

文字列検索結果に対するコンパクトな文脈集合の高速抽出

海野 裕也 坪井 祐太

日本アイ・ビー・エム株式会社 東京基礎研究所

{yunno, yutat}@jp.ibm.com

概要

大量の文書データに対して文字列検索した際、検索語の周辺文脈は有用な情報となる。KWIC (KeyWord In Context) をはじめとする従来の手法では、複数の文脈情報を纏め上げて概観することができなかった。本稿では指定された表示領域に対して最適な文脈情報をインタラクティブに表示する手法を提案する。特に面積最大化原理に基づく最適な文脈表示の定式化と、動的計画法による効率的なアルゴリズムを示す。また、探索において上限値を見積もることによる枝刈り手法と、そのための頻度順接尾辞木という新しいデータ構造に関して解説する。提案手法は主観評価で KWIC と比べて有用な情報を提供できた。また単純な動的計画法がデータ数に比例した時間がかかるのに対して、枝刈り手法では実験的にデータ数の対数に比例する程度の実行時間であった。

1 はじめに

文書から文字列を検索する際、検索語のヒット位置の周辺文脈は有用な情報となる。例えば“consist”で検索すれば“consist of”などの複合表現が、「徳川」で探せば歴代将軍の名前が、県名で探せばその県の都市名がわかるかもしれない。

こうした目的に対して、KeyWord in Context (KWIC) が古くから知られている。KWIC はヒット位置の後方あるいは前方の全文脈集合をアルファベット順に並び替えて表示する方法である。似た文字列から始まる文脈はまとまった位置に表示されるため、前後文脈の出現傾向を捉えるのに有用である。しかし、KWIC はヒット数が増えると、一度に全てを概観できなくなり、全体の傾向を捉えられなくなる。そのため、文脈をある程度まとめた上で表示する方法が必要になる。

本稿では面積最大化原理に基づく前方または後方の文脈集合の集約方法を提案する。基本のアイデアは以下の通りである。検索ヒット位置に対応する文脈集合が与えられたときに、最大 K 行 L 文字の範囲に圧縮して表示するとしよう。ある文字列 s を表示する場合、全文脈集合のうち s を接頭辞にも

つ部分をカバーすると考える。そして、最大 K 個の文字列集合 S の中で、カバーする面積の合計が最大になる S^* を探索する。本手法は表示する文字列の長さを自動的に選択し、また実験的にもほどほどの長さの単語分割位置まで表示することがわかった。また、最適な文字列集合の探索を行うための動的計画法のアルゴリズムと、それを高速化させるために上限値の見積もりによる枝刈り手法も提案する。全文検索と最適文字列集合の探索を高速に行うために、接尾辞木の各ノードを頻度順に並び替えた頻出接尾辞木という新しいデータ構造に関しても解説する。

関連研究として、山本らの KIWI が上げられる [5]。この手法は文脈を単語単位に切り分け、頻出する後続単語を表示することで用例をまとめ上げている。単語などの意味のあるアルファベット列の直後では後続する文字種数が多いことに着目し、言語に依存しない単語区切りを実現している。KIWI は表示する文脈単体の評価手法なので、複数文脈の最適な組み合わせを求めることはできない点で本研究と異なる。また、KIWI の採用したようなスコア関数を本研究の手法のスコア関数として採用し、スコアの総和を最大にするという応用も考えられる。

2 文脈探索アルゴリズム

2.1 面積最大化原理

本研究の基本アイデアである、面積最大化原理について説明する。ここでは、後方、あるいは前方のいずれか一方の文脈のみを表示する問題を解く。なお、解説は後方文脈を仮定するが、前方文脈に関しても、入力文書を逆向きにすることで自然に解くことができる。

文書集合に対して文字列検索を行ったときに得られる文脈、すなわち全ヒット位置から始まる後方文字列の集合を文脈 C と呼ぶ。ある文字列 s が文字列集合 C をカバーする面積を $LEN(s) \times PREF(s, C)$ で定義する。ただし、 $LEN(s)$ は文字列 s の長さを、 $PREF(s, C)$ は C 中で s を接頭辞としてもつ要素の数である。これは、図 1 の例で示すとおり、全文脈集合中で文字列 s がカバーする面積である。表示行が K 行に限られている場合、最大 K 個の文字列 s を表示することができる。そこで、元の全文脈集合をカバーする面積が最大になる K 個の文字列を選択することで、文脈に対して最大の情報を与えることができる。これは、以下であらわされる S^* を選択する問題となる。

$$S^* = \arg \max_S \sum_{s \in S} LEN(s) \times PREF(s, C)$$

ただし、以下の制約を課すことで、類似文脈が表示されないようにする。

制約 (1)

S 中の任意の 2 要素 $s_i, s_j \in S$ に関して s_i は s_j の接頭辞とはならないものとする。

この制約により、例えば「ボタン」というクエリに対して、「を」「を押」「を押す」のような類似した文字列が同時に選択されなくなる。

単純にこれを実装しようとすると、各 C の要素に対して部分文字列の候補が L 通り存在し、その中から K 個を選び出すので、 S の候補は $|C|L^K$ 存在する。従って、単純に列挙するだけでは現実的な時間で計算することはできない。

全文脈 C (KWIC)	文字列 s	面積
ボタンを押してください。	ボタンを押: 10=5×2	
ボタンを押す。	ボタンを: 12=4×3	
ボタンをクリックします。	ボタン: 15=3×5	
ボタンは押せません。		
ボタンは消えます。		

図 1 文脈と面積の例。

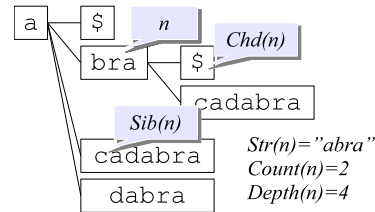


図 2 文脈木とそれに対する操作の例。

2.2 文脈木

与えられたクエリにヒットした位置に対する文脈集合 C に対して、TRIE を構築する。こうして得られた木構造を、文脈木と呼ぶことにする。各ノードは順序付けされているものとする。

文脈木の各ノード n に対して、以下の操作が定数時間で可能である。図 2 に文脈木の例とともに、各操作を示した。

ROOT ルートノードを示す。

CHD(n) n の最初の子ノードを返す。

SIB(n) n の弟ノード、すなわち n と親ノードが同じ次のノードを返す。

COUNT(n) n の子孫の葉ノードの総数を返す。

STR(n) ルートノードから n までのパス上の文字を並べた文字列を返す (n 中の文字も含む)。

DEPTH(n) STR(n) の長さを返す。

面積最大の文字列集合を得る上で、このデータ構造は有用である。まず、 $PREF(STR(n), C) = COUNT(n)$ である。これは、STR(n) を接頭辞とする文字列は、全てノード n の子孫であるからである。次に $LEN(STR(n)) = DEPTH(n)$ である。これは定義より明らか。従って、文字列 STR(n) がカバーする面積は、 $COUNT(n) \times DEPTH(n)$ で計算できる。また、選択される文字列集合に対する制約

1 は、「子孫関係にある 2 ノードは選択されない」という条件と等価である。したがって、最大 K 個の文字列を選択する問題は、文脈木から最大 K 個のノードを選択する問題を解けばよい。

2.3 動的計画法による解法

面積を最大化させる最大 K 個の文脈木上のノードを選択する問題の動的計画法による解法を示す。文脈木に対して最大面積を再帰的に計算しよう。関数 $f(n, k)$ は、ノード n とその子ノードと弟ノードの中から、最大 k 個のノードを選択したときの最大面積を返す関数である。最大 K 行を表示する場合、最大面積は $f(\text{ROOT}, K)$ で与えられる。さて、制約 1 により $f(n, k)$ は以下の場合分けの中で最大の値である。

- ノード n に対応する文字列 $\text{STR}(n)$ を選択し、残りの $k - 1$ 個は弟ノードから選択する
- $c \in \{0, \dots, k\}$ に対して、 c 個は n の子ノードから、残りの $k - c$ 個は n の弟ノードから選択する

制約 1 により、 n を選択したときはその子ノードからは選択できないことに注意する。以上を再帰的な式に書くと、以下の通りである。

$$f(n, k) = \max\{s(n) + f(\text{SIB}(n), k - 1), \max_{c \in \{0, \dots, k\}} (f(\text{CHD}(n), c) + f(\text{SIB}(n), k - c))\}$$

以上をメモ化再帰でそのまま実装した擬似コードを図 3 に示した。ただし、この再帰は木を 1 回たどるように変形することができる。この擬似コードを図 4 に示した。こちらの方がメモリ効率がよい。

全ノード数を N とすると、1 回の呼び出しで最大 K 回のループがあるので、時間計算量は $O(NK^2)$ である。通常の設定では K は例えば 10 程度の非常に小さい値である。一方で、 N は検索ヒット数に比例するため、扱う文書数が増えると非常に大きくなる可能性がある。

2.4 上限の見積もりによる枝刈り

前節で最大面積は検索ヒット数に対して線形時間で計算可能なことがわかった。しかし、文書が大規模化すると検索ヒット数は膨大になるため、十分な

```

1 functoin f(n, k):
2   if n = null:
3     return 0
4   if (n, k) in memo:
5     return memo[n, k]
6   a = s(n) + f(sib(n), k - 1)
7   for c in {0, ..., k}:
8     cs = f(chd(n), c)
9     ss = f(sib(n), k - c)
10    a = max(a, cs + ss)
11  return memo[n, k] = a

```

図 3 メモ化再帰による面積最大化法の擬似コード。

```

1 functoin f(n):
2   if n = null:
3     return {}
4   rc = f(chd(n))
5   rs = f(sib(n))
6   for k in {0, ..., K}:
7     a = s(n) + rs[k - 1]
8     for c in {0, ..., k}:
9       cs = rc[c]
10      ss = rs[k - c]
11      a = max(a, cs + ss)
12   r[k] = a
13  return r

```

図 4 メモリを消費しない面積最大化の擬似コード。

速度を得られない。そこで効率化のために探索を枝刈りする方法を示す。

基本的なアイデアは、ノード n が占める面積の上限を見積もり、これが現在わかっている最大値を超えないことがわかった時点で探索を打ち切ることである。まず、関数 f に、超えなければならない下限値 m を渡した新しい関数 g を設計する。関数 $g(n, k, m)$ は、 $f(n, k) > m$ のとき $f(n, k)$ を返し、それ以外のときは m 以下の任意の値を返す関数である。そのため、 $f(\text{ROOT}, K)$ は $g(\text{ROOT}, K, 0)$ で求められる。関数 g は、 $f(n, k) \leq m$ であることがわかったら、ただちに 0 を返してよい。すなわち、 $u(n, k) \geq f(n, k)$ なる上限関数 u を設計し、

$u(n, k) \leq m$ なら処理を打ち切る .

打ち切りが可能な場合は複数存在する . 1 つは g の呼び出しの最初である . $u(n, k) \leq m$ であれば , $f(n, k) \leq m$ なので探索せずに 0 を返してよい . もう一方は子ノードと兄弟ノードの組み合わせの探索時である . 探索中に子ノードから最大 c 個 , 弟ノードから最大 $k - c$ 個選択するとき , 弟ノードの面積の上限は $u(\text{SIB}(n), k - c)$ なので , 子ノードは $m - u(\text{SIB}(n), k - c)$ を超える必要がある . また , 子ノードのカバーする面積 cs に対して , 弟ノードは $m - cs$ を超える必要がある .

打ち切ったときの情報は次の打ち切りに活用できることにも注意する . 引数 $g(n, k, m)$ が打ち切られたということは , $f(n, k) \leq m$ であることを意味している . つまり , 探索に失敗したとき m が $f(n, k)$ に対する上限になっている . 上限値を管理するテーブル `upper` を用意して , 計算のたびに更新する . この値は u の計算のときに活用することができる . 以上を取り入れたアルゴリズムの擬似コードを図 5 に示した .

枝刈りを効率よく行うためには , 探索順に関して工夫が必要である . g は任意の順番で探索を行っても同じ値が返ってくるが , なるべく最大値をとる場合から順に探索した方が枝刈りの効率が良くなる . なぜなら , 先に最大値を見つけ出せば , 上限に関する条件が厳しくなるからである . 経験的に , ノード n を選ぶとき (図 5 中 9 行目) , そして子ノードから選択する個数 c を小さい順にループさせる (同 11 行目) と早く最適解にたどり着く . より効率的な探索順も存在するかも知れないが , それは今後の課題とする .

2.5 上限値の計算

上限値 $u(n, k)$ はなるべく正確な値 $f(n, k)$ に近い方が良いが , 時間がかかりすぎると枝刈りの効果が薄い . そのため , なるべく効率よく計算できる u を設計する . 再帰を一切行わずにわかる上限は 2 つある . 1 つは , 最悪ケースの見積もりである . 面積が最大となるのは , 全ての子ノードに分岐が一切無い場合である . このとき , n を含む n の兄弟ノードの内 , 頻度の高い k 個がそれぞれ長さ L まで選

```

1  functoin g(n, k, m):
2      if n = null:
3          return 0
4      if (n, k) in memo:
5          return memo[n, k]
6      if u(n, k) <= m:
7          upper[n, k] = u(n, k)
8          return 0
9      s = s(n) + g(sib(n), k - 1, m - s(n))
10     a = max(m, s)
11     for c in {0, ..., k}:
12         su = u(sib(n), k - c)
13         cs = g(chd(n), c, a - su)
14         if cs <= a - su:
15             continue
16         ss = g(sib(n), k - c, a - cs)
17         a = max(a, cs + ss)
18     if a > m:
19         return memo[n, k] = upper[n, k] = a
20     else:
21         upper[n, k] = m
22         return 0

```

図 5 上限値による枝刈りを取り入れたアルゴリズムの擬似コード .

```

1  function u(n, k):
2      u = 0, m = n
3      for i = {1, ..., k}:
4          u += s(m) * L
5          m = sib(m)
6      for k' in {k, ..., K}:
7          u = min(u, upper[(n, k')])
8      return u

```

図 6 上限値関数の擬似コード .

択される . この上限を効率的に計算するために , あらかじめ文脈木の子ノードは頻度順にソートされていると都合が良い . 頻度順になっていれば頻度上位 k 個の兄弟ノードは n の弟ノードを k 個たどるだけで取得できるからである . もう 1 つ簡単にわかる上限は , 動的計画法のテーブルを見る方法である . 任意の n, k に対して , $f(n, k) \leq f(n, k + 1)$ が

成り立つので，与えられた k より大きい値に関して計算済みか調べ，計算済みなら上限値として利用する．上限は真の値に近い方が，つまり小さい方が良いので，以上の中での最小値を利用する．

以上の上限值関数 u の擬似コードを図 6 に示した．実装上は関数 u を呼び出すたびに最大 K 回のループをまわさないよう， g の呼び出しの最後で $upper$ の最小値を更新した方が高速である．

2.6 頻度順接尾辞木

最後の問題は，頻度順に並び替えた文脈木をいかにして高速に構築するかである．単純に検索を行ってから文脈木を構築すると，最低でも葉ノード数に比例した時間がかかる．従って検索ヒット数に比例した時間がかかる．これでは枝刈りの恩恵を受けれない．

この問題を解決する方法は，あらかじめ文脈木を作っておくことである．KWIC を高速に構築する方法の一つが接尾辞木を構築することである [1]．長さ n の文字列 s の接尾辞木とは， s 中の全ての位置 $i \in \{0, \dots, n-1\}$ における全接尾辞を，TRIE 状の木で表現したデータ構造である．一般的に分岐の無いノードの列は一つにまとめられた形にする．接尾辞木を利用すると検索と後方文脈集合の構築が用意に可能である．クエリ文字列を接尾辞木に対して順番にたどっていったときに到達するノードをルートとする部分木は，全後方文脈集合に対する TRIE，すなわち文脈木となっている．これはすなわち，考えうる文脈木があらかじめ構築されているといえる．

そこで，接尾辞木の各子ノードを葉ノード数で並び替える．こうしてできたデータ構造を，頻度順接尾辞木 (Frequency-Ordered Suffix Tree; FOST) と呼ぶことにする．頻度順接尾辞木と接尾辞木の例を，図 7 に示した．頻度順接尾辞木に対してクエリ文字列をマッチさせて到達するノードをルートとする部分木は，接尾辞木同様クエリの全後続文字列になっているので，文脈木となっていることがわかる．しかも，子ノードは頻度順に並んでいるので前節で示した枝刈りに利用できる．実装上はメモリ中に文脈木を再構築することなく，探索に必

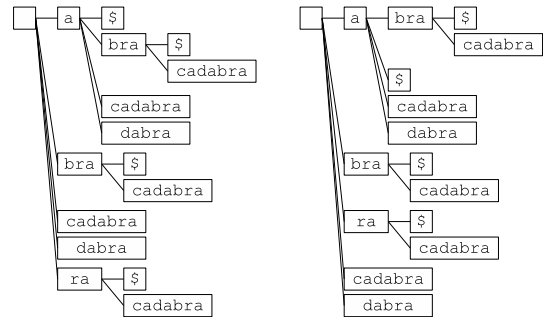


図 7 “abracadabra” に対する接尾辞木 (左) と頻度順接尾辞木 (右)．頻度順接尾辞木が葉ノードの総数で並び替えること以外は共通．

要なノードだけたどればよい．

頻度順接尾辞木の構築は接尾辞木の構築と同等の処理で行える．子ノードリストを構築する際に頻度順に並べ替えることで行われる．ノード数は文書長に比例する．また，各ノードに対して子ノードは最大でもアルファベット数 $|\Sigma|$ と同数存在する．従って，最悪時の計算量は $O(N|\Sigma| \log |\Sigma|)$ である．

3 実験

3.1 実装

頻度順接尾辞木の構築は，まず接尾辞配列を構築し，それを元に接尾辞木を構築する．木のノードを構築する際に，葉ノードの総数でソートすることで頻度順接尾辞木となる．接尾辞配列の構築には SAIS [4] の実装である，Mori の実装を利用した [3]．構築された接尾辞配列の最大共通接頭辞長を計算すると，接尾辞配列を順番にたどることで，対応する接尾辞木を帰りがけ順にたどることができる [2]．これを利用して接尾辞木を構築する．また，最大共通接尾辞長は線形時間で構築できる [2]．

IO の影響を少なくするために，実験では接尾辞木をメモリ中に展開して利用した．しかし，帰りがけ順にたどりながら HDD 上に接尾辞木を構築すれば，親子関係にあるノードを HDD 中の近い位置に配置することができる．そのため実用上は HDD 上にデータを置いたまま探索しても十分なパフォーマンスを得ることができる．

表 1 コーパス一覧 .

略称	説明	サイズ	言語
san	産経新聞 1 年分	78 MB	日本語
ptb	Penn Treebank	45 MB	英語

表 2 コーパスごとの頻度順接尾辞木の比較 .

データ	サイズ	構築時間	ソート	ノード数
san	257 MB	61.7 sec	4.0 sec	13.6 M
ptb	178 MB	27.5 sec	1.1 sec	11.6 M

3.2 実験設定

利用したコーパスの一覧を表 1 に示した . 日本語と英語の両方を利用した . 実装には Java 言語を使用し , JVM として IBM Java version 1.6.0 を利用した . 利用した計算機は CPU が Intel の Xeon 3.6GHz , 主記憶 8GB の Linux 機である .

3.3 頻度順接尾辞木の構築

各コーパスに対して頻度順接尾辞木の構築を行った . 構築された接尾辞木ファイルのサイズと構築時間を表 2 に示した . 構築時間全体に占めるソートの割合は 1 割に満たない程度であった . 従って , 通常の接尾辞木を構築するのに大差ない時間で構築可能であった .

3.4 探索で見つかる文字列

面積最大化に従って得られた文字列集合と , 単純な KWIC の結果とを比較した . 表 3 は ptb に対して “New” と “day” で , san に対して “衆” と “議会” で検索し , それぞれの手法で集計した結果である . “New” と “衆” に関しては後方文脈で , “day” と “議会” に関しては前方文脈で集計した . ptb に関しては文字列長 $L = 15$, san は $L = 10$ とした . 選択する文字列数はいずれも $K = 10$ とした . KWIC は , 各クエリ文字列の後方 L 文字を取得して頻度順に並べた上位 K 件の結果である . 面積最大化では類似表現をまとめるため , 頻度順文脈木中での出現位置の順に並べてある .

まず , 面積最大化手法ではほとんどが意味のありそうな単位で切れている . KWIC は固定長のためその傾向がない . また , KWIC では高頻度の文字

表 3 面積最大化と KWIC での結果比較 . 空白文字は $_$ で示す .

面積最大化	頻度	KWIC	頻度
New (following)			
New York $_$	1222	New York Stock Exc	289
New York,	223	New York Federal R	13
New York.	149	New York banks on	13
New York-based	43	New York Mercantil	12
New England	121	New York, gold for	11
New Jersey	130	New York investmen	10
New Hampshire	79	News & World Repor	10
New Orleans	88	New York trading y	9
New Mexico	44	New York Times Co.	9
Newport	56	Newark to Clevelan	9
day (previous)			
$_$ day	3819	trading yesterday $_$	89
yesterday	977	very within 30 day	26
Saturday	229	the end of the day	19
today	917	s closed yesterday	17
Friday	800	ite trading Friday	16
Tuesday	318	ou have a good day	14
Wednesday	229	or the past 30 day	13
Thursday	213	d with the 300-day	13
Monday	468	it with the 30-day	13
Sunday	285	lion barrels a day	12
衆 (following)			
衆院議員 $_$	1475	衆院有事法制特別委員会	46
衆院議運委員長	148	衆院議員、鈴木宗男被告	34
衆院議院運営委員	119	衆院選でどの党に投票す	32
衆院予算委	429	衆 5 5 0 0 0 巨 $_$	28
衆院選	310	衆院選挙区画定審議会が	24
衆院本会議	258	衆院議員 (自民党を離党	22
衆院第 1 委員室	59	衆 4 8 0 0 0 ダ $_$	18
衆院和歌山 2 区	67	衆院第 1 委員室を出て、	17
衆院有事法制特別委	62	衆院議運委員長の証人喚	15
衆 $_$	852	衆院議員の鈴木宗男容疑	12
議会 (previous)			
審議会	724	世界ボクシング評議会	26
協議会	657	救出するための全国協議会	19
世界ボクシング評議会	52	$_$ 都議会	13
パレスチナ評議会	30	教科用図書検定調査審議会	11
国家平和発展評議会	19	生労働省の社会保障審議会	10
県議会	316	衆院選挙区画定審議会	8
、議会	295	府の衆院選挙区画定審議会	8
都議会	186	ーバのカストロ国家評議会	7
市議会	155	フジテレビ番組審議会	7
連邦議会	77	諮問機関「行革断行評議会	7

列 , 例えば “New York” や “審議会” が候補の大部分を占めてしまっている . 実際は高頻度の文字列である “New England” や “県議会” などが一切含まれていない . 面積最大化ではこうした文字列を集約して頻度を数えるため , 抜けが少なく , また取得された出現頻度も KWIC に比べて大きな数字になっている . 全体を通じて , 面積最大化によって取得された文字列集合の方が , KWIC よりも周辺文脈を的確に集約していることが言える .

3.5 枝刈りによる高速化の効果

枝刈りを使った場合と , 単に動的計画法を使った場合とで速度を比較した . 図 8 は , 各クエリ単語に

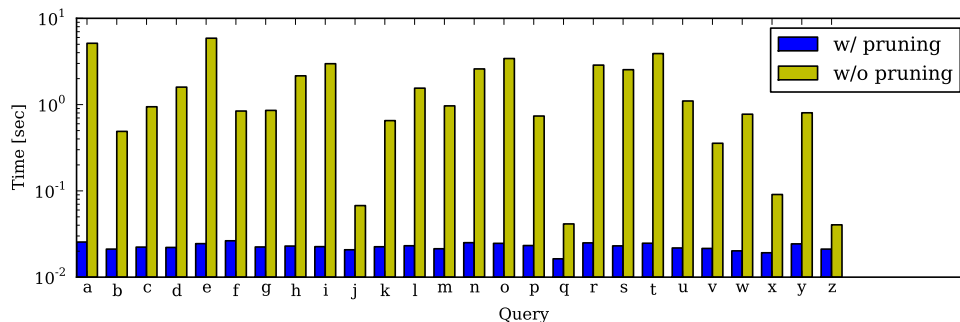


図 8 枝刈りあり (w/ pruning) となし (w/o pruning) での各クエリに対する実行時間 ($L = 15, K = 10$) .

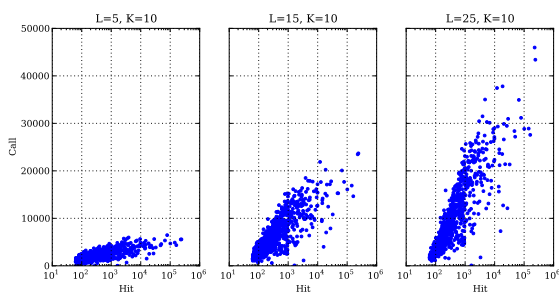


図 9 検索ヒット数と関数呼び出し回数の関係 .

対して計算に係った時間を 2 つの手法間で比較した図である . 横軸がクエリ文字列 , 縦軸が実行時間である . 対象データとして ptb を利用し , クエリとして 26 種類のアルファベットを , 1 文字ずつ与えた .

グラフから最大で 100 倍以上の高速化を果たしていることがわかる . 特にヒット数が多いときに両者の実行速度の違いは顕著となる . これは枝刈りが有効に働いていることを示す . 特に今回利用したデータは 50 MB 程度の小さいものであったため , より巨大なデータでは差がさらに大きくなることが予想される . 一方でヒット数が少ないときは差が小さい . ヒット数をもっと少ない場合は枝刈りのオーバーヘッドによって速度が逆転されることも考えられる . しかし , そもそもヒット数が少ないときは十分な速度が得られるので大きな問題にはならないと予想される .

3.6 検索条件の違いの影響

探索に係る時間を , L と K の設定を変えて比較した . 対象として , ptb を利用し , 空白文字で挟まれた文字列の出現頻度上位 5000 単語の内 5 件おき , 1000 件をクエリとして与えた結果である . 横軸に検索ヒット数 , 縦軸に再帰関数の呼び出し回数をプロットした .

まず , 図 9 がヒット数と関数呼び出し回数のプロットである . 横軸を対数軸にしてある . 実験条件を変えるため , $K = 10$ で固定して , $L = \{5, 15, 25\}$ と変化させた . ばらつきが大きい , およそその設定でもヒット数の対数に対して関数呼び出し回数は線形に近い形をしている . すなわち , ヒット数の対数におよそ比例することがうかがえる .

図 10 と 11 は , それぞれ $L = 15$ と $K = 10$ で固定したときのプロットを , 縦軸に対しても対数軸にしたものである . それぞれ , K と L を $\{5, 10, 20\}$ としたときのプロットを重ねて表示した . すると , いずれの場合も K や L の値を 2 倍にするにつれて , 対数プロット上で同程度の量だけ上にシフトしていることがうかがえる . この傾向は , 他のデータに対して行った実験でも観察された . 従って , おおよそ K と L に対して , 指数時間以下の時間計算量で計算が完了していることが予想される .

4 まとめ

文字列検索結果の周辺文脈を纏め上げ , 高頻度の表現を制限された領域内に最も効率的に表示する

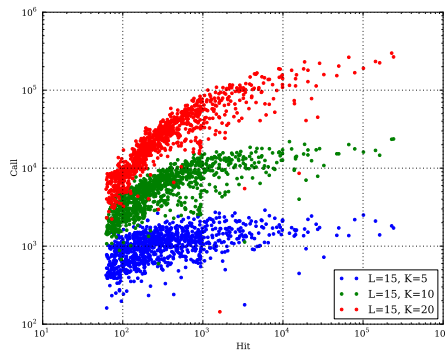


図 10 $L = 15$ のときの K に対する検索ヒット数と関数呼び出し回数の関係 .

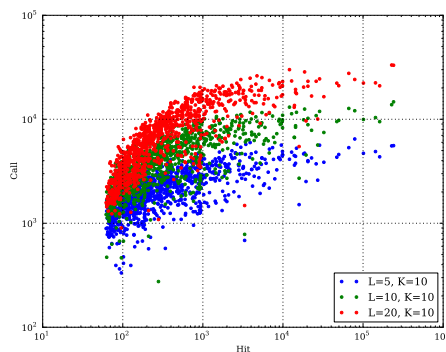


図 11 $K = 10$ のときの L に対する検索ヒット数と関数呼び出し回数の関係 .

手法を提案した．特に，面積最大化原理に基づく定式化と，その動的計画法による多項式時間の解法を示した．また，巨大な文書に対しても適応できるように効率化するための，上限値の見積もりによる枝刈りと，それを実現するための頻度順接尾辞木というデータ構造に関して解説した．実データによる実験を行い，単純な KWIC による文脈表示より意味のある単位で集計されること，及び枝刈り手法によって劇的な速度向上を果たしたことを確認した．今後の課題として，計算量に対する理論的な裏づけや，より効率的な枝刈りのヒューリスティックスの開発が挙げられる．

参考文献

- [1] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN: 0-521-58519-8.
- [2] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, pp. 181–192, 2001.
- [3] Yuta Mori. [sais](http://sites.google.com/site/yuta256/sais). <http://sites.google.com/site/yuta256/sais>.
- [4] Ge Nong, Sen Zhang, and Wai Hong Chan. Two Efficient Algorithms for Linear Suffix Array Construction. *IEEE Transactions on Computers*. To appear.
- [5] 山本真人, 田中久美子, 中川裕志. 検索エンジンに基づく多言語用例指南ツール: KIWI. 言語処理学会第 9 回年次大会発表論文集, pp. 15–18, 2003.