準リアルタイム接尾辞木構築に関する応用について

Dominik Köppl　　Gregory Kucherov

概 要

Breslauer–Italiano の準リアルタイム接尾辞木構築アルゴリズム [JDS, 2013] を基盤として，いくつかの古典的な文字列問題に対する準リアルタイムアルゴリズムを開発する．具体的には，最長繰り返し接尾辞配列，Lempel–Ziv 77 分解，最小一意部分文字列のオンライン計算を含む．

キーワード： 文字列アルゴリズム，オンラインアルゴリズム，接尾辞木，リアルタイム計算，Lempel–Ziv 分解，最小一意部分文字列

1　はじめに

文字列は計算機科学における基本的なデータ型であり，文字列に関する多くの古典的な問題が広く研究されてきた．大規模な文字列を処理する際には，その処理が終了するまで待たずに，できるだけ早く結果を得ることが望ましい。そのために，我々は**オンライン**の方法で動作するアルゴリズムについて検討する．これらは文字列を左から右へ 1 文字ずつ処理し，これまでに読み込んだ文字列に関する情報を維持する．一方で，文字列に関するほとんどのオンラインアルゴリズムは，各文字あたりの**平均**時間保証のみ提供するため、各文字の処理に費やされる平均時間は入力サイズの関数で抑えられるが，一部の読み込まれた文字の処理にはより長い時間がかかる可能性がある．これは，各文字の処理時間を予測可能にすることが重要なリアルタイムアプリケーションでは問題となり得る．本論文では，各文字あたりの**最悪**時間保証を持つオンラインアルゴリズムに焦点を当てる．Lempel–Ziv 78 [19] 分解 [8]，回文認識，パターン照合 [6] に対するリアルタイムアルゴリズムが文献で提示されている一方で，文字列に関する多くの古典的な問題に対しては，各文字あたりの効率的な最悪時間を持つオンラインアルゴリズムは検討されていない．

文字列に関する問題の大部分は，これまでに読み込んだ文字列の全文索引データ構造を維持できる場合にのみ効率的に解くことができる．そのような基本的な索引データ構造の一つが接尾辞木である [16]．接尾辞木の最初のオンライン構築は Ukkonen によるものである [15]．これは定数サイズのアルファベットに対して時間 $O(n)$ でオンラインに接尾辞木を構築する．しかし

表 1: 文字列長 n の文字列に対して様々な文字列問題をオンラインで解くアルゴリズムの 1 文字あたりの最悪時間計算量．既知の結果は本研究以前で最も良いものである．MUS 計算については，より良い時間計算量を持つ乱択アルゴリズムも存在する．ビット単位のワードサイズを w で表す．SuffixUpdate を解くための時間計算量 t_{SU} の可能な値については，表 2 を参照されたい．

問題	既知の結果	決定時間	期待時間	節
LRS	$O(\lg^3 n)$ [12]	$O(t_{\mathrm{SU}})$	$O(t_{\mathrm{SU}})$	3
LZ77	$O(\lg^3 n)$ [12]	$O(t_{\mathrm{SU}})$	$O(t_{\mathrm{SU}})$	4
MUS	$O(n)$ [11]	$O(t_{\mathrm{SU}} + \log_w n)$	$O(t_{\mathrm{SU}})$	5

Ukkonen は，各文字の処理に対して $O(n)$ という自明な上限以外に最悪時間保証を提供しない．実際，この上限は特定の入力に対して厳密である：文字列 SSc について考える場合，$|S| \in \Theta(n)$．Ukkonen のアルゴリズムは接尾辞木を更新する際に，接尾辞木のアクティブポイントが文字深さ (string depth) $|S|$ を持つことになるが，もし c が新しい文字であれば，S のすべての接尾辞に対して分岐を追加する必要があり，最後の文字の処理だけで $\Theta(n)$ 個の頂点を生成することになる．（入力文字列の最後に一意の区切り文字 \$ を追加することが通常であるため，最後の文字に対して $\Theta(n)$ を費やすことは実際には一般的な現象かもしれない.）最悪時間保証を達成するために，この一連の研究は，これまでに読み込んだ文字列の接尾辞木（または類似のデータ構造）を維持するという考えに基づいている．このような索引データ構造の中で最も基本的なものの 1 つが接尾辞木である [16]．この一連の研究の出発点は Weiner のアルゴリズムであり [16]，定数サイズのアルファベットに対して線形時間で接尾辞木を構築する最初のアルゴリズムである．Weiner のアルゴリズムは元々，各文字の処理に対して最悪時間保証を提供しない．しかし，次節では，それを可能にしうる修正に関する研究をまとめる．これらの修正は，各文字あたり多項対数時間という時間計算量をもたらす．これはしばしば**準リアルタイム**と呼ばれる [4].

本研究の貢献 準リアルタイム接尾辞木構築アルゴリズムをブラックボックスとして利用することで，維持された接尾辞木に基づいて様々な古典的な文字列問題を準リアルタイムオンラインで解くことができる．まず，第 3 節で**最長繰り返し接尾辞配列**（LRS）の計算を行う．これは第 4 節での Lempel–Ziv 77（LZ77） [18] 分解の計算の基礎となる．第 5 節では，これまでに読み込んだ文字列の**最小一意部分文字列**（MUS）の集合について扱う．我々の結果と先行研究で得られてきた結果の概要については，表 1 を参照されたい．

2　準備知識

我々は，入力文字列の長さを n とした場合，ワードサイズ $w \geq \log n$ ビットを持つワード RAM を用いる．Σ をサイズ $\sigma = |\Sigma| = n^{O(1)}$ の整数アルファベットとする．Σ^* の要素は**文字列**と呼ばれる．文字列 $S \in \Sigma^*$ が与えられた場合，その長さを $|S|$ で表し，$i \in [1..|S|]$ に対してその i 番目の記号を $S[i]$ で表す．さらに，$S[i..j] = S[i] \cdots S[j]$ と表す．$\overleftarrow{S}$ は S の**反転**文字列を表し，すなわち $\overleftarrow{S} = S[|S|]S[|S|-1] \cdots S[1]$ である．文字列 S の**接尾辞木** $\mathsf{ST}(S)$ は，S のすべての接尾辞を表す接尾辞木である．

Weiner のアルゴリズムは，文字列 $T[1..n]$ の接尾辞木 $\mathsf{ST}(T)$ を最短の接尾辞 $T[n..]$ から最長の接尾辞 $T[1..]$ まで処理することで計算する．この構築は，Weiner リンクと呼ばれる追加の辺で接尾辞木頂点を拡張する：頂点 u から文字 c でラベル付けされた *Weiner* **リンク**（$\mathsf{W}_c(u)$ で表される）は，文字列 $c \cdot \mathsf{label}(u)$ の位置を指す．ここで，$\mathsf{label}(u)$ は根から頂点 u までの経路によって綴られる文字列である．$c \cdot \mathsf{label}(u)$ が（現在の）接尾辞木における位置である場合にのみ，$\mathsf{W}_c(u)$ は定義される．$c \cdot \mathsf{label}(u)$ の位置が頂点である場合，Weiner リンクは**ハード**と呼ばれ，そうでない場合は**ソフト**と呼ばれる．Breslauer–Italiano による Weiner のアルゴリズムの修正では，ソフト Weiner リンクは $c \cdot \mathsf{label}(u)$ の位置の最も近い子孫頂点への辺として表される．

Weiner のアルゴリズムは工程ごとに動作する．i 番目の工程では，以下の問題を解く．

SuffixUpdate
入力: 接尾辞木 $\mathsf{ST}(T[i+1..])$ と文字 $T[i]$.
出力: $\mathsf{ST}(T[i..])$.

SuffixUpdate を解決するために，アルゴリズムは新しい葉を接続すべき挿入点を特定する．挿入点は，$T[i..]$ の最長の接頭辞であり，$T[i+1..]$ に部分文字列として既に存在するものであり，既存の頂点または既存の辺上にある可能性がある．挿入点の計算は Weiner のアルゴリズムの重要な操作であり，以下のように定式化される．

InsertionPoint
入力: 接尾辞木 $\mathsf{ST}(T[i+1..])$ と文字 $T[i]$.
出力: $T[i..]$ のうち $T[i+1..]$ に出現する最長接頭辞の位置.

表 2 には，SuffixUpdate と InsertionPoint の既知の時間計算量の概要が示されている．

表 2: SuffixUpdate を解くための時間計算量 t_{SU} の一覧

時間計算量 t_{SU}	参考文献
$O(\lg n)$	[1]
$O(\sigma \log\log n)$	[4]
$O(\log\log n + \log\log \sigma)$ 期待	[9]
$O(\log\log n + \frac{\log^2 \log \sigma}{\log\log\log \sigma})$	[5]
$O(\log\log n)$ ただし $\sigma = O(\log^{1/4} n)$	[10]

3 最長繰り返し接尾辞配列

文字列 $T[1..n]$ の**最長繰り返し接尾辞配列**（LRS）は，配列 $\mathsf{LRS}[1..n]$ であり，各位置 $i \in [1..n]$ に対して，$\mathsf{LRS}[i]$ は $T[1..i]$ の中で少なくとも 2 回出現する最長の接尾辞の長さである．

Okanohara と Sadakane [12] は，この配列のオンライン計算を最初に研究し，各文字あたり $O(\log^3 n)$ 時間を得た．Prezza と Rosone [13] は，各文字あたり $O(n \lg n)$ の遅延を許容することで，各文字あたり平均 $O(\log^2 n)$ 時間を得た．我々はここで，ST を維持することの副産物として，LRS をオンラインでの計算可能性を示す．そのために，以下の問題を解く．

LongestRepeatingPrefix
入力: 接尾辞木 $\mathsf{ST}(T[i+1..])$ と文字 $T[i]$.
出力: $T[i..]$ のうち最長の繰り返し接頭辞.

我々は，Amir ら [2] の方法論を再検討することで，LongestRepeating-Prefix を解く．彼らはこの問題を InsertionPoint に帰着させる．なぜなら，挿入点の文字ラベルは $T[i..]$ のうち $T[i+1..]$ に既に出現する最長の接頭辞だからである．LRS を計算するために，我々は反転文字列 $\overleftarrow{T}$ 上で ST を計算する．文字 $T[i]$ を処理するとき，文字 $T[i]$ を用いて $\mathsf{ST}(\overleftarrow{T[i+1..]})$ 上で LongestRepeatingPrefix に回答する．その答えが $\mathsf{LRS}[i]$ である．

定理 1. 文字列 $T[1..n]$ の最長繰り返し接尾辞配列 LRS をオンラインで各文字あたり $O(t_{\mathrm{SU}} + 1)$ 最悪時間で計算できる．

4 Lempel–Ziv 77 (LZ77)

文字列 T の LZ77 分解は以下のように定義される．文字列 T の分解 $T = F_1 \cdots F_z$ が与えられたとき，各要素 F_x（$x \in [1..z]$）が，文字列 $F_1 \cdots F_x$ に少なくとも 2 回出現する最長の接頭辞であるか，または文字の最左出現である場合，この分解は T の *LZ77* **分解**である．

オンライン設定に対しては，Gusfield の教科書 [7, APL 16] は，各文字あたり $O(\log \sigma)$ の平均時間で動作するアルゴリズムを提示する．その主なアイデアは，Ukkonen のアルゴリズム [15] を用いてこれまでに読み込んだテキストの接尾辞木を維持し，Ukkonen のアルゴリズムによって維持されるアクティブポイントを用いて LPF を計算することである．我々は LRS から LZ77 分解を導出できるため，Okanohara と Sadakane [12] のアルゴリズムはオンライン LZ77 分解を副産物として計算するために使用できる．彼らは各文字あたり $O(\log^3 n)$ 時間を得た．その後の解法は，空間を $O(n \log \sigma)$ ビットに保ちながら時間計算量の改善に焦点を当てたが，各文字あたりの最悪時間を犠牲にした．この研究の流れでは，Starikovskaya [14] は Ukkonen の接尾辞木構築アルゴリズムに戻ることで，各文字あたり $O(\log^2 n)$ の平均時間に時間計算量を改善した．最後に，Yamamoto ら [17] は有向非巡回単語グラフ [3] を構築することで，各文字あたり $O(\log n)$ の平均時間を得た．

以下では，各文字あたりの最悪時間に注目し，$O(n)$ ワードの空間を許容することでこの研究の流れから路線を変更する．我々は LRS を計算する手段を持つことで，LongestRepeatingPrefix と同じ時間計算量で定数の超過で LZ77 分解を計算できることを示す．

Weiner の接尾辞木構築アルゴリズムを用いてオンラインで LZ77 分解を計算するために，これまでに読み込んだテキスト $T[1..j]$ の最長繰り返し接尾辞 S_j を LongestRepeatingPrefix を用いて維持する．現在の要素の開始位置を i とし，$\mathsf{ST}(\overleftarrow{T[1..j]})$ で $T[1..j]$ の反転を索引付けしているとする．もし $i = j$ かつ S_j が空であれば，現在の要素は文字 $T[j]$ の最初の出現である．S_j が $j - i + 1$ より短い場合，$T[1..j]$ の接尾辞は存在しないため，現在の要素は $j - 1$ で終了することが分かる．そうでない場合，現在の要素の終了位置が見つかるまで j を 1 ずつ増加させることで以下の処理を繰り返す．

定理 2. 文字列 $T[1..n]$ の LZ77 分解をオンラインで各文字あたり $O(t_{\mathrm{SU}} + 1)$ 最悪時間で計算できる．

5 最小一意部分文字列

文字列 $T[1..n]$ の部分文字列 S に対して，$\#occ_T(S)$ を T における S の出現回数とする．T における部分文字列 S が $\#occ_T(S) = 1$ であるとき，S は T において**一意**であると呼ばれ，$\#occ_T(S) \geq 2$ であるときは**繰り返し**であると呼ばれる．T における一意の部分文字列 S が，任意の真部分文字列が T において繰り返しであるとき，S は T の**最小一意部分文字列**（MUS）と呼ばれる．一意の部分文字列 S は，T において正確に 1 回出現するため，$S = T[\ell..r]$ を満たす一意の区間 $[\ell..r] \subseteq [1..n]$ で識別できる．我々は，T の MUS に対応する区間の集合を $\mathsf{MUS}(T) = \{[s..t] \mid T[s..t] \text{ は } T \text{ の MUS}\}$ で表す．MUS の定義から，$[s..t] \in \mathsf{MUS}(T)$ であることと次の条件 (a)–(c) が同値である．(a)

$T[s..t]$ は T において一意である，(b) $T[s+1..t]$ は T において繰り返しである，および (c) $T[s..t-1]$ は T において繰り返しである．

Mieno ら [11] はスライディングウィンドウにおける MUS の計算アルゴリズムを提示し，特に文字 $T[j+1]$ が $T[1..j]$ に追加されたときに $\mathsf{MUS}(T)$ をどのように更新できるかを研究した．$T[1..j]$ の最長繰り返し接尾辞を lrs_j とし，$T[1..j]$ において正確に 2 回出現する最短接尾辞（すなわち，正確に 1 つの以前の出現を持つもの）を sds_j とする．一方で，lrs_j は空文字列を許容できるため，常に存在する．他方で，sds_j は $T[1..j]$ の接尾辞が正確に 2 回出現しない場合，すなわちすべての $i \in [1..j]$ について $\#occ_{T[1..j]}(T[i..j]) \neq 2$ である場合には存在しない可能性がある．

lrs_j の長さと sds_j の以前の出現の終了位置（もし sds_j が存在するならば）は，文字 $T[j+1]$ が $T[1..j]$ に追加されたときに $\mathsf{MUS}(T[1..j])$ に対するすべての可能な変更を特定するために知る必要がある 2 つの指標である．これらの変更は，MUS の 1 つの削除と最大 3 つの新しい MUS の追加で構成される可能性がある [11]．第 3 節で lrs_j を維持するための以前の結果に基づき，我々は sds_j を追加で維持することで，オンラインで $\mathsf{MUS}(T[1..j])$ を維持できることを示す．これは $\mathsf{ST}(\overleftarrow{T[1..j]})$ における lrs_j の位置を持つことで定数時間で取得できる．

定理 3. 文字列 $T[1..n]$ の最小一意部分文字列集合 $\mathsf{MUS}(T[1..n])$ をオンラインで各文字あたり $O(t_{\mathrm{SU}} + t_{\mathrm{CDRA}})$ 最悪時間で計算できる．ここで，t_{CDRA} は動的配列を維持する動的データ構造の各操作あたりの時間である．

6 おわりに

本論では，準リアルタイム接尾辞木構築アルゴリズムをブラックボックスとして利用することで，様々な古典的な文字列問題を準リアルタイムオンラインで解く方法を示した．

謝辞

本研究は JSPS 科研費 25K21150 と 23H04378 の支援を受けたものである．

参考文献

[1] Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *Proc. SPIRE*, volume 3772 of *LNCS*, pages 67–78, 2005.

[2] Amihood Amir, Gad M. Landau, and Esko Ukkonen. Online timestamped text indexing. *Inf. Process. Lett.*, 82(5):253–259, 2002.

[3] Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.

[4] Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

[5] Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. CPM*, volume 9133 of *LNCS*, pages 160–171, 2015.

[6] Zvi Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proc. STOC*, pages 161–173, 1976.

[7] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[8] Alan Hartman and Michael Rodeh. Optimal parsing of strings. In *Combinatorial Algorithms on Words*, pages 155–167, Berlin, Heidelberg, 1985.

[9] Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. FOCS*, pages 283–292, 2012.

[10] Gregory Kucherov and Yakov Nekrich. Full-fledged real-time indexing for constant size alphabets. *Algorithmica*, 79(2):387–400, 2017.

[11] Takuya Mieno, Yuki Kuhara, Tooru Akagi, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Minimal unique substrings and minimal absent words in a sliding window. In *Proc. SOFSEM*, volume 12011 of *LNCS*, pages 148–160, 2020.

[12] Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Proc. ESA*, volume 5193 of *LNCS*, pages 696–707, 2008.

[13] Nicola Prezza and Giovanna Rosone. Faster online computation of the succinct longest previous factor array. In *Proc. CiE*, volume 12098 of *LNCS*, pages 339–352, 2020.

[14] Tatiana Starikovskaya. Computing Lempel–Ziv factorization online. In *Proc. MFCS*, volume 7464 of *LNCS*, pages 789–799, 2012.

[15] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[16] Peter Weiner. Linear pattern matching algorithms. In *Proc. SWAT*, pages 1–11, 1973.

[17] Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line Lempel–Ziv factorization. In *Proc. STACS*, volume 25 of *LIPIcs*, pages 675–686, 2014.

[18] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977.

[19] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.