2009－BIO－16 (11)
2009／3／6

未知の文字列集合を、それらを連結した文字列から推測する線形時間アルゴリズム

安田 知弘
株式会社日立製作所 中央研究所

$\mathcal{T}$ を未知の文字列の集合、$\mathcal{S}$ をそれらを連結した文字列の集合とし、$\mathcal{S}$ から $\mathcal{T}$ を推測する問題を考察する。関連研究から、最適化問題として定式化し解くことは計算量的に困難と予想される。本研究では、$\mathcal{S}$ の文字列を相互比較して共通部分文字列を抽出し、それらを互いの境界で分割することにより $\mathcal{T}$ を推測する線形時間アルゴリズムを開発した。本手法は $\mathcal{S}$ の大きさにかかわらず、$\mathcal{S}$ の文字列長の総和 L に対して $O(L)$ 時間で処理可能なスケーラブルな手法である。本アルゴリズムの有効性を、ランダムに生成された文字列とヒト cDNA 配列に対して適用する計算機実験で評価した。

A linear time algorithm that infers hidden strings from their concatenations

Tomohiro Yasuda
Central research laboratory, Hitachi, Ltd.

Let $\mathcal{T}$ be a set of hidden strings and $\mathcal{S}$ be a set of their concatenations. We address the problem of inferring $\mathcal{T}$ from $\mathcal{S}$. Any formalization of the problem as an optimization problem would be computationally hard. Here, we propose a new algorithm that infers $\mathcal{T}$ by finding common substrings in $\mathcal{S}$ and splitting them. This algorithm is scalable and can be completed in $O(L)$-time regardless of the cardinality of $\mathcal{S}$ where L is the sum of the lengths of all strings in $\mathcal{S}$. We evaluated the effectiveness of our method by computational experiments against randomly generated strings and human cDNA sequences.

1. Introduction

Let $\mathcal{T}$ be a set of hidden strings and $\mathcal{S}$ be a set of their concatenations. We consider the problem of inferring $\mathcal{T}$ from $\mathcal{S}$ when only $\mathcal{S}$ is given. The problem is motivated by the analysis of cDNA sequences. A gene might have more than one cDNA sequence by inserting or deleting alternative segments. This phenomenon is prevalent in many organisms[8]. Usually, alternatively spliced sequences are detected by aligning them with genomic sequences[4]. Nonetheless, methods that require as input only cDNA sequences would be useful because not all organisms have had their genomic sequences determined.

To clarify the problem, we show a small example.

Example1 Suppose that we are given $\mathcal{S} = \{S_0, S_1, S_2\}$, where
$S_0 = \texttt{ACGGTCTAGAATAGCAGGCTCGTCCTATGGCATTTT}$,
$S_1 = \texttt{CATCTGGTAGCAGGCTCGTCCTATCCAAGTAAAGGAC}$,
$S_2 = \texttt{CATCTGGTAAGTGGGCCGTCCTAT}$.
These are concatenations of strings in a set $\mathcal{T} = \{T_i | 0 \leq i < 8\}$, where
$T_0 = \texttt{ACGGTCTAGAAT}$, $T_1 = \texttt{AGCAGGCTC}$,
$T_2 = \texttt{GTCCTAT}$, $T_3 = \texttt{GGCATTTT}$,
$T_4 = \texttt{CATCTGGT}$, $T_5 = \texttt{CCAAGT}$,
$T_6 = \texttt{AAAGGAC}$, $T_7 = \texttt{AAGTGGGCC}$.
In fact, S_0, S_1 and S_2 can be rewritten as:
$S_0 = T_0T_1T_2T_3$, $S_1 = T_4T_1T_2T_5T_6$, $S_2 = T_4T_7T_2$.
We aim at inferring $\mathcal{T}$ from $\mathcal{S}$. □

If we formalize the problem as an optimization problem, it is difficult to give an efficient solution. Let $|\mathcal{S}|$ and $|\mathcal{T}|$ respectively be the cardinalities of $\mathcal{S}$ and $\mathcal{T}$. Néraud[9] considered the problem of determining, for a given set $\mathcal{S}$ of strings and an integer k, whether there exists a set $\mathcal{T}$ of strings such that $\mathcal{S} \subseteq \mathcal{T}^*$ and $|\mathcal{T}| \leq k$. Néraud proved that this problem is NP-complete even when $k = |\mathcal{S}| - 1$; that is, it is NP-complete to determine the existence of $\mathcal{T}$ that is smaller than $\mathcal{S}$. Even if only two strings are compared, it is NP-hard to find the smallest common collection of substrings that do not overlap each other and that cover the whole of the given two strings[2),7]. As to comparison of two strings, there are approximation algorithms, and also solutions for relaxed conditions. However, any extension for more than two strings would be computationally hard. Despite this difficulty, a method scalable to $|\mathcal{S}|$ is preferable for applications. For practical uses, a lot of MSA programs have been available[1]. Unfortunately, their purpose is not to decompose given strings into substrings of which given strings are concatenations, but to obtain alignments by finding similar regions.

Here, we propose a fast and scalable algorithm for inferring $\mathcal{T}$ from $\mathcal{S}$. Our approach is based not on optimization but on finding common substrings and splitting them.

2. Preliminaries

Let N be the cardinality of $\mathcal{S}$, and L be the sum of the lengths of all strings in $\mathcal{S}$. We denote by Σ a finite alphabet of which strings in $\mathcal{S}$ consist, by Σ^* the set of possibly empty strings, and by Σ^+ the set of non-empty strings. For a string $s \in \Sigma^*$, the length of s is denoted by $|s|$. When $s = s_1 s_2 s_3$ for some $s_1, s_2, s_3 \in \Sigma^*$, they are respectively called a *prefix*, a *substring*, and a *suffix* of s. Each of them is *proper* if it is not identical to s. When s is a substring of $S_i \in \mathcal{S}$ beginning at the j-th position of S_i, we say that s *occurs at* (i,j) or that (i,j) is an *occurrence* of s. Let $Occ(s)$ be the set of all occurrences of $s \in \Sigma^+$. For an integer k, we define $Occ(s)+k = \{(i,j+k) | (i,j) \in Occ(s)\}$. An empty set is denoted by $\emptyset$.

Let STree($\mathcal{S}$) be a generalized suffix tree[3] of all strings in $\mathcal{S}$. Each string in $\mathcal{S}$ is appended a distinct termination symbol at its right end[3]. A *path-label* of a node v in STree($\mathcal{S}$) is the concatenation of edge labels from the root to v. We denote by $p(v)$ the string obtained by removing any termination symbol from the path-label of v. Let $\mathcal{L}(i,j)$ be a leaf of STree($\mathcal{S}$) that represents the j-th suffix of $S_i \in \mathcal{S}$.

We capture common substrings in $\mathcal{S}$ as maximal common substrings defined below.

Definition1 (MCS) A string $m \in \Sigma^+$ is *a maximal common substring (MCS) for* $\mathcal{S}$ *and* l, if m is a substring of some $S_i \in \mathcal{S}$ and has the following properties:
(M1)$|m| \geq l$.
(M2)$Occ(m) \neq Occ(ms)$ for any $s \in \Sigma^+$.
(M3)$Occ(m) \neq Occ(sm) + |s|$ for any $s \in \Sigma^+$.
Let MCS($\mathcal{S},l$) be the set of all MCS's for $\mathcal{S}$ and l. □

MCS's are a natural extension of maximal repeats[3]. MCS's were also known to as *core blocks*[6].

Definition2 RightMCS($\mathcal{S},l$) is a set of non-empty strings, each of which is a substring of some $S_i \in \mathcal{S}$ and satisfies (M1) and (M2). □

RightMCS($\mathcal{S},l$) is a natural extension of strings considered in the DNA contamination problem[3].

3. Definition of DCS's

In **Fig. 1**, m_1(=CGTCCTAT) is an MCS shared by all of S_0, S_1, and S_2 of Example 1, while m_2(=TAGCAGGCT CGTCCTAT) is shared by only S_0 and S_1. This suggests that there is a string in $\mathcal{T}$ shared by all of S_0, S_1, and S_2, and on its left, there is another shared by only S_0 and S_1. To infer both of them, we should split m_2 at a boundary of m_1. We generalize this inference.

Definition3 (Boundary set) The *boundary set for* $\mathcal{S}$ *and* l is the set defined by $\mathcal{B}(\mathcal{S},l) = \mathcal{B}_L(\mathcal{S},l) \cup \mathcal{B}_R(\mathcal{S},l)$, where
$\mathcal{B}_L(\mathcal{S},l) = \bigcup_{m \in \mathrm{MCS}(\mathcal{S},l)} Occ(m)$,
$\mathcal{B}_R(\mathcal{S},l) = \bigcup_{m \in \mathrm{MCS}(\mathcal{S},l)} (Occ(m) + |m|)$. □

We infer strings in $\mathcal{T}$ as substrings of given strings that do not cross over any boundaries of MCS's.

Definition4 (DCS) A string $e \in \Sigma^+$ is *a disjoint common substring (DCS) for* $\mathcal{S}$ *and* l, if e is a substring of some $S_i \in \mathcal{S}$ and has the following properties:
(D1)$|e| \geq l$.
(D2)For any $(i,j) \in Occ(e)$ and any integer k such that $1 \leq k < |e|$, $(i, j+k) \notin \mathcal{B}(\mathcal{S},l)$.
(D3)$\mathcal{B}(\mathcal{S},l) \cap (Occ(e)+|e|) \neq \emptyset$.
(D4)$\mathcal{B}(\mathcal{S},l) \cap Occ(e) \neq \emptyset$.
Let DCS($\mathcal{S},l$) be the set of all DCS's for $\mathcal{S}$ and l. □

4. Algorithm that identifies DCS's

We introduce a class of strings that bridge DCS($\mathcal{S},l$) and STree($\mathcal{S}$).

Definition5 Let $H(\mathcal{S},l)$ be a set of strings such that any $h \in H(\mathcal{S},l)$ has the following properties:
(H1)$h \in$ RightMCS($\mathcal{S},l$).
(H2)For any proper prefix s of h, $s \notin$ RightMCS($\mathcal{S},l$).
(H3)$\mathcal{B}(\mathcal{S},l) \cap Occ(h) \neq \emptyset$. □

Then, the following lemma hold[11].

Lemma 1 For a non-empty substring e of some $S_i \in \mathcal{S}$, $e \in$ DCS($\mathcal{S},l$) if and only if e satisfies (D1)–(D3) and e is a prefix of some $h \in H(\mathcal{S},l)$. □

By Lemma 1, we can identify DCS($\mathcal{S},l$) by the following algorithm.

Algorithm: GET-DCS($\mathcal{S},l$)

Step 1:	Construct STree($\mathcal{S}$).
Step 2:	Identify RightMCS($\mathcal{S},l$) and MCS($\mathcal{S},l$).
Step 3:	Identify $\mathcal{B}(\mathcal{S},l)$.
Step 4:	Identify $H(\mathcal{S},l)$.
Step 5:	Identify DCS($\mathcal{S},l$).

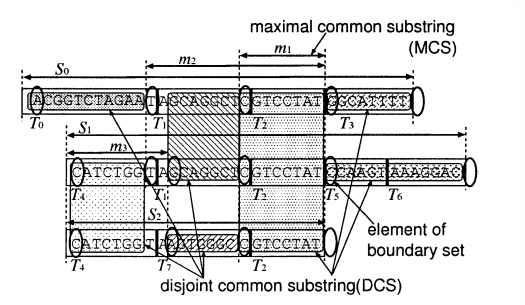

Fig. 1 MCS($\mathcal{S},l$), $\mathcal{B}(\mathcal{S},l)$, and DCS($\mathcal{S},l$) for Example 1. Bidirectional arrows indicate MCS's, while each circle at the j-th position of S_i indicates that $(i,j) \in \mathcal{B}(\mathcal{S},l)$. DCS's, indicated by hatched areas, do not cross over any boundaries of MCS's.

Steps 1 and 2 can be completed in $O(L)$-time[3].

Step 3: Identify $\mathcal{B}(\mathcal{S}, l)$

Clearly, $\mathcal{B}_L(\mathcal{S}, l)$ can be identified by a depth-first traversal on STree($\mathcal{S}$). Let us focus on $\mathcal{B}_R(\mathcal{S}, l)$. After initializing a set B to $\emptyset$, a depth-first traversal on STree($\mathcal{S}$) is conducted. For any $\mathcal{L}(i, j)$ encountered, we add $(i, j + l)$ to B if there is a node v such that $p(v) \in$ RightMCS$(\mathcal{S}, l)$ and $|p(v)| = l$ on the path from the root to $\mathcal{L}(i, j)$. When this process is completed, $B = \mathcal{B}_R(\mathcal{S}, l)$[11].

Step 4: Identify $H(\mathcal{S}, l)$

We identify $H(\mathcal{S}, l)$ by discarding any $p(v)$ from RightMCS$(\mathcal{S}, l)$ if $p(v)$ does not satisfy any one of (H2) or (H3), where v is a node in STree($\mathcal{S}$).

Step 5: Identify DCS$(\mathcal{S}, l)$

To avoid exhaustive search, we use variables $\lambda(h)$ for each $h \in H(\mathcal{S}, l)$, and the following pointers.

Definition6 $P[i, j]$ $(0 \leq i < N, 0 \leq j < |S_i|)$ is a pointer such that:

- $P[i, j] \rightarrow \lambda(h)$ if $(i, j) \in Occ(h)$ for $h \in H(\mathcal{S}, l)$, where $P[i, j] \rightarrow \lambda(h)$ means $P[i, j]$ points to $\lambda(h)$,
- $P[i, j]$ is a null pointer otherwise. □

We conduct the following procedures.

(1) Each $P[i, j]$ is initialized to a null pointer.

(2) Conduct a depth-first traversal on STree($\mathcal{S}$). For each $\mathcal{L}(i, j)$ encountered, $P[i, j]$ is set so that $P[i, j] \rightarrow \lambda(p(v))$ if there is a node v such that $p(v) \in H(\mathcal{S}, l)$ on the path from the root to $\mathcal{L}(i, j)$.

(3) For each $h \in H(\mathcal{S}, l)$, $\lambda(h)$ is initialized to $|h|$.

(4) Apply the algorithm PREFIX-DCS$(\mathcal{S}, l)$ in **Fig. 2**.

After these procedures are completed, the prefix of each $h \in H(\mathcal{S}, l)$ whose length is $\lambda(h)$ satisties (D2) and (D3). Therefore, it is a DCS if $\lambda(h) \geq l$. All Steps 1–5 can be completed in $O(L)$-time. Therefore,

Theorem 1 There is an algorithm that identifies DCS$(\mathcal{S}, l)$ in $O(L)$-time, where L is the sum of the lengths of all strings in $\mathcal{S}$.

5. Computational experiments

We evaluated GET-DCS$(\mathcal{S}, l)$ by computational experiments. We say $e \in$ DCS$(\mathcal{S}, l)$ is *consistent* with a string $t \in \mathcal{T}$ if and only if $|Occ(e)| = |Occ(t)|$ and

Algorithm: PREFIX-DCS$(\mathcal{S}, l)$

for $i := 0$ **to** $N - 1$ **begin**
  $x := 1$, $j := |S_i| - 1$
  repeat
    if $P[i, j] \rightarrow \lambda(h)$ **then** $\lambda(h) := \min\{x, \lambda(h)\}$
    $x := x + 1$
    if $(i, j) \in \mathcal{B}(\mathcal{S}, l)$ **then** $x := 1$
    $j := j - 1$
  until $j < 0$
end

Fig. 2 The algorithm PREFIX-DCS$(\mathcal{S}, l)$.

Table 1 Consistency of DCS's against $\mathcal{T}_1$.

n_{OK}	recall	precision
96,198	0.9895	0.9922

Table 2 Consistency of DCS's against $\mathcal{T}_2$.

	n_{OK}	recall	precision
(A)	21,803	0.7308	0.4208
(B)	23,798	0.7977	0.6278

the overlap of e and t occupies at least 90% of both e and t wherever e or t occurs. Let n_{OK} be the number of strings in DCS$(\mathcal{S}, l)$ consistent with some $t \in \mathcal{T}$. Below *recall* means $n_{OK}/|\mathcal{T}|$, while *precision* means $n_{OK}/|\text{DCS}(\mathcal{S}, l)|$. We used a Linux server with Opteron(tm) 252 processors.

5.1 Randomly generated strings

Let $\mathcal{T}_1$ be a set of 97,217 random strings consisting of `A`, `T`, `G`, and `C`, whose lengths were 50–240 bases and 145 bases on average. We applied GET-DCS$(\mathcal{S}, l)$ to 40,000 strings, each of which was a concatenation of nine strings of $\mathcal{T}_1$. The lengths of the 40,000 strings were 5.218×10^7 bases in total. We set l to 30. As shown in **Table 1**, the result was quite accurate.

To demonstrate the scalability of our implementation of GET-DCS$(\mathcal{S}, l)$, we measured the increase in computation time while the number of given strings was increased. As shown in **Fig. 3**, the computation time increased only linearly.

5.2 Transcriptome of *Homo sapiens*

Next, we tested our method against all cDNA sequences of *Homo sapiens* in the RefSeq database[10] of release 28. Although sequence differences were reconciled to finished genomic sequences in this database[10], they still contain plenty of complex features of real cDNA sequences. We removed consecutive `A`'s at the end of each sequence to exclude poly(A) tails. There were 25,199 sequences, whose lengths were 3,050 bases on average and 7.686×10^7 bases in total. We set l to 30. It took 826 seconds for GET-

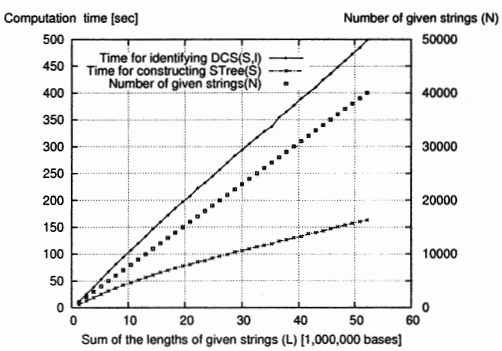

Fig. 3 Increase in computation time to identify DCS$(\mathcal{S}, l)$ while the number of given strings was increased.

Table 3 Results of our method and POA against the CREM gene. The exons 8, 11 and 14 had alternative ends, which were treated as independent exons. For example, exon 8 was divided into 8′ and 8″.

exon	length of exons	overlap with DCS's	overlap with POA substrings
1	321	320(0)	321(0)
2	108	108(0)	108(0)
3	98	97(0)	98(0)
4	124	122(0)	122(0)
5	265	263(0)	263(0)
6	110	109(0)	109(0)
7–8′	98+143	241(1)	241(1)
8″	571	570(0)	570(0)
9	189	187(0)	188(0)
10	88	86(0)	87(0)
11′	198	196(0)	198(0)
11″	43	40(0)	42(0)
12	36	33(0)	35(0)
13	157	154(0)	157(10)
14′	402	389(0)	392(10)
14″	1302	1292(0)	1292(0)

DCS($\mathcal{S}, l$) to identify DCS($\mathcal{S}, l$) from $\mathcal{S}$. For 23,777 sequences out of the 25,199 sequences, positions of exons and alternative ends of exons were available*1. Let $\mathcal{T}_2$ be a set of strings obtained by splitting the 23,777 sequences at alternatively spliced positions. As shown in row (A) of **Table 2**, the result was not satisfactory. Major causes of problems are sequence variations such as SNPs, repeated elements, and family genes sharing long identical regions irrelevant to alternative splicing. To partly circumvent these problems, we merged DCS's that always occurred adjacent in the same order. We also removed DCS's that occurred at least twice in a sequence. Then, we obtained an improved result shown in row (B) of Table 2.

5.3 Comparison with an MSA program POA

We compared the accuracy of our method with that of an MSA program POA[5]. Although we tried several MSA programs, all except POA suffered from weak similarities between different exons. As a test data set, we picked up 21 cDNA sequences of the cAMP-responsive element modulator (CREM) gene from the data set of the previous experiment. Their lengths were 41,535 bases in total. We set the mismatch parameter of POA to a huge negative value (-10^6). For GET-DCS($\mathcal{S}, l$), we set l to 20.

As shown in **Table 3**, results of our method and that of POA were quite consistent with exons of the gene, although exons 7 and 8′ were fused since they always occurred together. Both methods wrongly dropped 10 bases at 5'-end of exons 14′ and 14″ due to their identical 10-base prefixes. It took only 0.035 seconds for our method to obtain the result, while it took 97.797 seconds for POA.

*1 http://www.ncbi.nlm.nih.gov/mapview/

6. Conclusions

We proposed a linear time algorithm that infers a set $\mathcal{T}$ of hidden strings from a set $\mathcal{S}$ of their concatenations, and evaluated its effectiveness. If there is possibility that given strings are concatenations of an unknown set of strings, it is worth trying our method to identify such set of strings.

Acknowledgments The author thanks Prof. S. Miyano and Assoc. Prof. S. Imoto of the Institute of Medical Science, the University of Tokyo, for helpful advices and discussions. This work was partly supported by the New Energy and Industrial Technology Development Organization (NEDO), Japan.

References

1) Blackshields, G., Wallace, I., Larkin, M. and Higgins, D.: Analysis and comparison of benchmarks for multiple sequence alignment, *In Silico Biology*, Vol.6, No.4, pp.321–339 (2006).
2) Goldstein, A., Kolman, P. and Zheng, J.: Minimum common string partition problem: hardness and approximations, *Proc. 15th International Symp. Algorithms and Computation (ISAAC)*, pp.473–484 (2004).
3) Gusfield, D.: *Algorithms on strings, trees, and sequences*, Cambridge University Press, New York (1997).
4) Lander, E. et al.: Initial sequencing and analysis of the human genome, *Nature*, Vol.409, pp. 860–921 (2001).
5) Lee, C., Grasso, C. and Sharlow, M.: Multiple sequence alignment using partial order graphs, *Bioinformatics*, Vol.18, No.3, pp.452–464 (2002).
6) Leung, M., Blaisdell, B., Burge, C. and Karlin, S.: An efficient algorithm for identifying matches with errors in multiple long molecular sequences, *J. Molecular Biology*, Vol. 221, No. 4, pp. 1367–1378 (1991).
7) Lopresti, D. and Tomkins, A.: Block edit models for approximate string matching, *Theoretical Computer Science*, Vol. 181, No. 1, pp. 159–179 (1997).
8) Maniatis, T. and Tasic, B.: Alternative pre-mRNA splicing and proteome expansion in metazoans, *Nature*, Vol.418, pp.236–243 (2002).
9) Néraud, J.: Elementariness of a finite set of words is co-NP-complete, *Theoretical Informatics and Applications*, Vol. 24, No. 5, pp. 459–470 (1990).
10) Pruitt, K.D., Tatusova, T. and Maglott D.R.: The Reference Sequence (RefSeq) Project, *The NCBI Handbook*, NCBI, chapter18 (2002).
11) Yasuda, T.: A linear time algorithm that infers hidden strings from their concatenations, *IPSJ Trans. Bioinformatics*, Vol.1, pp.13–22 (2008).