SIGURE Hash: 数式検索のための高速な類似検索アルゴリズム

大橋　駿介[†]　　相澤　彰子[††,†]

[†] 東京大学大学院 情報理工学系研究科　〒113–8656 東京都文京区本郷 7–3–1
[††] 国立情報学研究所　〒101–8430 東京都千代田区一ツ橋 2–1–2
E-mail: †{sohashi,aizawa}@nii.ac.jp

あらまし　数式は科学技術を厳密に記述する言語として幅広く用いられている。数式は一般的に木構造に変換可能な MathML や LaTeX といった形式で記述されているが、このような数式データに対する情報検索技術は十分とはいえない。例えば、$a^2 + b^2 = c^2$ と $x^2 + y^2 = z^2$ のように同義ではあるが表記が異なる数式が存在するが、これは例えば一般的な木構造類似検索アルゴリズムでは検索できない。そこで、本研究においては数式の変数名の書き換えに対して不変な類似度計算アルゴリズムを開発し、このような表記の異なる数式を検索できるようにした。また、このアルゴリズムは入力の大きさに対してほとんど線形時間で動作するため、高いスケーラビリティを持つ。大規模な数式データセットを用いた実験により、我々のアルゴリズムが既存の木構造類似検索アルゴリズムである pq-gram と比較して高速に索引を作成でき、検索結果の精度も向上していることが示された。

キーワード　数式検索，類似検索，MathML

1. はじめに

数式は科学技術のコミュニケーションにおいて幅広く用いられている記述言語である。数式は例えば論文や研究報告の形で専門家同士の相互理解に用いられるばかりでなく、数学や理科の教科書といった形で教育においても必要不可欠な存在となっている。近年の情報化の進展により数式の情報も増大しており、大量の数式から目的の情報をすばやく取り出す数式検索の技術は専門家・非専門家を問わず大きな需要がある。しかしその一方で、現在よく用いられている Web 検索エンジンは自然言語で記述された文書を主な対象として実装されており、記号列である数式をうまく検索することはできない。木構造類似検索やそれを用いた XML 文書検索の研究も進められており、これを数式記述のための XML である MathML に適用するという方法も考えられるが、数式そのものが持つ性質は考慮されないため検索の精度には問題が残る。例えば、コーシー・シュワルツの不等式 $|u \cdot v| \leqq ||u||||v||$ に登場する変数 u, v は名前を自由に書き換えてもその数学的な本質は変化しない。ところが既存の木構造類似検索はラベルの変化に対して鋭敏であり、変数名が書き換わるとうまく検索することはできない。このような現状を踏まえ、本研究では数式の変数名の書き換えを考慮した類似度計算アルゴリズムを提案し、これを実際に実装して大規模なデータセットを用いた実験を通じて評価した。

以下、本稿では 2. 節で関連研究を紹介し、3. 節で変数名の書き換えを考慮した数式の類似度計算アルゴリズムを提案する。4. 節で既存の木構造類似検索アルゴリズムと比較して提案手法を評価する。5. 節で結論を述べる。

2. 関連研究

2.1 XML 検索

数式データは多くの場合 LaTeX 形式で記録されているが、これは LaTeXML (注1) を用いて数式を記述する XML の一種である MathML (注2) に自動的に変換することができる。XML を木構造とみなして類似検索を行う手法をいくつか紹介する。Tree Edit Distance (TED) [7] は文字列の編集距離を拡張したものであり、ある木から頂点の挿入・削除もしくは頂点のラベルの書き換えの 3 種類の操作をできるだけ少ない回数行って別に木に書き換えたときの操作回数を 2 つの木の距離とするものである。ところが、TED の時間計算量は現在知られている最も高速なものでも木の頂点数を n として $O(n^3)$ [4] と次数が大きく、大きな木に対して何回も計算する必要がある XML 類似検索にそのまま適用するのは難しい。pq-gram [2] は、編集操作にある重みを付けたときの TED の下限という形で TED への近似を与える類似度計算アルゴリズムである。いったん木を特徴集合に変換してから集合の類似度を元の木の類似度とする手法であり、特徴集合への変換と集合の類似度計算がそれぞれ $O(n), O(n \log n)$ 時間で可能である。MinHash [3] は集合の類似度として用いられるジャカード係数の計算を高速化する手法である。pq-gram と MinHash を組み合わせて XML 検索に応用した研究 [8] も存在する。これらの木構造類似検索アルゴリズムを用いることで構造の類似性を踏まえた XML 類似検索が可能であるが、構造や表記には直接反映されない数式の意味的な類似性を捉えることは難しい。

2.2 数式検索

Kamali ら [5] は数式検索の手法を分類し、それらを実験により比較した。これによると、主な手法を分類すると、クエリと完全に一致するものを取り出す完全マッチングと多少の表記ゆれなどを許す類似マッチングの 2 つに大きく分けられ、類似マッチングについてはさらに共有する部分構造の多さを類似度

(注1)：http://dlmf.nist.gov/LaTeXML/
(注2)：http://www.w3.org/Math/

とする部分構造マッチングとクエリにワイルドカードなどを許したパターンマッチングに分類できる。厳密マッチングは正確だが取り出せる式の数が非常に少ない。類似マッチングの 2 つの比較ではパターンマッチングは部分構造マッチングに比べてややよい精度を示したが、パターンマッチングのためのクエリをユーザーが構成するのは手間がかかるとし、一般的なケースでは部分構造マッチングによる手法がもっともよいと報告されている。本稿で提案する手法は部分構造マッチングに基づいているが、マッチングの際に適切に変数を考慮する点でパターンマッチングの要素を取り入れている。大橋ら [9] は数式の意味を反映した検索法を提案した。数式処理システム Mathematica を用いて検索対象の数式に予め生成しておいた乱数を代入することで確率的に同値性を判定し、表記が異なるが同値な数式を検索できるようにしたが、検索の対象が数値を代入できるような式に限られたり、そもそも Mathematica で処理できない数式が存在するなどの問題があった。本稿では変数名を正規化してからハッシュすることで変数名について表記が異なる数式が検索可能でありつつ、入力の数式を一般の XML パーサを用いて処理することで MathML で記述されたすべての数式を処理できるようにした。また、数学検索システムの性能比較のために NTCIR-11 Math task [1] において、評価用データセットが構築されている。この評価用データセットを用いた参加システム同士の比較によると、変数名を考慮した検索をするシステムがより高い精度を達成した。本稿では提案手法の評価に際して NTCIR11 Math task のデータセットを使用した。

3. 提案手法

3.1 定義

手法の説明に必要な定義を述べる。数式検索問題は与えられた数式を含む文書集合 (データセット) から、数式として与えられるクエリとの関連度が高いものから順に並べたリストを返すものである。本研究で提案する手法において、あとで説明する通りデータセットやクエリの数式は XML パーサによってまず根・順序・ラベル付き木、すなわち木の頂点には根と呼ばれる頂点を基準とした子孫・先祖関係があり、ある頂点に付いている子には左から右といった順序があり、各頂点には文字列が付いているもの、に変換される。以下、本稿で単に木といった場合には根・順序・ラベル付き木であるとする。数式を MathML で表現する時、もとの数式の演算子・変数・定数の識別子はそれぞれ異なるタグ付けがされる。Presentation MathML ではそれぞれ mo, mi, mn タグが、Content MathML では csymbol, ci, cn タグが用いられる。したがって、あるノードのラベルが変数名であるかどうかはその親ノードのラベル (タグ名) が何であるかを見ればわかる。本稿では親ノードのラベルが ci, mi であるノードを変数ノードと呼ぶこととする。

3.2 変数の性質

数式の変数をそれが表す意味という観点から 2 つの性質を持つことを説明する。変数名には名前それ自身に意味がある場合とそうでない場合がある。たとえば、単に n といった場合には一般に自然数が想定されるし、$F = ma$ といった場合には F, m, a はそれぞれ力、質量、加速度を表現していると解釈される。このような数式においては変数名を書き換えると意味が大きく変わる。例えば $V = RI$ は F, m, a をそれぞれ V, R, I に書き換えただけであるが、一般的にはそれぞれ電圧、電気抵抗、電流を表していると解釈される。このような数式に対しては変数名が書き換わると類似度が低下するような類似尺度が望ましい。一方で、1. 節でも挙げたコーシー・シュワルツの不等式は変数名を書き換えても定理が表す内容は不変である。例えば $|x \cdot y| \leqq ||x||||y||$ と書いても同じことである。これは元の定理が変数の間に成り立つ関係を表現しているからである。このような数式に対しては変数名を書き換えても類似度が変わらないような類似尺度が望ましい。本研究で提案する手法においては、それぞれの種類の変数に対応する類似尺度を提案し、これをともに用いることで両方の数式を検索できるようにした。

3.3 概観

数式検索システム全体の構成を説明する。数式検索システムは MathML を木に変換する第 1 段階、木を特徴集合に変換する第 2 段階、特徴集合の大きさを削減する第 3 段階の 3 つの部分からなる。第 1 段階では一般的に使用される XML パーサーによって MathML をパースして木に変換する。第 2 段階では、3.2 で説明した 2 種類の変数の性質それぞれに対応する特徴集合の生成を行う 2 種類のアルゴリズムを用いる。1 つめの Subtree Hash は入力の木の部分木をハッシュした整数の集合を生成する。この集合が Subtree Hash で生成された他の集合と多くの共通要素を持てば、それらのもととなった木は多くの共通する部分構造を持つことを意味し、木としての類似尺度となる。2 つめの SIGURE Hash(注3) は Subtree Hash と類似した処理を行うが、取り出された部分木はハッシュされる前に変数名を正規化する。これによって Subtree Hash と同様に木の部分構造を取り出しつつ、変数名の書き換えに対して不変な類似尺度となる。この 2 つのアルゴリズムの出力集合の和集合をとって第 2 段階の出力とする。数式同士の類似度は、入力の数式をこの特徴集合に変換したあとで式 1 によって定まるジャカード係数によって計算する。

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

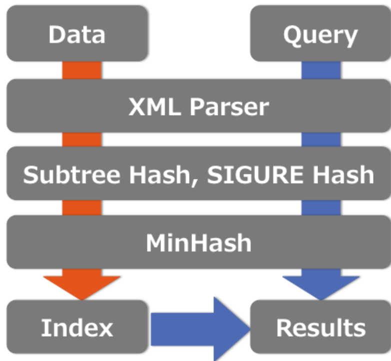
第 3 段階では、ジャカード係数を効率的に計算するための前処理として MinHash [3] を用いる。MinHash は乱択アルゴリズムの一種であり、元の集合の要素に乱択されたハッシュ関数を適用しその最小値を返すものである。この操作により、MinHash が衝突する確率はもとの集合でのジャカード係数と一致するようになる。したがって、十分な数の MinHash 関数を用意することで高い精度でもとのジャカード係数を推定することができる。提案手法では前処理として MinHash 関数を n 個乱択し、それぞれの関数に対してデータセット中の数式のハッシュ値を転置インデックスに保存した。

ユーザーからクエリが与えられた時には、クエリにも 3 段階の処理を適用し、この転置インデックスを参照してデータセッ

(注3)：日本語の時雨と同じように発音する。

ト中の数式とクエリの類似度を推定し、類似度の高いものから順に出力とする。この時の計算時間は転置インデックスで参照した数式の個数に比例するため、検索結果リストにおける類似度の総和に比例する。

図 1　提案手法の全体図



3.4　Subtree Hash

Subtree Hash は入力の木の部分木を表す整数の集合を特徴集合として返す関数である。擬似コード 1 にアルゴリズムの擬似コードを示した。関数の引数 n は特徴集合を求めたい木の根ノードである。この擬似コードでは再帰を実装する都合により、関数は特徴集合だけでなく与えたノードを根とする部分木を表す整数も組み合わせたタプルを返している。従って、引数のノードを根とする部分木に対応する整数をどのように求めるかは返り値の第 1 要素の動きを追うことで理解できる。子ノードがない場合はそのノードのラベルにハッシュ (文字列のハッシュならば何でもよい) を適用した値をその部分木に対応する整数とする (2 行目から 6 行目)。子ノードがある場合は、まずそれらの子ノードに対して再帰的に処理を実行し、それぞれを根とする部分木を表す整数を求める (10 行目)。これを子の順序に従って整数列 X とし、これに対してローリングハッシュ[6] を適用する。ハッシュに用いる定数は現在のノードのラベルから決定する (3 行目の a を用いる)。関数全体として木全体の特徴集合を返すために、再帰的処理の結果もマージしている (擬似コード中の H)。

この関数の時間計算量は、入力の木のノードを n として $O(n)$ である。木の各ノードに対して関数 $SubtreeHash$ はちょうど 1 回呼び出され、また関数 $RollingHash$ において $array$ の要素として各部分木が登場するのもたかだか 1 回であるからである。

3.5　SIGURE Hash

3.5.1　naive SIGURE Hash

Structure Information Greedy Unification REcursive (SIGURE) Hash は Subtree Hash と同様に部分木の集合を取り出すが、取り出したあとで部分木に登場する整数を正規化する点が他と異なる。Subtree Hash は部分木を取り出してその集合の類似度を比較することと同等であるが、変数名が異なるもの

Algorithm 1 SubtreeHash(n)

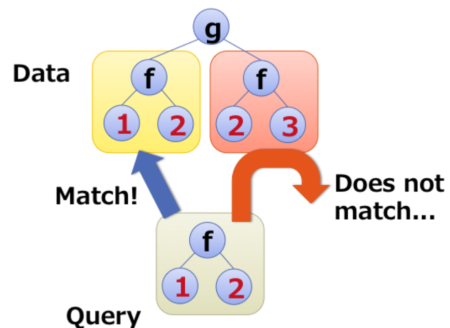
Require: n : root node of the tree

1: **function** SUBTREEHASH(n)
2: 　$C \leftarrow \{c \mid c \text{ is child of } n\}$
3: 　$a \leftarrow hash(label(n))$
4: 　**if** $|C| = 0$ **then**
5: 　　**return** $(a, \{a\})$
6: 　**else**
7: 　　$X \leftarrow newVector()$
8: 　　$H \leftarrow \{\}$
9: 　　**for** c in C **do**
10: 　　　$(x_{sub}, H_{sub}) \leftarrow SubtreeHash(c)$
11: 　　　$X.append(x_{sub})$
12: 　　　$H \leftarrow H \cup H_{sub}$
13: 　　**end for**
14: 　　$x \leftarrow RollingHash(a, X)$
15: 　　**return** $(x, H \cup \{x\})$
16: 　**end if**
17: **end function**

18: **function** ROLLINGHASH(a, $array$)
19: 　$x \leftarrow 0$
20: 　**for** h in $array$ **do**
21: 　　$x \leftarrow x * a + h$
22: 　**end for**
23: 　**return** x
24: **end function**

の数学的に同値な数式に対しては異なる特徴集合を得るため検索結果に現れることはない。このような問題に対し変数名を正規化する手法が考えられるが、1 つの数式に対して正規化を 1 回しか行わないと図 3.5.1 のように部分木だけで見ると同値であるが全体で見た時とは変数の出現順が異なるために書き換え結果がクエリと異なり検索できないことがある。そこで、各部分木をあらわす整数を求めるときに、その部分木のスコープを考慮して変数名を書き換えるようにすることでこの問題を回避する事を考える。naive SIGURE Hash はこれを実現するハッシュ関数である。擬似コード 2 にアルゴリズムを示す。

図 2　単純な変数書き換えで生じうる問題の例



基本的な処理の流れは Subtree Hash に類似しているが、引数に変数名書換の結果を表す連想配列 V が追加されている。これが空の状態で各ノードを根としたときの部分木を表す整数を求めるのが $NSHrec$ 関数である (4 行目から 7 行目)。$NSHrec$ を再帰的に呼び出すことで木を pre-order で走査し (23 行目から 28 行目)、V に現れない変数名が登場するたびに新しい変数名をそれに追加して正規化の結果を V に保存する (12 行目から 14 行目)。そのノードが変数ノードであるかどうかは $isVariableNode$ 関数で検査している (11 行目) 。

関数 $NSHrec$ の時間計算量は $O(n)$ である。再帰によって各ノードを巡回し、各ノードでならし $O(1)$ 時間 ($RolingHash$ 関数の処理に最悪 $O(n)$ 時間かかるが、引数 X の長さの総和はノード数に一致するのでならし計算量では定数時間となる) でハッシュ値を計算しているからである。従って、関数 $NaiveSIGUREHash$ 関数の時間計算量は $O(n^2)$ である。

Algorithm 2 NaiveSIGUREHash(n)

Require: n : root node of the tree
1: **function** NaiveSIGUREHash(n)
2: $H \leftarrow \{\}$
3: $S \leftarrow \{s \mid s$ is a descendant of $n\}$
4: **for** s in S **do**
5: $(x, V) \leftarrow NSHrec(n, newHashMap())$
6: $H \leftarrow H \cup \{x\}$
7: **end for**
8: **return** H
9: **end function**
10: **function** NSHrec(n, V)
11: **if** $isVariableNode(n)$ **then**
12: **if** $label(n)$ not in $key(V)$ **then**
13: $V[label(n)] \leftarrow hash(length(V))$
14: **end if**
15: **return** $(V[label(n)], V)$
16: **end if**
17: $C \leftarrow \{c \mid c$ is child of $n\}$
18: $a \leftarrow hash(label(n))$
19: **if** $|C| = 0$ **then**
20: **return** (a, V)
21: **else**
22: $X \leftarrow newVector()$
23: **for** c in C **do** ▷ Traverse from left child.
24: $(x_{sub}, V) \leftarrow NSHrec(c, V)$
25: $X.append(x_{sub})$
26: **end for**
27: $x \leftarrow RollingHash(a, X)$
28: **return** (x, V)
29: **end if**
30: **end function**

3.5.2 SIGURE Hash

Naive SIGURE Hash によって、適切な正規化を考慮した特徴集合を生成することが可能となったが、Subtree Hash よりも計算量が悪化しており、大規模なデータに対して適用するには不適である。ところで、Naive SIGURE Hash が子ノードでの計算結果を再利用できなかったのは子ノードでの変数出現順がそのままには利用できないからである。例えば、図 3.5.1 での右側の f のノードが根ノードならばそれらの子ノードの変数は図に示された 2, 3 ではなく 1, 2 と書き換えられ、これに基づいて計算されたハッシュ値は再利用できない。そこで、子ノードでの結果を再利用するために根ノードによって変化する各変数に割り当てられるハッシュ値を変数のまま残し、各ノードでの計算結果のハッシュ値は多項式として保持するようにしたのが以下に説明する SIGURE Hash である。

擬似コードをアルゴリズム 3,4 に、木が与えられたときの計算例を図 3.5.2 に示した。

Algorithm 3 SIGUREHash(n)

Require: n : root node of the tree
1: **function** SIGUREHash(n)
2: $(X, V, H) \leftarrow SHrec(n)$
3: **return** H
4: **end function**
5: **function** SHrec(n)
6: $V \leftarrow newHashMap()$
7: $X_v \leftarrow newHashMap()$
8: **if** $isVariableNode(n)$ **then**
9: $V[label(n)] \leftarrow length(V)$
10: $X_v[label(n)] \leftarrow 1$
11: $X \leftarrow (X_v, 0)$
12: **return** $(X, V, \{Eval(X, V)\})$
13: **end if**
14: $C \leftarrow \{c \mid c$ is child of $n\}$
15: $a \leftarrow hash(label(n))$
16: **if** $|C| = 0$ **then**
17: $X \leftarrow (X_v, a)$
18: **return** $(X, V, \{Eval(X, V)\})$
19: **else**
20: $X \leftarrow (X_v, 0)$
21: $H \leftarrow \{\}$
22: **for** c in C **do** ▷ Traverse from left child.
23: $(X_{sub}, V, H_{sub}) \leftarrow SHrec(c)$
24: $X \leftarrow Merge(X, V, X_{sub}, V_{sub}, a)$
25: $H \leftarrow H \cup H_{sub}$
26: **end for**
27: **return** $(X, V, H \cup \{Eval(X, V)\})$
28: **end if**
29: **end function**

このアルゴリズムでは、各部分木の計算結果を整数ではなく登場する変数とその順序 V とその変数からなる多項式の係数 X に分離することで結果を再利用できるようにしている。X はさらに係数部分の X_v と定数項 c に分かれる。 X と V の組が決まるとその部分木に対応する値が決まるようになっており、これは $Eval$ 関数で計算する。木を巡回した時に現在見ているノードと子ノードの X, V を結合する処理は $Merge$ 関数で行われている。$Merge$ 関数はローリングハッシュの 1 ステップを X, V の組に対して行う関数である。X は多項式の係数として計算すればよく、V は V_{sub} にのみ登場する変数を

Algorithm 4 Auxiliary function used in SIGUREHash

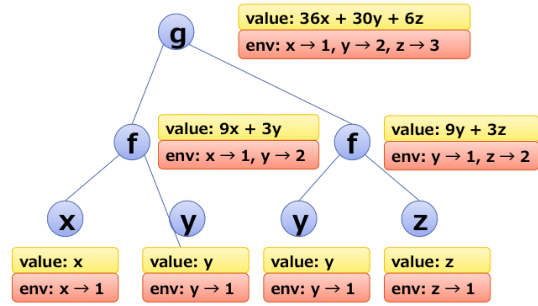
```
1:  function EVAL(X, V)
2:      (X_v, c) ← X
3:      h ← c
4:      for v in key(V) do
5:          h ← h + X_v[v] * hash(V[v])
6:      end for
7:      return h
8:  end function
9:  function MERGE(X, V, X_sub, V_sub, a)
10:     X_v,ret ← newHashMap()
11:     V_ret ← newHashMap()
12:     X_v, c ← X
13:     X_v,sub, c_sub ← X_sub
14:     for v in key(V) do ▷ Traverse from the key which has the
        smallest value.
15:         V_ret[v] ← length(V_ret)
16:         X_v,ret[v] ← X_v[v] * a
17:     end for
18:     for v in key(V_sub) do ▷ Traverse from the key which has
        the smallest value.
19:         if v not in key(V) then
20:             V_ret[v] ← length(V_ret)
21:             X_ret[v] ← 0
22:         end if
23:         X_ret[v] ← X_ret[v] + X_sub[v]
24:     end for
25:     c_ret ← c * a + c_sub
26:     return ((X_v,ret, c_ret), V_ret)
27: end function
```

V の後ろに追加すれば pre-order での順序が保たれる。この時、V の計算においてそれぞれの係数に掛かる数字は定数 a しかないので V に 2 次以上の項が含まれることはない。ゆえに $|V| = |X|$ が常に成り立つ。呼び出し元にはこれらの再利用に必要な X, V を返しつつ、 $Eval(X, V)$ の結果を H に加えて特徴集合を作っていく (27 行目)。

具体的な入力例を用いて動作を説明する。例えば、入力の式が $g(f(x, y), f(y, z))$ であったとしよう。この式は図 3.5.2 に示されるような木に変換されたとする。それぞれの葉ノードはアルゴリズム 3 の 8 行目から 12 行目までの処理によりその変数のみが 1 番目に登場したいう情報 (env) と、部分木を表す整数はその変数のハッシュ値そのものになるという情報 (value) が求められる。左側の f のノードは子ノードの結果と $Merge$ 関数を用いることで自身の結果を求める。図の例では $hash(f) = 3$ であったとしており、これによって、value には $3^2(x) + 3^1(y)$ となっている。右側の f のノードでも同様にして value は $3^2(y) + 3^1(1)$ となる。そして根の g のノードでは、$hash(g) = 4$ として value は $4^2(9x + 3y) + 4^1(9y + 3z)$ となる。

このアルゴリズムの時間計算量は $O(nk)$ である。$Eval$, $Merge$ 関数はともに各ノードにつき 1 回呼び出され、これらの計算量はともにループに着目して最悪 $O(k)$ であるから、処理全体として $O(nk)$ となる。実用上 k は n に対して十分小さいので、このアルゴリズムは実データに対してほぼ線形時間で動作する。

図 3　SIGURE Hash の計算例



4. 実　　験

4.1 実験内容

提案手法の有効性を確かめるために、NTCIR11-Math2 [1] で用いられた数式検索用のデータセットを用いて実際に提案手法を実装し、索引付け時間と検索精度の観点で既存の木構造類似検索と比較した。検索精度の基準としては、出力リストの上位 10 個以内に関連する文書が含まれる割合である P@10 とそれぞれの関連文書がリストに登場した位置での適合率の平均である MAP (Mean Average Precision) を用いた。また、Subtree Hash と SIGURE Hash が実際に異なる種類の類似性を捉えていることを確かめるため、これらを単独で用いた際の検索結果を比較した。

4.2 実験データ・実装

実験においては、NTCIR11-Math2 データセットを用いた。これは数式検索システムの評価を目的としたデータセットであり、約 10 万の論文を約 830 万の検索単位に分割したものである。データサイズは約 180GB であり、約 6000 万個の数式が含まれる。また、このデータセットに対する 50 の検索クエリと一部の検索単位に対するクエリへの関連度も含まれる。

実験では、提案手法である Subtree Hash と SIGURE Hash を統合したものの他に、Subtree Hash 単独, SIGURE Hash 単独をそれぞれ用いるもの、そして既存手法として pq-gram の 4 通りの類似検索手法を実装して比較した。pq-gram のパラメータには、[2] を参考に $p = q = 3$ を用いた。XML パーサ、MinHash の部分については 4 手法で共通のものを用い、MinHash 関数の個数は 30 個とした。また、大きなデータセットに対応するため、これを分割してそれぞれに対して担当のプロセスを割り当てることによる並列化を実装した。

実験環境は Intel Xeon E7-4870 @ 2.40GHz * 4, 1TB RAM, Python 2.7 を使用し、32 プロセスで並列化した。

4.3 処理時間

表 4.3 に処理時間を示した。pq-gram はノード数 n の木に

表 1　クエリごとの比較

クエリ	クエリのキーワード	*Subtree*	*SIGURE*
$f(z) = z^d + c$	Mandelbrot, dynamical plane	0.0745	0.1492
$\|u \cdot v\| \leqq \|\|u\|\|\|\|v\|\|$	Cauchy Schwarz	0.0019	0.0572
$\|\|x+y\|\|_p \leqq \|\|x\|\|_p + \|\|y\|\|_p$	minkowski inequality	0.0215	0.0748
$E(\lambda) = -m^2_{\mathrm{dyn}}(\lambda)$	Electron Energy, Quantum Dynamics, NJL	0.2539	0.0250
$S_{EH} = \frac{1}{G_3} od^3x \sqrt{-g^{(3)}}$	entropy, bound, Bekenstein-Hawking	0.2222	0.0054
$A = USV^T$	singular value decomposition, matrix	0.2167	0.0277

表 2　処理時間の比較

類似尺度	索引作成 [秒]	平均クエリ処理時間 [秒]
Subtree	18865	5.69
SIGURE	20854	10.29
Subtree + SIGURE	21854	8.07
pq-gram	112388	3.68

対して線形時間で動作し $(p+q)n$ 個の要素を持つ特徴集合を生成するのに対して、Subtree, SIGURE Hash はともに n 個の要素を持つ特徴集合を生成する。本実験では $p = q = 3$ に設定しているためちょうどその要素数に 6 倍の差があり、それがそのまま索引作成時間に反映されている。クエリ処理時間については、転置インデックスを採用しているため数式の平均類似度が高ければ処理時間が長くなる傾向がある。たとえば SIGURE Hash では変数 1 個を含む部分木はクエリを含むほとんどの数式に含まれているので、これに対する転置インデックスの参照などに長時間を要した。

4.4 検索精度

表 3　検索精度の比較

類似尺度	*P@10*	*MAP*
Subtree	0.0700	0.0461
SIGURE	0.0700	0.0288
Subtree + SIGURE	0.0820	0.0637
pq-gram	0.0420	0.0446

表 4.4 に検索精度を示した。P@10, MAP いずれの基準においても提案手法が pq-gram を上回っている。Subtree Hash, SIGURE Hash それぞれの精度は MAP で pq-gram 以下か同程度であるが、これらを結合するとより高い精度が達成されており、これらの手法がそれぞれ異なる数式を検索していることを示している。NTCIR11-Math task [1] に参加した他のシステムと比較すると、クエリとして与えられる数式とキーワードを共に活用したものには劣るが、数式だけを用いたものとほぼ同等の精度を達成している。

4.5 クエリ特性

Subtree Hash と SIGURE Hash で特に MAP の差が大きいクエリについて、それぞれが優位なものを 3 つずつ、クエリとその精度を表 4.2 示す。キーワードは数式を表す英語のフレーズである。SIGURE Hash が優位な 3 つは数式中の変数の関係がその数式の意味の中心に据えられているという特徴がある。例えばマンデルブロ集合を記述する漸化式では数列の項の関係が意味の中心であるし、残りの 2 つの不等式も比較される 2 つの変数の関係が意味の中心となっている。一方、Subtree Hash が優位な 3 つはいずれも変数名そのものに意味が含まれている。物理学に由来する 2 つの数式はそれぞれの変数が物理量に対応しているし、特異値分解についても同一の変数が複数回登場することはなく、その関係性というよりは相異なる行列によって分解することに意味があるような式である。以上で確認したように、Subtree Hash と SIGURE Hash はそれぞれ、変数名そのものが重要である数式と変数の関係性が重要である数式をそれぞれ検索することができていた。

5. 結　　論

数式には同義ではあるが表記が異なるものが存在し、これが一般的な木構造類似検索手法を数式検索に適用することを難しくしていた。本研究で新しく開発した SIGURE Hash アルゴリズムはこの問題を解決するとともにほとんど線形の時間計算量も併せ持つことで高速な処理時間を実現することを可能にした。また、このアルゴリズムを並列化した上で実装し、実際に大規模な数式検索データセットを用いて実験することにより、既存の木構造類似検索アルゴリズムよりも高い精度かつ短い前処理時間で検索を行えることを確かめた。

今後の課題としては、提案手法はハッシュ関数であるため衝突の可能性などがあるが、これに関する理論的な評価を与えることが挙げられる。

6. 謝　　辞

本研究の一部は JSPS 科研費 2430062, 25245084 助成を受けた。

文　　献

[1] Akiko Aizawa, Michael Kohlhase, Iadh Ounis, and Moritz Schubotz. Ntcir-11 math-2 task overview. In *Proceedings of NTCIR-11 Math-2 task Workshop Meeting [1]*, 2014.

[2] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. The pq-gram distance between ordered labeled trees. *ACM Transactions on Database Systems (TODS)*, 35(1):4, 2010.

[3] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[4] Erik D Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms (TALG)*, 6(1):2, 2009.

[5] Shahab Kamali and Frank Wm Tompa. Retrieving documents with mathematical content. In *Proceedings of the 36th international ACM SIGIR conference on Research and*

development in information retrieval, pages 353–362. ACM, 2013.

[6] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[7] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979.

[8] Peisen Yuan, Chaofeng Sha, Xiaoling Wang, Bin Yang, Aoying Zhou, and Su Yang. Xml structural similarity search using mapreduce. In *Web-Age Information Management*, pages 169–181. Springer, 2010.

[9] 大橋駿介, 高須淳宏, and 相澤彰子. 表記が異なる同義の数式の高速な検索法. **第 *6* 回データ工学と情報マネジメントに関するフォーラム** *(DEIM2014)*, 2014.