \hat{\delta}(s,w)\}$と同じ状態の対になるからである．

穴埋めアルゴリズムを用いることにより，二つの正規表現が同じ言語であるかどうかを判定する簡単な方法が得られる．言語 L と M が，二つの正規表現によって与えられているとする．まず，それぞれの表現を DFA に変換する．その上で，L を受理する DFA の状態集合と M を受理する DFA の状態集合の集合和を状態集合として持つもう一つの DFA を想像する．定義上この DFA は二つの開始状態を持つことになるが，状態の同値性の判定が考察の対象となっている限りでは，開始状態の役割は不要である．任意の状態を開始状態として定めておいても差し支えない．

次は元の DFA で定義されていた開始状態同士について，それらが同値であるかどうかを，穴埋めアルゴリズムを用いて調べる．それらが同値であれば$L = M$であり，同値でないならば$L \neq M$である

状態の同値性を利用して，DFA の状態数を最小化することができ，最小 DFA を求めることができる．同一正規表現の最小 DFA が唯一であるため，正規表現の同一判定に利用できる．

4. 正規表現同一判定の効率化

4.1. 言語特徴量

正規表現の言語特徴量とは，正規表現が他の正規表現と区別する性質のことである．表 2 では本研究で使

われる特徴量をまとめている．

要素数を$\#(\alpha)$で表し，文字列の最小長さを$|\alpha|_{min}$，最大値を$|\alpha|_{max}$で表す．要素数とは文字列の集合$L(\alpha)$の要素の数である．文字列の最小長さと最大長さは，文字列の集合の要素の中で最も短い要素の文字列の長さと最も長い文字列の長さである．

表 2 正規表現の言語特徴量

記号	説明
$\|\alpha\|_{min}$	$L(\alpha)$の文字列の最小長さ
$\|\alpha\|_{max}$	$L(\alpha)$の文字列の最大長さ
$\#(\alpha)$	言語$L(\alpha)$の要素数，$\#(\alpha) = \|L(\alpha)\|$

定理：同じΣ上の二つの正規表現αとβが同一であれば，下記の等式が成立する．

① $\#(\alpha) = \#(\beta)$

② $|\alpha|_{min} = |\beta|_{min}$

③ $|\alpha|_{max} = |\beta|_{max}$

④ 任意の$x \in L(\alpha)$であるとき，$x \in L(\beta)$である，逆に任意の$x \in L(\beta)$であるとき，$x \in L(\alpha)$である

4.2. 言語特徴量の求め方

規表現の特徴量で同一判定をする前に特徴量を計算する必要がある．特徴量は構文木を用いて計算することができる．構文木とは構文解析の経過や結果を木構造で表したものである．木構造の各ノードは枝と葉に分かれ，葉のノードは値が必要である．枝のノードでは演算子とオペランドが必要である．オペランドは別の枝と葉のノードになる．正規表現の言語特徴量の計算に主に使われているオペレータは表 3 のようになる．

表 3 特徴量の計算に使われているオペレータ

正規表現	意味	構文木表現
|	また	union
	連結	concat
[ ]	括弧内の 1 文字	anyof
{n} {n,m}	n 回繰り返し n 回以上 m 回以下	quant range, min, max
*	0 回以上繰り返し	star
+	1 回以上繰り返し	plus

4.2.1. 言語の要素数の求め方

まずは構文木を用いて要素数を計算する方法から説明する．要素数を計算するとき，主に使われている記号の計算方法は下記の数式のようになる．

① anyof の計算

$\#([\ldots]) = |[\ldots]|$

② union の計算

$\#(\alpha|\beta) = \#(\alpha) + \#(\beta)$

この計算式は$L(\alpha)$と$L(\beta)$の間に共通部分が存在しないときに有効である．「a[bc]|ac」のように共通部分が空でない場合は，正しく計算できない．

③ concat の計算

$\#(\alpha\beta) = \#(\alpha) \times \#(\beta)$

④ quant の計算

$\#(\alpha\{m\}) = \#(\alpha)^m$

$\#(\alpha\{m,n\}) = \#(\alpha)^m + \#(\alpha)^{m+1} + \cdots + \#(\alpha)^n$

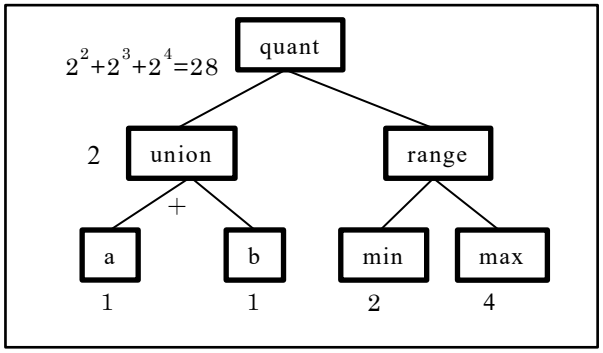

図 1 構文木における正規表現要素数の計算例

まず言語の要素数の求め方を例によって説明する．

① 正規表現$\alpha = [a-c]\{2\}$であるとき

$L(\alpha) = \{aa, ab, ac, bb, ba, bc, ca, cb, cc\}$

$\#(\alpha) = |L(\alpha)| = 3^2 = 9$

② 正規表現$\alpha = [a-c][ab]$であるとき

$L(\alpha) = \{aa, ab, ba, bb, ca, cb\}$

$\#(\alpha) = |L(\alpha)| = 3 \times 2 = 6$

③ 正規表現$\alpha = [ab]\{3\}|[bc]\{2\}$であるとき

$L(\alpha) = \{aaa, aab, \cdots, bbb, bb, bc, cc, cb\}$

$\#(\alpha) = |L(\alpha)| = 2^3 + 2^2 = 12$

④ 正規表現$\alpha = a*$であるとき

$L(\alpha) = \{\varepsilon, a, aa, aaa \cdots\}$

$\#(\alpha) = |L(\alpha)| = \infty$

4.2.2. 文字列の長さの求め方

次は構文木を用いて文字列の長さを計算する方法である．文字列の長さを計算するとき，主に使われている記号の計算方法は下記の数式のようになる．

① anyof の計算

$|[\cdots]|_{min} = |[\cdots]|_{max} = 1$

② union の計算

$|\alpha|\beta|_{min} = \min(|\alpha|_{min}, |\beta|_{min})$
$|\alpha|\beta|_{max} = \max(|\alpha|_{max}, |\beta|_{max})$

③ concat の計算

$|\alpha\beta|_{min} = |\alpha|_{min} + |\beta|_{min}$
$|\alpha\beta|_{max} = |\alpha|_{max} + |\beta|_{max}$

④ quant の計算

$|\alpha\{n\}|_{min} = n \times |\alpha|_{min}$
$|\alpha\{n\}|_{max} = n \times |\alpha|_{max}$
$|\alpha\{m,n\}|_{min} = n \times |\alpha|_{min},$
$|\alpha\{m,n\}|_{max} = m \times |\alpha|_{max}$

⑤ star の計算

$|\alpha*|_{min} = 0, |(\alpha)*|_{max} = \infty$

⑥ plus の計算

$|\alpha+|_{min} = |\alpha|_{min}, |\alpha+|_{max} = \infty$

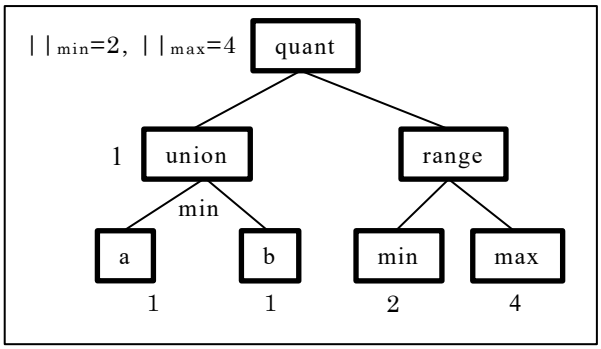

図 2 構文木における文字列長さの計算例

次は文字列の長さの求め方を例によって説明する.

① 正規表現$\alpha = [a-c]\{2\}$であるとき

$L(\alpha) = \{aa, ab, ac, bb, ba, bc, ca, cb, cc\}$
$|\alpha|_{min} = 2,\ |\alpha|_{max} = 2$

② 正規表現$\alpha = [ab]\{2,3\}$であるとき

$L(\alpha) = \{aa, ab, ba, bb, aaa, aab, \cdots, baa, bab\}$
$|\alpha|_{min} = 2,\ |\alpha|_{max} = 3$

③ 正規表現$\alpha = [ab]\{3\}|[bc]\{2\}$であるとき

$L(\alpha) = \{aaa, aab, \cdots, bbb, bb, bc, cc, cb\}$
$|\alpha|_{min} = \min(|[ab]\{3\}|_{min}, |[bc]\{2\}|_{min}) = 2$
$|\alpha|_{max} = \max(|[ab]\{3\}|_{max}, |[bc]\{2\}|_{max}) = 3$

④ 正規表現$\alpha = a*$であるとき

$L(\alpha) = \{\varepsilon, a, aa, aaa \cdots\}$
$|\alpha|_{min} = 0,\ |\alpha|_{max} = \infty$

4.3. マッチングテストによる同一判定

マッチングテストによる同一判定である．正規表現αとβを与えられたとき，αにマッチする文字列（以降「αの文字列」とする）をテストデータして生成して，βにマッチするかをテストする．マッチしなければ同一ではないと判定する．すべてマッチした場合に，次はβの文字列を生成し，αにマッチするかをテストする．マッチしなければ，同一ではないと判定する．

入力：正規表現αとβ, テストに必要な文字列数 k
出力：αとβの同一判定の結果（True/False）
手順：

1. αの文字列をランダムに k 個生成
2. 生成された文字列がβにマッチするかをテストし，マッチしなければ，False を返して終了
3. βの文字列をランダムに k 個生成
4. 生成された文字列がαにマッチするかをテストし，マッチしなければ，False を返して終了
5. ここまで判断できなければ True を返して終了

リスト 1 マッチングテストによる同一判定

要素数 n の二つの正規表現 α と β の言語を L(α)と L(β)とする．L(α)と L(β)の要素の中で共通の要素の数を m とする($m \le n$)．このとき，False Positive，つまり，同一ではないが同一と判定される確率が$p = (m/n)^k$である．k が大きくなるほど確率 p は下がるため，同一ではない正規表現を非同一と正しく判定する確率$1-p$が大きくなる．正規表現が同一ではない場合も確実に判定でき，しかも，判定する時間が既存の方法よりは短い．

しかし，この方法は，α と β のすべての文字列を使ってテストしない限り，テストデータがすべてマッチしたとしても，α と β は必ずしも同一と判定することができない．この場合は，ほかの方法，例えば穴埋めアルゴリズムによる判定を行う必要がある．

5. 評価実験

本節では，提案の正規表現の同一判定手法を評価するための実験方法と実験結果について述べる．

5.1. 実験方法

提案手法は非同一である場合に素早く判定でき，オートマトンによる同一判定を実行しなくて済むが，最終判定ができないケースもある．提案手法と既存のオートマトンによる同一判定と組み合わせで実験を行う必要がある．まず，オートマトンによる同一判定のみ実行し，時間を測る．次はオートマトンによる同一判定を実行する前に提案の方法を入れて，時間を測り，時間が節約できて効率化が実現できるかどうかを評価する．

実験に使う正規表現は五つのセットを用意する．(A)正規表現セット，(B) 同一正規表現セット，(C) 四割非同一正規表現セット，(D)六割非同一正規表現セットと (E) 非同一正規表現セットである．各セットの中には事前に用意した正規表現が入っている．この五つのセットを使って同一判定を行う．

作った正規表現の要素数は多くないので，今回はマッチングテストによる同一判定で生成する文字列数 k=5 にして実験を行う．正規表現セット A と他の三つのセットから一セットずつ同一判定をして時間を測る．同一判定はオートマトンによる同一判定のみ実行し，提案方法とオートマトンによる同一判定を一つずつ合わせて実行して，最後は四つの方法とオートマトンによる同一判定を合わせて実行する．実験をするために使う変数について説明する．

表 4 パラメータの説明

パラメータ	意味	値
loop	設定した同一判定の繰り返し回数	100
step	繰り返し回数の増加量	100
max_loop	最大繰り返し回数	1000
config	最小長さ，最大長さ，要素数，マッチングテストの実行制御	1：実行 0:実行しない

ここで loop は設定した同一判定の繰り返し回数である．例えば config が(0,0,0,1)の場合はマッチングテストによる同一判定をした後オートマトンによる同一判定をするが，このセットを繰り返す意味である．最初は設定した同一判定の回数は 100 回から始め，次から回数を 100 ずつ増やしながら，最大 900 回まで同一判定を行う．

また，オートマトンによる同一判定を行う前に，採用される言語特徴量は Config で指定する．Config＝(最小長さ，最大長さ，要素数，マッチングテスト)，つまり，Config の括弧の値は，それぞれ，文字列の最小長さによる同一判定，文字列の最大長さによる同一判定，言語の要素数による同一判定，マッチングテストによる同一判定を実行するかどうかを表している．つまり

・(0,0,0,0)：言語特徴量を使用しない
・(1,0,0,0)：最小長さのみ使う
・(0,1,0,0)：最大長さのみ使う
・(0,0,1,0)：要素数のみ使う
・(0,0,0,1)：マッチングテストのみ使う
・(1,1,1,1)：すべての言語特徴量を使う

5.2. 実験結果

実験用プログラムは Perl で開発している．正規表現の構文解析に Regexp::Parser[11]が使われた．マッチングテストでは String::Random[12]というパッケージを使って正規表現からランダムに文字列を生成している．オートマトンによる正規表現同一判定に Regex::EXE というパッケージを使用した[13]．主な機能として

① ere_to_nfa：正規表現の NFA を作成する．
② nfa_to_dfa：NFA を DFA に変換する．
③ dfa_to_min_dfa：DFA の最小化を行う．
④ nfa_isomorph:最小化 DFA の同一性を判定する．

使用するパッケージにより，サポートする正規表現に多少違いがあるため，共通する正規表現に限定して実験を行った．

実験結果は図 3～図 6 に示している．まず，同一正規表現セット B と元正規表現セット A の同一判定の実験を行った．図 3 では同一正規表現セットの場合の結果を比較した結果を示している．

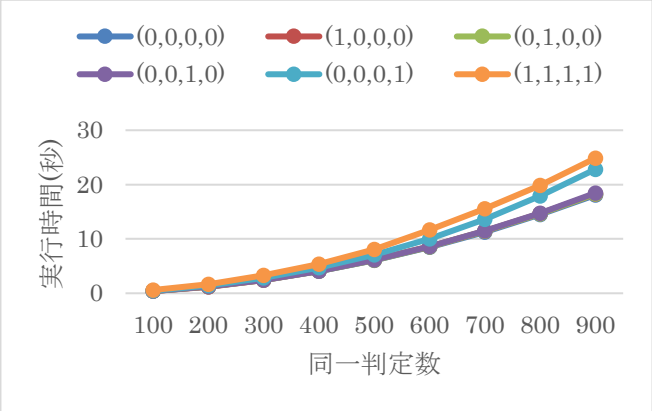

図 3 同一正規表現の評価結果

特徴量による判定は同一という最終判断ができないため，オートマトンによる同一判定のみ実行する場合とほぼ同じであるが，特徴量計算に少しだけコストが増えている．

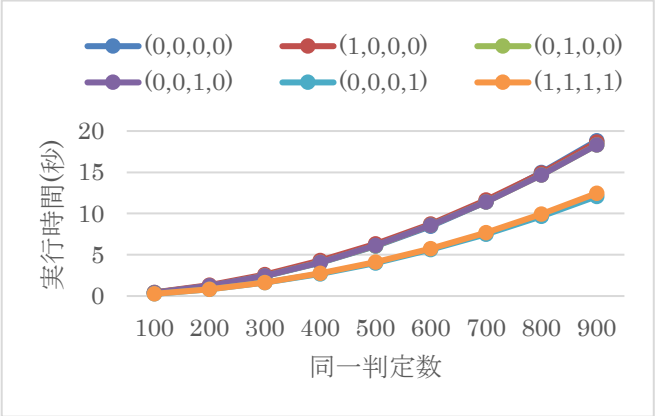

図 4 部分（40%）非同一正規表現の評価結果

部分同一正規表現セットの実験結果は図 4 と図 5 で示している.図 4 は 40%非同一の正規表現セットの実験結果であり，図 5 は 60%非同一の正規表現セットの実験結果である．非同一部分の割合が増えることにつて，特徴量による非同一の効果が顕著となり，とりわけ，マッチングテストのほうが寄与するところが一番大きいことも確認できた．

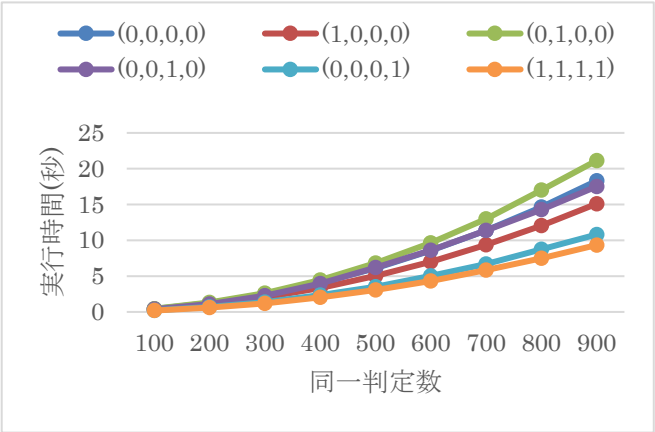

図 5 部分（60%）非同一正規表現の評価結果

最後に 100%非同一正規表現セット E と元正規表現セット A の同一判定の実験を行った.図 6 は非同一正規表現セットの場合の結果を比較した結果を示している．非同一正規表現セットを使う場合は，非同一と判定されたら，オートマトンによる同一判定が実行せずに済むため，実行時間が明らかに短くなっている．特に 4 つの方法を合わせて実行する場合は最も短くなっており，提案方法が有効であることを示している

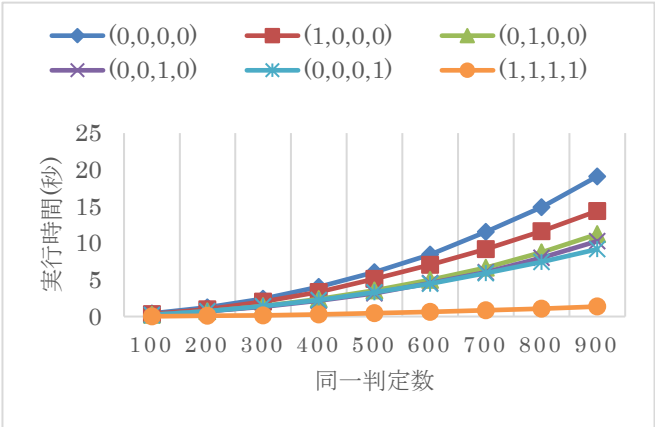

図 6 非同一正規表現の評価結果

6. おわりに

正規表現は書き方が違っても実質同じであることがありうる．正規表現の同一判定のため，既存手法ではオートマトンを利用することが一般的であるが，状態数やΣの大きさに依存しており計算コストが高い問題がある．本研究では，正規表現の言語特徴量による同一判定,マッチングテストによる同一判定を提案し，同一判定の効率化を図った．

正規表現の言語特徴量がオートマトンの状態数や記号列のサイズに依存せず，非同一判定の効率化に寄与できる．提案手法を評価するために，任意の正規表現に対応する言語の最小長さ，最大長さ，要素数を求めるアルゴリズムを開発し，特徴量，マッチングテストによる同一判定を Perl で実装し,評価実験を行った．その結果，非同一の割合が高ければ，有限オートマトンを使わず，判定でき，効率化を確認できた．

今後の課題として，マッチングテストにおいて，特徴量に基づき，必要な文字列数 k の最適化や，実験データの増加などが挙げられる．

謝辞 本研究は令和 3 年度 KSU 基盤研究費による支援を頂いており，ここで謹んで感謝の意を表する．また，発表会で座長やコメンテータの方から大変有益なコメントをいただき，今後の参考になった．

参 考 文 献

[1] J.E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages and Computation. Addison Wesley (2000).

[2] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical Report. Stanford University, Stanford, CA, USA. (1971)

[3] J. E. Hopcroft and R. M. Karp, A linear algorithm for testing equivalence of finite automata, Technical Report 71-114, University of California (1971).

[4] V. M. Antimirov, P. D. Mosses, Rewriting extended

regular expressions. In: G. Rozenberg and A. Salomaa, editors: Developments in Language Theory, World Scientific, pp.195-209(1994).

[5] V. M. Antimirov, Partial derivatives of regular expressions and finite automaton constructions. Theoret. Comput. Sci.155, pp.291-319 (1996).

[6] M. Almeida, N. Moreira, R. Reis, Antimirov and Mosses's rewrite system revisited. In: O. Ibarraa & B. Ravikumar, editors: CIAA 2008, LNCS 5448, Springer-Verlag, pp.46-56 (2008).

[7] M. Almeida, N. Moreira, R. Reis, Testing the Equivalence of Regular Languages, Journal of Automata, Languages and Combinatorics 15 (2010) 1/2, pp.7-25.

[8] C. Chapman, P. Wang and K. T. Stolee, Exploring regular expression comprehension, In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 405-416, doi: 10.1109/ASE.2017.8115653.

[9] C. Chapman and K. T. Stolee. Exploring regular expression usage and context in Python. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). ACM, pp.282–293. doi.org/10.1145/2931037.2931073

[10] M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd Edition, 2018, Addison-Wesley. ISBN 978-0-134-75759-9.

[11] Todd Rinald, Regexp::Parser - base class for parsing regexes (2020.12), https://metacpan.org/pod/Regexp::Parser

[12] String::Random-Perl module to generate random strings based on a pattern, (2020.12), https://metacpan.org/pod/String::Random

[13] Regexp::ERE-extended regular expressions and finite automata, (2020.12), https://metacpan.org/pod/Regexp::ERE

[14] J.ホップクロフト, R.モトワニ, J.ウルマン, オートマトン言語理論 計算論 I [第 2 版], サイエンス社, 2003.

[15] 雜賀 翼, 崔 恩瀞, 吉田 則裕, 春名 修介, 井上 克郎, Code Smell の深刻度がリファクタリングに与える影響の調査, ソフトウェアエンジニアリングシンポジウム 2015 論文集, pp.176-181

[16] 杉山 貴章, 反復学習ソフト付き正規表現書き方ドリル, 技術評論社(2011).

[17] 宮前 竜也, [改訂新版]正規表現ポケットリファレンス, 技術評論社, 2015.

[18] 新屋 良磨, 鈴木 勇介, 高田 謙, 正規表現技術入門, 技術評論社, 2015.

[19] 佐藤 竜一, 正規表現辞典 改訂新版, 翔泳社, 2018

[20] 小倉 久和, 形式言語と有限オートマトン入門, コロナ社, 1996.

[21] Jeffrey E. F. Friedl, 詳説 正規表現 第 3 版, オライリー・ジャパン, 2008.