社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－DBS－136（6）
2005－FI－79（6）
2005／5／19

テキスト構文構造類似度を用いた類似文検索手法

市川 宙† 橋本 泰一† 徳永 健伸† 田中 穂積‡
† 東京工業大学 大学院情報理工学研究科 計算工学専攻
‡ 中京大学 情報科学部 認知科学科
{ichikawa,taiichi,take,tanaka}@cl.cs.titech.ac.jp

本論文では，構文木付きコーパスから，構文的に類似した文を検索する手法を提案した．構文的類似度の計算手法としては Tree Kernel （Collins）が提案されている．しかし，Tree Kernel の類似度計算は時間を要するため，これを類似文検索に応用すると，検索速度が問題になる．検索時間短縮のためには，予め検索対象のインデックスを作成しておくのが一般的だが，Tree Kernel ではその性質上，検索対象のインデックス化が困難である．そこで，Tree Kernel を近似する高速な新しいアルゴリズムとして Tree Overlapping と Subpath Set を提案した．これらのアルゴリズムは，Tree Kernel とは異なり，検索対象のインデックス化が可能なため，高速な検索が可能である．本論文では Tree Kernel, Tree Overlapping, Subpath Set の 3 種類のアルゴリズムについて述べ，実験結果を示し，比較した．

New methods to retrieve sentences based on syntactic similarity

Hiroshi Ichikawa†, Taiichi Hashimoto†, Takenobu Tokunaga†, Hozumi Tanaka‡
† Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
‡ Cognitive Science Major, Graduate School of Computer and Cognitive Sciences, Chukyo University
{ichikawa,taiichi,take,tanaka}@cl.cs.titech.ac.jp

This paper proposes a method to retrieve sentences which have a similar syntactic structure to the syntax tree of the query sentence. Tree Kernel has been proposed by Collins as a method to calculate structural similarity. However, the similarity retrieval by Tree Kernel is not practicable because Tree Kernel computation requires significant resources. A general method to shorten the retrieving time and to reduce required computation is indexing the corpora beforehand. However, in case of Tree Kernel, it is too hard to index the corpora. Therefore, we propose faster approximation algorithms: Tree Overlapping and Subpath Set. These algorithms are faster than Tree Kernel because indexing is possible. This paper describes three algorithms: Tree Kernel, Tree Overlapping and Subpath Set, and shows the result of evaluations and algorithm comparison.

1 はじめに

近年，情報検索や機械翻訳のために，様々な類似文検索の技術が考案されている．1 つは，文中の語（形態素など）の出現頻度に基づくものである．類似度の基準は様々だが，多くの語を共通に含む文を類似文とする手法である．これは，話題や内容が類似する文を検索するのに適した方法であり，情報検索などで用いられている．もう 1 つは，文中の品詞や助詞，助動詞の並びに注目したものであり，用例ベース機械翻訳などに使われている．[1, 2]

しかし，品詞や助詞，助動詞の並びが類似していても，構文構造や係り受け関係が全く異なる場合もある（表 1）．このような文を類似文とみなして翻訳をすると，誤訳になってしまう．これは，従来の類似文検索の手法が，語や品詞などの表層の情報のみを扱い，構文構造などの抽象度の高い情報を扱っていないのが原因である．

構文木の類似度の基準として，Collins[3, 4] の提案する Tree Kernel がある．しかし，数万もの文を持つコーパスから類似文を検索するのに，個々の構文木との Tree Kernel による類似度を計算するのでは，時間がかかりすぎる．そのため，検索時間短縮のために，予め検索対象のインデックスを作ることが考えられる．しかし Tree Kernel を用いた類似文検索では，アルゴリズムの性質上，インデックス化は困難である．

そこで，検索に適した高速な類似度算出アルゴリズム，Tree Overlapping と Subpath Set を提案する．

これらのアルゴリズムが使用する類似度は，それぞれ Tree Kernel とは異なるが，類似の傾向を示すため，これらは Tree Kernel の近似アルゴリズムと見なせる．また，Tree Kernel とは異なり，両者ともインデックス化を用いた検索の高速化が可能である．

表 1: 表層が似ているが構文構造が異なる例

クエリ文	高層ビルで火災発生と新聞が伝えた．
類似文	道路で自動車と自転車が衝突した．

2 Tree Kernel

2.1 類似度の定義

Collins ら [3, 4] は木構造間の類似度を与える Tree Kernel という手法を提案した．Tree Kernel では，2 つの木構造の類似度を，それらの木構造が共通に含む部分木の数と定義している．ただし，ここで言う「部分木」には，

- 2 個以上のノードを持つ．
- 個々の導出規則の一部だけを含んではいけない．

という制約がある．

類似度に求められる性質は応用によって異なるため，Collins の Tree Kernel がどの応用にも適しているとは限らない．そこで高橋ら [5] は，様々な応用に適用できるように，Tree Kernel をベースとした 3 種類の類似度を提案した．本研究では，高橋らが提案した類似度 C（K_C）を用いる．

$$K_C(T_1, T_2) = \max_{n_1 \in N_1} \max_{n_2 \in N_2} C(n_1, n_2) \qquad (1)$$

ここで $C(n_1, n_2)$ は，n_1 を根とする部分木としても，n_2 を根とする部分木としても出現する部分木の数である．

2.2 アルゴリズム

Collins[3, 4] は，以下の規則で $C(n_1, n_2)$ を求めることで，効率的に Tree Kernel を計算する手法を提案した．

- n_1 と n_2 の子ノードを導出する規則が異なる場合，$C(n_1, n_2) = 0$
- n_1 と n_2 の導出規則が等しく，n_1 と n_2 がともに前終端記号の場合，$C(n_1, n_2) = 1$
- n_1 と n_2 の導出規則が等しく，n_1 と n_2 がともに前終端記号でない場合，

$$C(n_1, n_2) = \prod_{i=1}^{nc(n_1)} (1 + C(ch(n_1, i), ch(n_2, i))) \qquad (2)$$

ここで $nc(n)$ は n の子ノードの数を示し，$ch(n, i)$ はノード n の i 番目の子ノードを示す．式 2 では子ノードの C を用いるが，各ノード間の C を後順序で求めれば，再計算は不要である．その結果，T_1, T_2 のノード数をそれぞれ m, n とおくと，$K_C(T_1, T_2)$ の時間計算量は $O(mn)$ となる．

Collins や高橋は，Tree Kernel の類似度を用いた検索手法については述べていない．ここでは，クエリとなる木構造と，検索対象の全ての木構造との間で個々に類似度 $K_C(T_1, T_2)$ を計算し，それをソートするという素朴なアルゴリズムを採用した．

2.3 特徴

共通する部分木全てを考慮に入れるため，使用する情報量が多いという利点がある．しかし毎回，検索対象の全ての木構造との間で個々に類似度を計算するため，時間を要するという欠点がある．

多くの検索手法では，検索時間を短縮するために，予め検索対象のインデックスを作成しておく．Tree Kernel の場合，あらゆる部分木をキーとして，それを含む木をインデックス化できれば，高速な検索が期待できる．しかし部分木は 1 つの構文木の中に数百万も存在するため，実際には，部分木をキーとするインデックス化は困難である．

3 Tree Overlapping

3.1 類似度の定義

Tree Overlapping が用いる類似度 $S_{TO}(T_1, T_2)$ を以下のように定義する．

木 T_1 のノード n_1 と木 T_2 のノード n_2 が同じ位置に重なるように，T_1 と T_2 を重ね合わせることを考える．この時,「同じ場所に同じ導出規則[1]が重なる」ことがある．そのような「完全に重なった導出規則」の数を $C_{TO}(n_1, n_2)$ とする．$S_{TO}(T_1, T_2)$ はこの $C_{TO}(n_1, n_2)$ を使って，以下のように定義される．これは Tree Kernel の類似度 K_C の式に対応する．

$$S_{TO}(T_1, T_2) = \max_{n_1 \in N_1} \max_{n_2 \in N_2} C_{TO}(n_1, n_2) \qquad (3)$$

ここまで,「T_1 と T_2 を重ね合わせる」といった表現をしてきたが，ここで厳密な定義を述べる．変数，関数を以下のように定義する．

m_1, m_2	T_1, T_2 の任意のノード.
i	任意の自然数.
$nonterm(T)$	木 T が持つ非終端ノードの集合.
$ch(n, i)$	ノード n の i 番目の子ノード.
$prod(n)$	ノード n の子ノードを導出する規則. 図 1 の例では $prod(b) = (b \rightarrow de)$
$L(n_1, n_2)$	「n_1 と n_2 が同じ位置に重なるように T_1 と T_2 を重ね合わせた時に，重なるノードの組」の集合. 図 1(3) の例では $L(g, g') = \{(g, g'), (i, i')\}$

$L(n_1, n_2)$ を以下のように定義する．

1. $(n_1, n_2) \in L(n_1, n_2)$
2. $(m_1, m_2) \in L(n_1, n_2)$ ならば，
   $(ch(m_1, i), ch(m_2, i)) \in L(n_1, n_2)$
3. $(ch(m_1, i), ch(m_2, i)) \in L(n_1, n_2)$ ならば，
   $(m_1, m_2) \in L(n_1, n_2)$
4. 2〜3 を再帰的に適用してもたどり着けないものは $L(n_1, n_2)$ に含まれない．

$C_{TO}(n_1, n_2)$ は以下のように定義できる．

$$C_{TO}(n_1, n_2) = \left| \left\{ (m_1, m_2) \;\middle|\; \begin{array}{ll} & m_1 \in nonterm(T_1) \\ \wedge & m_2 \in nonterm(T_2) \\ \wedge & (m_1, m_2) \in L(n_1, n_2) \\ \wedge & prod(m_1) = prod(m_2) \end{array} \right\} \right| \qquad (4)$$

3.2 類似度の例

例として，図 1(1) の T_1 と T_2 の類似度 $S_{TO}(T_1, T_2)$ を計算する．

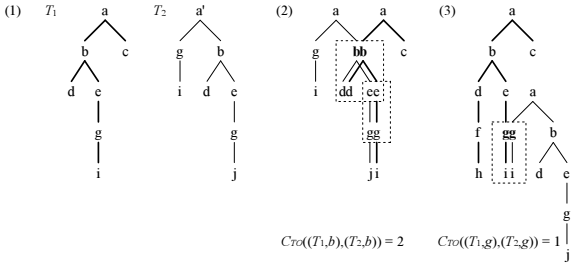

図 1: 類似度の計算例

まず，$C_{TO}((T_1, b), (T_2, b))$ を求めてみる．T_1 と T_2 のノード b が重なるように T_1 と T_2 を重ね合わせたのが，図 1(2) である．こうすることで，$b \rightarrow de, e \rightarrow g$ の 2 つの導出規則が同じ位置に重なった．よって，$C_{TO}((T_1, b), (T_2, b)) = 2$ となる．

同様に，$C_{TO}((T_1, g), (T_2, g))$ を求めてみる．T_1 と T_2 のノード g が重なるように T_1 と T_2 を重ね合わせたのが，図 1(3) である．図から，完全に重なっている導出規則は $g \rightarrow i$ だけだと分かる．よって，$C_{TO}((T_1, g), (T_2, g)) = 1$ となる．

同様に他のノードについても C_{TO} を計算すると，3 以上のものは無いことが分かる．よって $S_{TO}(T_1, T_2) = 2$ と求まる．

[1]木構造の中では,「親ノード（左辺）と子ノード（右辺）」という形で現れる．

3.3 アルゴリズム

3.3.1 基本的なアイディア

基本的には Tree Kernel の検索（2.2 節）と同じく，以下の手法で類似度の高い木を見つけ出す．

- クエリ T_0 と，検索対象中の個々の木 T との間の類似度 $S_{TO}(T_0, T)$ を全て求める．
- それをソートする．

ただし，Tree Overlapping では，Tree Kernel と異なり，予めインデックスを作っておくことで，類似度計算を高速化できる．例を使って説明する．

例として，クエリを図 2 の T_0 とし，検索対象を図 2 の T_1, T_2 の 2 つだけを含むコーパスとする．

予め，あらゆる導出規則 p に対して「その規則がコーパス中のどの木のどこに出現するか」というインデックス $I[p]$ を作成しておく．今回の例では，表 2 のようになる．

表 2: インデックスの例

p	$I[p]$
$a \to bc$	$\{(T_1, a)\}$
$b \to de$	$\{(T_1, b), (T_2, b)\}$
$e \to g$	$\{(T_1, e), (T_2, e)\}$
$g \to i$	$\{(T_1, g), (T_2, g)\}$
$a \to gb$	$\{(T_2, a)\}$
$g \to j$	$\{(T_2, g)\}$

次に，クエリに含まれる各導出規則（ここでは $a \to bc$ と $b \to de$）について，インデックスを使って「同じ規則がコーパス中のどこに出現するか」を探し出す．まず $a \to bc$ を探すと，T_1 の a が見つかる．そこで,「a が同じ位置に来るような T_1 と T_0 の重ね方」（図 2(2)）に対して，1 ポイントを与える．(この「重ね方」によって重なる導出規則が 1 つ見つかった，という意味である)

同様に，$b \to de$ をコーパス中から探すと，T_1 の b と T_2 の b が見つかる．そこで,「b が同じ位置に来るような T_1 と T_0 の重ね方」（図 2(2)）と,「b が同じ位置に来るような T_2 と T_0 の重ね方」（図 2(3)）に，それぞれ 1 ポイントを与える．

結局，図 2 の (2) の「重ね方」に 2 ポイント，(3) の「重ね方」に 1 ポイントが入る．このポイントが C_{TO} の値に等しくなるため，ここから $S_{TO}(T_0, T_1) = 2, S_{TO}(T_0, T_2) = 1$ が求まる．

ここで，ポイントを与える対象である「重ね方」をどう表現するかが問題となる．つまり,「T_1 と T_0 の a が同じ位置に来るような重ね方」と「T_1 と T_0 の b が同じ位置に来るような重ね方」を同一視できなければ，アルゴリズムは正しく働かない．

そこで,「n と m が同じ位置に重なるように，2 つの木を重ねた時に，重なったノードの組（$L(n, m)$）の中で最も根に近い物」を与える関数 top を導入する．図 2 T_0, T_1, T_2 における $top(n, m)$ の例を表 3 に示す．この $top(n, m)$ は「重ね方」と 1 対 1 対応することが分かる．よって,「n と m が同じ位置に来るような重ね方」を $top(n, m)$ で表現する．

表 3: $top(n, m)$ の例

(n, m)	$top(n, m)$
$((T_0, a), (T_1, a))$	$((T_0, a), (T_1, a))$
$((T_0, b), (T_1, b))$	$((T_0, a), (T_1, a))$
$((T_0, d), (T_1, d))$	$((T_0, a), (T_1, a))$
$((T_0, b), (T_2, b))$	$((T_0, b), (T_2, b))$
$((T_0, d), (T_2, d))$	$((T_0, b), (T_2, b))$

3.3.2 検索アルゴリズム

以上のアイディアをまとめたのが，以下のアルゴリズムである．この節では，以下の記号を用いる．

T_0	クエリとなる木.
F	検索対象となる木の集合.
$nonterm(T)$	木 T が持つ非終端ノードの集合.
$tree(n)$	ノード n が所属する木.
$prod(n)$	ノード n の子ノードを導出する規則.
$parent(n)$	ノード n の親ノード.
$order(n)$	ノード n が $parent(n)$ の k 番目の子なら，$order(n) = k$

予め，あらゆる導出規則 p に対して，以下のようなインデックス $I[p]$ を作成する．

$$I[p] = \{m | T \in F \wedge m \in nonterm(T) \wedge p = prod(m)\} \tag{5}$$

以下のアルゴリズムで $C[n, m]$ の値（スコア）を求める．

$C[n, m] := 0$ **for all** (n, m)
foreach n **in** $nonterm(T_0)$ **do**
　　foreach m **in** $I[prod(n)]$ **do**
　　$(n', m') := top(n, m)$

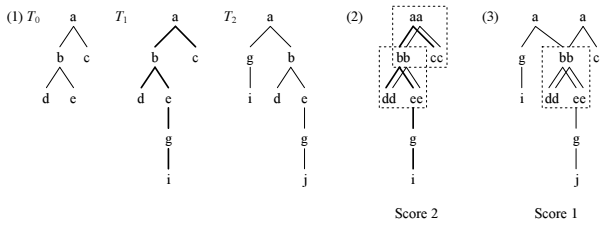

図 2: Tree Overlapping の検索の例

$$C[n', m'] := C[n', m'] + 1$$

$top(n, m)$ を求めるアルゴリズムは以下のとおり.

function $top(n, m)$;
begin
  $(n', m') := (n, m)$
  while $order(n') = order(m')$ **do**
    $n' := parent(n')$
    $m' := parent(m')$
  return (n', m')
end

ここまでで $C[top(n, m)] = C_{TO}(n, m)$ となるので,

$$S_{TO}(T_0, T) = \max_{n \in nonterm(T_0)} \max_{m \in nonterm(T)} C[n, m] \quad (6)$$

によって，各木の類似度 $S_{TO}(T_0, T)$ が求まる.

3.4 特徴

$S_{TO}(T_1, T_2)$ の値は,「T_1, T_2 に共通する最大の部分木に含まれる導出規則の数」とほぼ一致する．つまり，Tree Kernel の K_C と同様,「両方の木に共通する部分木の大きさ」を表す値と言える．ただし，部分木の「大きさ」の定義は多少異なる.

ただし，図 3 のように，途中にラベルの異なるノードが挟まっていても，まとめてカウントしてしまう.

また，Tree Kernel と異なり，検索対象のインデックス化が可能なため，Tree Kernel より高速な検索が可能である.

4 Subpath Set

4.1 類似度の定義

テキスト間の類似度としては，テキストの単語頻度ベクトルに基づいた類似度がよく用いられる．これを木構造に応用したのが Subpath Set の類似度である.

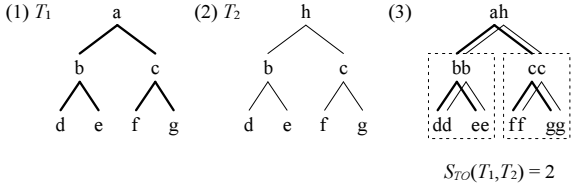

図 3: 別の共通部分木をまとめてカウントしてしまう例

Subpath Set では，テキストを構成する「語」に相当するものを，木構造の「根から葉までの経路とその一部」とした．これらを木 T の部分経路と呼ぶ.

そして，部分経路の出現頻度ベクトルに基づいて，類似度を算出する．頻度ベクトルに基づく類似度には様々なものがあるが，ここでは「両方の木に共通する部分経路の数（重複はカウントしない）」という単純なものを用いる.

4.2 類似度の例

例えば，図 4 の T_1 と T_2 の部分経路を列挙すると図 4(2) のようになるため，SS による T_1 と T_2 の類似度（共通する部分経路の数）は 15 となる.

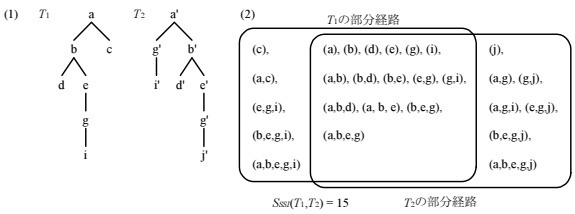

図 4: 部分経路の例

4.3 アルゴリズム

クエリの木を T_0，検索対象の木の集合を F とする．また，木 T の部分経路の集合を $P(T)$ と表す.

予め，あらゆる部分経路 p について，以下のようなインデックス $I[p]$ を作成しておく．

$$I[p] = \{T | T \in F \wedge p \in P(T)\} \qquad (7)$$

以下のアルゴリズムで $S[T]$ を計算する．

$S[T] := 0$ **for all** T
foreach p **in** $P(T_0)$ **do**
　foreach T **in** $I[p]$ **do**
　　$S[T] := S[T] + 1$

このようにして計算した $S[T]$ が「T_0 と T に共通する部分経路の数」になるので，これをソートして比較すればよい．

4.4 特徴

Tree Overlapping と同様，検索対象のインデックス化による検索の高速化が可能である．更に，一般的な単語ベースの類似文検索手法を用いるため，既存のツールの利用も可能である．

ただし，Tree Kernel や Tree Overlapping に比べると，何番目の子ノードかを区別しないなど，粗い情報に基づく検索である．そのため，検索結果にノイズが入りやすい．

しかしその分アルゴリズムが簡単なので，Tree Overlapping より更に高速である．

5 実装と実験

5.1 実装

以下の各アルゴリズムについて,「木構造をクエリとする検索」と「PSF をクエリとする検索」を Ruby 1.8.2 で実装した．

- Tree Kernel (TK)
- Tree Overlapping (TO)
- Subpath Set (SS)

5.2 実験方法

新しく提案したアルゴリズムの有効性を評価するため，以下の条件で実験を行った．

- 実行するのは，5.1 節で実装した 3 種類のアルゴリズム．
- 検索対象は東工大コーパス [6] から抜き出した 2483 文．全て，人手による木構造が付いている．
- その中からランダムに選んだ 100 文の木構造をクエリとして，検索を行う．
- 個々の検索の所要 CPU 時間と，検索結果（1 位から最下位まで）を記録する．
- 実験環境のマシンは CPU Xeon 2.4GHz，メモリ 2GB．

5.3 実験結果

今回の実験では，検索対象にクエリ自身が含まれる．実験の結果，どのアルゴリズムでも，クエリ自身は 1 位でヒットすることが分かった．よって，以下に示す実験結果では，クエリ自身を検索結果から除外した．

この実験には絶対的な正解データが存在しないため，各アルゴリズムを相互に比較することで評価した．

ここで示すのは以下の値である．

- 各アルゴリズムでの 1 クエリ当たりの平均検索時間（CPU 時間）．
- 各アルゴリズムで 1 位でヒットした文が，他のアルゴリズムで何位になるか（相加平均，相乗平均）．
- 各アルゴリズムの n 位（$n = 1, 5, 10, 50$）以上同士に，共通する文が何文あるか．

表 4: 1 クエリ当たりの平均検索時間

アルゴリズム	平均検索時間 [s]
TK	529.42
TO	6.29
SS	0.47

表 5: A で 1 位だった文の B での順位の相加平均

$B \diagdown A$	TK	TO	SS
TK		67.30	75.97
TO	**8.79**		34.90
SS	**81.27**	37.27	

表 6: A で 1 位だった文の B での順位の相乗平均

$B \diagdown A$	TK	TO	SS
TK		11.64	17.86
TO	**3.65**		6.57
SS	**15.95**	8.31	

表 7: TK の n 位までと各アルゴリズムの n 位までに共通する文の数と割合

アルゴリズム	1 位まで	5 位まで	10 位まで	50 位まで
TK	1.00 (100.0%)	5.00 (100.0%)	10.00 (100.0%)	50.00 (100.0%)
TO	0.25 (25.0%)	1.68 (33.6%)	3.63 (36.3%)	19.45 (38.9%)
SS	0.15 (15.0%)	0.98 (19.6%)	2.18 (21.8%)	13.07 (26.1%)

表 8: TO の n 位までと各アルゴリズムの n 位までに共通する文の数と割合

アルゴリズム	1 位まで	5 位まで	10 位まで	50 位まで
TK	0.25 (25.0%)	1.68 (33.6%)	3.63 (36.3%)	19.45 (38.9%)
TO	1.00 (100.0%)	5.00 (100.0%)	10.00 (100.0%)	50.00 (100.0%)
SS	0.25 (25.0%)	1.59 (31.8%)	3.38 (33.8%)	17.96 (35.9%)

表 9: SS の n 位までと各アルゴリズムの n 位までに共通する文の数と割合

アルゴリズム	1 位まで	5 位まで	10 位まで	50 位まで
TK	0.15 (15.0%)	0.98 (19.6%)	2.18 (21.8%)	13.07 (26.1%)
TO	0.25 (25.0%)	1.59 (31.8%)	3.38 (33.8%)	17.96 (35.9%)
SS	1.00 (100.0%)	5.00 (100.0%)	10.00 (100.0%)	50.00 (100.0%)

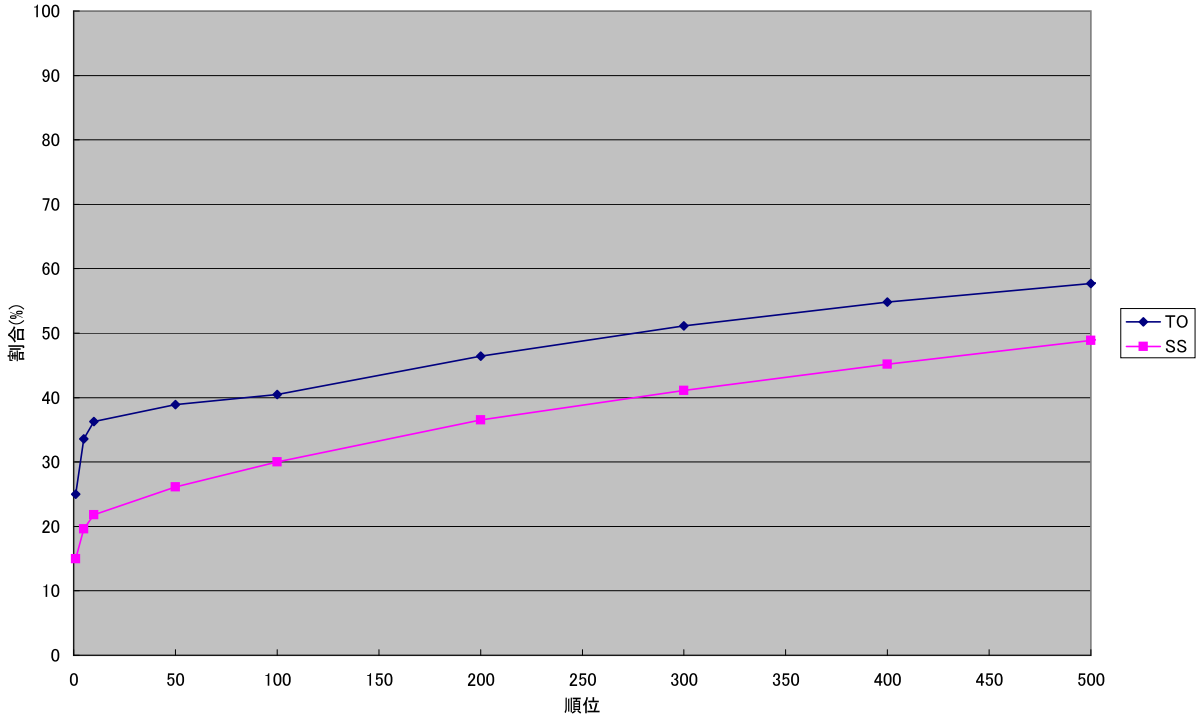

図 5.1 TK と各アルゴリズムの上位に共通する文の割合

5.4 実験結果の評価

ここでは，TK（Tree Kernel）の検索結果を基準とする．Tree Kernel は既存のよく知られた類似度の尺度であり，使用される情報も多いと考えられるためである．

表 4 から，TO（Tree Overlapping）の検索時間は TK の約 1/84 である．SS-T と SS-S（Subpath Set）に至っては，TK の約 1/4800 の検索時間を実現している．このことから，検索の高速化には成功したと言える．

TO の検索時間（1 クエリ当たり 6.29 秒）は，用途によってはやや大きいと感じるかもしれない．しかし現在の TO は完全にインタプリタ方式の言語（Ruby）で実装したため，これを効率に留意してコンパイラ型の言語で再実装すれば，更なる高速化が可能である．

TK で 1 位だった文の順位を比較すると（表 5，表 6 の太字部分），TO の検索結果の方が SS より TK に近いということが分かる．図 5.1 からも同様のことが分かる．

以上から，テキストの構文的類似度を用いた検索には，

- 検索結果の「質」を求めるなら，TO
- 検索の「速度」を求めるなら，SS

が適していることが分かる．

6 本研究のまとめ

既存の木構造類似度の尺度である Tree Kernel を用いた検索をベースラインとし，木構造から類似の木構造を検索する，高速なアルゴリズムを 2 種類提案した．

- Tree Overlapping．Tree Kernel に近い結果を出力し，Tree Kernel よりはるかに高速である．
- Subpath Set．精度は落ちるが，Tree Overlapping より更に高速である．

本論文で提案したアルゴリズムで出力された類似文を人手でチェックしたところ，人間が見ると類似文とは思えないものが多数含まれていた．

これは，人間にとっての類似文には，構造だけではなく，意味や表層の語の影響が大きいためである．

よって，人間から見ても似ている文を検索するためには，

- 表層の語の一致も考慮する．
- 構文構造以外の構造（依存構造など）を用いる．

などの工夫が必要である．

また，望ましい類似文の性質は応用によって異なるため，実際の応用に適用して評価することが必要である．1 節で例として述べた機械翻訳や，構文木付きコーパスの作成作業支援などへの応用を模索していきたい．

参考文献

[1] Somers H, I McLean, D Jones. Experiments in multilingual example-based generation. CSNLP 1994: 3rd conference on the Cognitive Science of Natural Language Processing, Dublin, 1994.

[2] Nagao, Makoto. A framework of a mechanical translation between Japanese and English by analogy principle. In Alick Elithorn and Ranan Banerji, editors, Artifical and Human Intelligence, pages 173-180. Amsterdam, 1984.

[3] Collins, M. and Duffy, N. Parsing with a Single Neuron: Convolution Kernels for Natural Language Problems. Technical report UCSC-CRL-01-01, University of California at Santa Cruz, 2001.

[4] Collins, M. and Duffy, N. Convolution Kernels for Natural Language. In Proceedings of NIPS 2001, 2001.

[5] 高橋哲朗, 乾健太郎, 松本裕治. テキストの構文的類似度の評価方法について. 情報処理学会自然言語処理研究会, NL-150-7, 2002.

[6] 野呂智哉, 橋本泰一, 徳永健伸, 田中穂積. 大規模日本語文法の開発. 自然言語処理, Vol.12, No.1, pp.3-32, 2005.