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

2005－BIO－3（14）
2005／12／22

文書頻度計数の線形時間アルゴリズムの実装と評価

尾　田　　晃[†]　梅　村　恭　司[†]

統計的自然言語処理において、文字列を特徴付ける統計量として、文字列の出現頻度や、文字列の表れる文書の総数などが用いられる。しかし、単純な方法では文書集合中の全ての任意文字列に対して、これらの統計量を計算するには $O(N^2)$ のメモリを必要とし、大規模なコーパスに対しては実際的でない。本稿では Range Minimum Query のアルゴリズムを用いて $O(N)$ のメモリ空間と $O(N)$ の時間で全ての任意文字列の重複条件付き文書頻度を求めるアルゴリズムを実装し評価する。

An implementation and evaluation for counting number of documents that contain substrings

Akira ODA [†] and Kyoji UMEMURA[†]

In statistical natural language processing, the number of substring in document collection and the number of documents that contain substrings is used for substring's feature. It is hardly that to compute document frequency for all substring in document collection by simple method, because it requires $O(N^2)$ memory space. This paper reports an implementation and evaluation of this problem by using Range Minimum Query Algorithim that requires $O(N)$ preparation time and $O(1)$ time each query.

1. はじめに

文書集合中の任意文字列を特徴付ける量として、文字列を k 回以上含む文書の頻度がある。本稿では、これを重複度 k の文書頻度と呼び、特に k を指定しない場合を重複条件付き文書頻度と呼ぶ。この重複条件付き文書頻度はキーワード抽出などの際に、有用であることがわかっている。しかし、単純な手法で文書集合中のすべての任意文字列の重複条件付き文書頻度を求めるには、すべての文書の文字列長の総和 N に対して、$O(N^2)$ の表が必要となり、実際的ではない。これに対し、Suffix Array とクラス分類により、$O(N)$ のメモリと $O(n \log n)$ の時間ですべての任意文字列の重複条件付き文書頻度を求めるアルゴリズムが提案されている。[1)]本稿ではこの重複条件付き文書頻度計数アルゴリズムに、Range Minimum Query アルゴリズムを組み合わせることにより、$O(N)$ のメモリと $O(N)$ の時間ですべての任意文字列の文書頻度を求めるアルゴリズムを提案する。

† 豊橋技術科学大学 情報工学系
Department of Information and Computer Science, Toyohashi University of Technology

2. Suffix Array

Suffix Array はあるテキストに対して、任意の位置から末尾までの文字列 (接尾辞:suffix) を考え、その集合を辞書順に整列したものである。ここで、テキストがすべてメモリ上にあれば、Suffix Array は文字列そのものではなく、文字列の開始位置のみを記憶しておけばよい。このため、テキストの大きさ N に対して、$O(N)$ のメモリ空間を用意すれば、任意に文字列に対して、その出現位置を二分探索を用いて $O(\log N)$ で検索することができる。“abcabedcabcde” という文字列に対して Suffix Array を構築した場合の例を**図 1** に示す。

Suffix Array を単純な手法で構築した場合、suffix 辞書順比較は最悪で $O(N)$ の計算時間がかかるので最悪で $O(N^2 logN)$ の時間がかかるが、$O(N)$ の時間で Suffix Array を構築するアルゴリズムが Kärkkäinen らによって提案されている。[2)]

3. Longest Common Prefix

Suffix Array 上の文字列は辞書順に並んでいるため、Suffix 上の隣り合う文字列は先頭部分が共通であることが多い。そこで、$suffix_i$ と $suffix_{i+1}$ の文字列の共通する先頭部分の文字列の長さを lcp_i(Longest

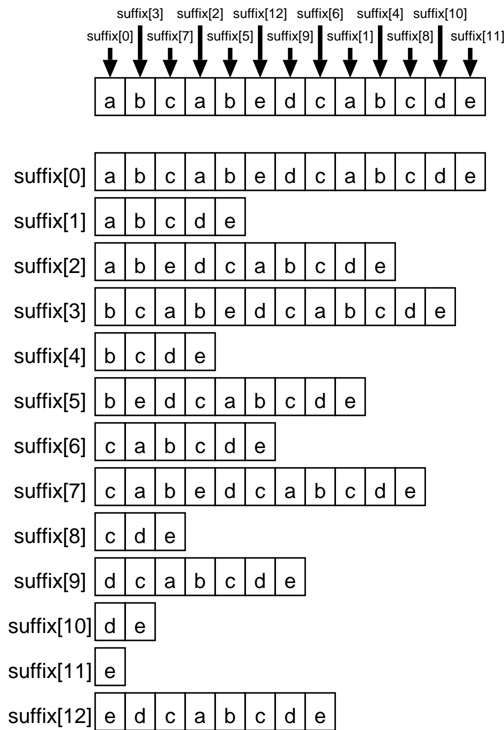

図 1 Suffix Array の例
Fig. 1 Example of Suffix Array

Common Prefix) と定義する。*lcp* は後述するクラスを求める際に使用される。単純に手法で求めると、最悪で $O(N^2)$ の計算量になってしまうが、Kasai らにより $O(N)$ の計算時間で *lcp* を計算するアルゴリズムが提案されている。[3]

4. 重複条件付き文字列頻度および文書頻度

すべての文字列の出現は、Suffix Array 上の suffix の位置と関連付けることができる。ここで、ある位置 i における文字列 x の重複度は suffix の順序でその位置 i 以下であり、かつ、同一文書内に含まれる、文字列 x の出現の回数と定義する。

d_0 = “Heigh_Ho_Heigh_Ho”, d_1 = “Heigh_Ho”、d_2 = “Heigh” とした場合に文字列 “H” の重複度 k を求めた例を**図 2** に示す。なお、図中の document_id 対応する suffix の属する文書の番号をあらわす。

重複条件付き文字列頻度 $cf_k(x)$ は文書集合内で重複度が k 以上のもの文字列 x の出現回数と定義される。また、ある文書 d 内に文字列 x が出現する回数を $tf(d,x)$ とすれば、重複条件付き文字列頻度 $df_k(x)$ は以下のように定義される。

$$df_k(x) = |\{d_i | tf(d_i, x) \geq k\}|$$

重複条件付き文字列頻度 $cf_k(x)$ と重複条件付き文字列頻度 $df_k(x)$ の間には以下の関係が成り立つことが知られている。[1]

$$df_k(x) = cf_k(x) - cf_{k+1}(x)$$

このため、重複条件付き文書頻度を計算する問題は重複条件付き文字列頻度を計算する問題に帰着する。

i	*suffix*	document_id	*k*
3	Heigh	2	1
4	Heigh_Ho	0	1
5	Heigh_Ho	1	1
6	Heigh_Ho_Heigh_Ho	0	2
7	Ho	0	3
8	Ho	1	2
9	Ho_Heigh_Ho	2	2

図 2 “H” に対する重複度の例
Fig. 2 Example of overlap count in “H”

4.1 計数処理

重複条件付き文字列頻度の計数を行う場合、計数用のカウンタを複数用意し、Suffix Array 上の各 suffix に対して対応するカウンタを加算することで行う。文献 1)、4) ではこの加算する対象を決定する際に、二分探索を行う。この二分探索の対象は最大で N 個の配列になるので、最悪のケースではカウンタの決定に $O(\log N)$ の計算時間が必要になる。この場合、文書集合全体の処理は $O(N \log N)$ となる。

$O(N)$ の文書頻度計数アルゴリズム[5] と文献 1), 4) を対比しながらすると、この二分探索は *lcp* の変化パターンに対する区間最小値の問い合わせ (RMQ, Range Minimum Query) と等価と判明した。RMQ は $O(N)$ の前処理を行うことで、問い合わせごとに $O(1)$ の計算量で計算できることがわかっている[6]。これから、加算カウンタの決定に $O(1)$、計数処理全体は $O(N)$ で行うことができる。

5. 線形時間文書頻度計数の計算時間

文献 1) から、文書集合全体に含まれるすべての任意文字列の重複条件付き文字列頻度を計数するには、事前に Suffix Array、Longest Common Prefix、Document Link[4] の計算を行う必要があるが、これらはそれぞれ $O(N)$ で計算することが可能である[2)〜4)]。

さらに、重複条件付き文字列頻度の計数処理は大きさ N の Suffix Array を 1 度走査し、各 suffix 上で

表 1　文書集合の大きさ
Table 1　A size of document collection

結合数	文字数
1	11874
3	35335
10	119154
31	374911
100	1195348
316	3868042
1000	12186761

加算カウンタの決定を行うことになるが、前述したとおり、RMQ を用いれば加算カウンタの決定は $O(1)$ で計算することができる。よって、計数処理における Suffix Array の走査は $O(N)$ の計算時間に収まる。また、文献 5) では、Suffix Tree を使用するが、本稿では Suffix Array 上で文書頻度を計数する。

これらを用いて、重複条件付き文字列頻度を $O(N)$ で計算するアルゴリズムを実装した。

6. 実行時間の計測

システムの比較を行うために、実際に重複条件付き文書頻度計数を行うプログラムの実行時間を計測する。実験には Opeteron 240 Dual, 8GByte メモリのシステムを使用した。

6.1 ベースラインシステム

比較用のベースラインシステムとして、文献 1) に基づく文書頻度計数プログラムを用いる。このプログラムは加算カウンタの決定に二分探索を用いている。以下、このシステムを base と呼ぶ。

6.2 提案システム

提案するシステムは加算カウンタの決定に RMQ を使用したものである。二分探索の代わりに RMQ を使用する以外はベースラインのシステムとまったく同じ実装を用いる。以下、このシステムを rmq と呼ぶ。

6.3 文 書 集 合

実験に使用する文書集合は何でもいいが、本稿では論文アブストラクトを使用した。文書長が変化した時の実行時間の計測を観測するために、文書数を 50 文書で固定し、m 個の文書を 1 つに結合して 1 文書として扱っている。以下、この文書集合を abst と呼ぶ。**表 1** にこの文書集合の大きさを示す。なお、表中の結合数は m を表す。

また、上記論文アブストラクトの文書集合中のすべての文字を “A” に置き換えた文書集合に対しても実験を行う。以下、この文書集合を same と呼ぶ。このような文書集合の場合、base システムでは二分探索の対象がかなり大きくなる。

結合数	base[sec]	rmq[sec]
1	0.01	0.01
3	0.01	0.02
10	0.04	0.10
31	0.10	0.27
100	0.36	0.97
316	1.14	2.95
1000	3.87	9.17

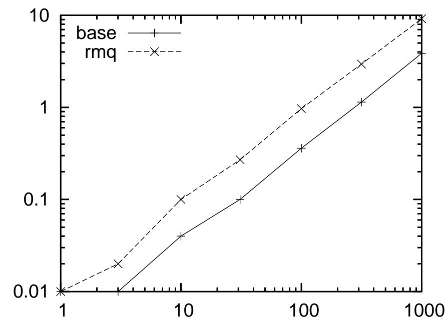

図 3　論文アブストラクトに対する実験
Fig. 3　Experiment in abstracts of papers

表 2　二分探索の対象配列の大きさの平均
Table 2　An average size of binary search target array

結合数	abst	same
1	1.33	6.53
3	1.78	8.05
10	2.62	9.78
31	3.56	11.43
100	4.21	13.10
316	4.43	14.80
1000	4.52	16.45

6.4 計　　測

上記 2 種類の文書集合に対して、df_1、df_2、df_3、df_4 を一度にまとめて計数した。**図 3**、**図 4** に abst および same における、実行時間を示す。計数には文書集合全体の Suffix Array、Longest Common Prefix、Document Link を計算しておく必要があるが、ここではそれらの計算にかかる時間は除いて、重複条件付き文字列頻度の計数部分のみについて計測している。

abst に対する実験ではすべての場合において、rmq の実行時間は base より長くなっていることがわかる。これに対し、same でも文書長が短い場合は、base より rmq の実行時間の方が長くなっているが、文書長が長くなるにつれ、rmq の方が base より実行時間が短くなるのがわかる。

結合数	base[sec]	rmq[sec]
1	0.00	0.01
3	0.01	0.02
10	0.06	0.06
31	0.19	0.23
100	0.65	0.68
316	2.37	2.19
1000	7.91	6.86

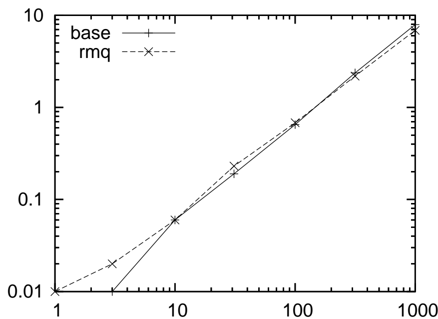

図 4　"A" のみで構成される文書集合に対する実験
Fig. 4　Experiment in document collection that contains only "A"

7. 考　察

位置 i における二分探索の対象配列の大きさを $level_i$ ととして、$\mathrm{E}[\log_2 level_i]$ を計算すると**表 2** のようになる。この表からわかるとおり、same のようなケースでは二分探索の対象配列が非常に大きくなる傾向があるが、逆に、自然言語である abst に対してはさほど大きくなっていない。使用した RMQ のアルゴリズムは $O(1)$ で問い合わせが出来るが、1 回の問い合わせに多くのメモリ参照と比較を必要とするので、abst の用に数百程度の配列を二分探索する方が計算量が少なくて済む。

このため、二分探索対象の配列があまり大きくならないようなケース、つまり文字列クラスの階層[4] があまり大きくならないケースにおいては、本稿で述べた RMQ を使用するアルゴリズムを用いるより、二分探索を使用するアルゴリズムを用いたほうが良いといえる。

ただし、これは使用した RMQ はナイーブな実装であり、RMQ アルゴリズムの改善があれば、実行時間の改善につながる。また、今回使用できるメモリ量の制限上、実験できる文書集合のサイズに制限があったが、非常に大きな文書集合を扱う場合は上記の $\mathrm{E}[\log_2 level_i]$ が大きくなるため、RMQ アルゴリズムを用いた手法を用いることで、実行時間の改善につながると考えられる。

8. ま　と　め

文献 1)，4) で示されたアルゴリズムの一部を変更して $O(N)$ の文書頻度計数アルゴリズムを実装した。実際的なデータでは、プログラムの実行時間は長くなったが、最悪のケースで実行時間の改善効果があることを確認した。

参　考　文　献

1) 梅村恭司，真田亜希子：文字列を k 回以上含む文書の計数アルゴリズム，自然言語処理， Vol.9, No.5, pp.180–186 (2002).
2) Kärkkäinen, J. and Sanders, P.: Simple linear work suffix array construction, *Proc. 30th International Conference on Automata, Languages and Programming*, Springer, pp.943–955 (2003).
3) Kasai, T., Lee, G., Arimura, H., Arikawa, S. and Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, London, UK, Springer-Verlag, pp. 181–192 (2001).
4) Yamamoto, M. and Church, K.W.: Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus, *Computational Linguistics*, Vol. 27, No. 1, pp. 1–30 (2001).
5) Hui, L. C.K.: Color Set Size Problem with Application to String Matching, *CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, London, UK, Springer-Verlag, pp.230–243 (1992).
6) Bender, M. A. and Farach-Colton, M.: The LCA Problem Revisited, *Latin American Theoretical INformatics*, pp.88–94 (2000).