Vol.2015-AL-153 No.7
2015/6/12

制約付き Re-Pair アルゴリズムと等価な半オンライン型置換アルゴリズム

正木 拓也[1]　喜田 拓也[1]

概要：
Re-Pair アルゴリズムは長さ n の入力テキストを $O(n)$ 時間で等価な文法に変換する．しかし，その動作はオフライン的であるのでテキスト全体を一度にメモリ上に読み込む必要がある．本稿では，Re-Pair によって生成される文法に制約を付け，それと同じテキスト置換を半オンライン的に実行するアルゴリズムを提案した．提案アルゴリズムは $O(n \log \hat{g})$ 時間で動作し，最悪時 $O(g)$ 領域を使用する．ここで，g は生成規則の数，$\hat{g}$ は生成規則の構文木の高さの最大である．また，100MB の DNA データに対して Re-Pair と提案手法の比較実験を行い，計算時間はほぼ同等で使用メモリ領域を 1/70 と大幅に削減できたことを確認した．

1. はじめに

Larsson と Moffat ら [4] によって提案された Re-Pair アルゴリズムは，文法変換に基づく圧縮アルゴリズムである．文法変換に基づく圧縮アルゴリズムとは，入力テキストから，それのみを導出する形式文法を生成し，生成された文法を符号化することによって圧縮を行うデータ圧縮法である．このような圧縮方式は，文法圧縮とも呼ばれる．

Re-Pair アルゴリズム (Re-Pair) は，入力テキスト中に最頻する文字ペア (bigram) を優先的に選択しながら，選択したペアを文法の非終端文字へと再帰的に置き換えるというシンプルなヒューリスティックを採用している．圧縮の過程で得られる文法は，既存の文法圧縮の中では比較的コンパクトなものになる傾向にあり，結果として非常に高い圧縮率を達成する．一般的に文法圧縮は，繰り返し部分の多いテキスト (highly repetitive text) に対しては有利であると言われる [7]．Re-Pair は，それに加え，通常の自然言語テキスト等に対しても良好な圧縮率を達成することが知られている [14]．

Re-Pair の問題点は，そのメモリ使用量の多さである．Re-Pair の時間・領域計算量は共に入力テキスト長 n に対して $O(n)$ であるものの，再帰的な置換処理の時間計算量を $O(n)$ で抑えるために複雑なデータ構造が必要であり，実装の仕方にもよるが，元の入力テキストの数十倍ものメモリを使用する．しかも，その処理はオフライン的であり，圧縮時には入力テキスト全体をメモリ上に読み込む必要がある．このことから，Re-Pair アルゴリズムはギガバイトを超えるテキストに対して適用することが，現状では困難である．

こうした傾向はオフラインな文法圧縮に共通したものであり，これまでにいくつかの改善手法が提案されている．Wan と Moffat ら [13] は，入力テキストをブロックに区切って Re-Pair を適用し，ブロック毎に生成された文法規則の集合（辞書）を再帰的に統合する手法 (Re-Merge) を提案している．これにより，Re-Pair の高い圧縮率を保ったままメモリ使用量が抑えられることを実験的に示した．その反面，Re-Merge は圧縮速度に大きな犠牲を払っている．同様にブロック毎に Re-Pair を行う改善手法として，Sekine ら [10, 11, 15] は，ブロック毎の辞書の一部を共有することによって圧縮率と圧縮速度のバランスを取る手法（Blocked-Repair-VF）を提案している．ただし，大規模なテキストに対して優れた圧縮率を得るためには，事前に適切なパラメータの設定を必要とするうえに，ブロック長を数十〜百メガバイト以上とする必要がある．

オンラインな文法圧縮として，Maruyama ら [7] は，FOLCA と呼ばれる手法を提案している．FOLCA は，Sakamoto ら [9] が提案した LCA アルゴリズムを基にしており，入力テキストの文字集合にのみ依存したヒューリスティックを用いて文法変換を行う．それにより，データ圧縮時に必要とする辞書の量を大幅に抑え，省メモリな処理を可能としている．他の文法圧縮同様に，繰り返し部分の

[1] 北海道大学大学院情報科学研究科
Hokkaido University, Graduate School of Information Science and Technology, {tmasaki, kida}@ist.hokudai.ac.jp

多いテキストに対しては強力に圧縮を行うが，その一方で自然言語テキスト等に対する圧縮率は優秀であるとは言い難い [16]．その理由は，生成される文法が Re-Pair ほどコンパクトにならないためである．

本稿では, Re-Pair に基づいた半オンライン (semi-online) な文法変換アルゴリズムを提案する．ここで言う半オンラインな処理とは，入力テキスト全体をメモリに置かずに二次記憶装置から文字を逐次に読みだしつつ圧縮を行うが，ある大きさのバッファを用い，結果の出力に遅延を許す処理を指す．当然ながら，バッファのサイズは入力テキストと比較して十分小さいことが期待される．提案アルゴリズムは，Re-Pair の文法規則の生成方法にある制約を課すことで，元の置き換え方と同じ文法変換を半オンラインな処理で実行する．これにより，入力テキスト全体をメモリに読み込むことなく，省メモリで文法変換を行うことができる．実際に実証実験を行い，この制約付き Re-Pair の圧縮率および圧縮速度が元の Re-Pair とほぼ同等であり，使用メモリ量を劇的に抑えることができることを示す．

ただし，提案アルゴリズムでは，辞書となる文法規則の集合は最初に与えられるものと仮定している．実用上，入力テキストから先に辞書だけを構築する方策は突飛なものではない．全体に渡ってテキストの性質があまり変化しない，あるいは十分なサイズの辞書を準備できるという状況であれば，圧縮対象のテキストとは別に，前もって辞書を準備するほうが適切な場合がある．例えば，入力テキストを 2 回走査することが許される場合には，一旦先にテキスト全体に対する辞書を構築するほうが，適応的に辞書を構築するよりも圧縮率を良くすることができる．先に述べた Sekine らの手法においても，ブロック間で共有するための辞書はテキストの置換処理に先んじて構築される．

2. 関連研究

Kieffer と Yang [3] は，2000 年に文法圧縮の枠組みを提案した．彼らは，文脈自由文法から導出される文字列が唯一となるような制約の付けかたを明確にした．Re-Pair [4] や SEQUITUR [8] は，Kieffer と Yang らの仕事よりも先んじて提案された圧縮法であるが，彼らの枠組みに含まれる．Bisection [2] は，より制約の強い straight line program に属する文法圧縮アルゴリズムである．一方で，Maruyama ら [6] は，文脈依存文法に基づく巧妙な文法圧縮を提案している．

Shibata ら [12] は，Byte Pair Encoding [1] と呼ばれる圧縮法を発掘し，それによって圧縮されたテキストに対して文字列照合処理が高速に行えることを示している．Byte Pair Encoding は，実質，Re-Pair と同一のアルゴリズムである．ただし，辞書のサイズ（文法規則の数）を 256 に制限し，8 ビット固定長の符号語を採用している．それにより，圧縮後のファイルがバイト単位で簡便に取り扱えるという利点がある．

先にも述べたが，Wan と Moffat [13] によって提案された Re-Merge は，大規模テキストに対して Re-Pair を適用するための拡張手法の一つである．彼らの実験結果によると，Re-Merge による圧縮率はきわめて良好であり，英語の自然言語テキストデータである `WSJ508` に対して，およそ 20% の圧縮率を達成している．実験に用いられた `WSJ508` とは，508MB の SGML によるマークアップがなされた新聞記事であり，1987 年から 1992 年までの記事が含まれている．一方で，圧縮時間に関しては芳しくなく，彼らの実験環境（2.8GHz の Intel Xeon，2GB メモリ，Debian GNU/Linux）において 4400 秒かかっている．

3. 準備

3.1 記法の定義

記号の有限集合を**アルファベット**と呼ぶ．アルファベット S の要素を 0 個以上並べたものを**文字列**と呼ぶ．すなわち，S 上の文字列とは S^* の要素である．文字列 $x \in S^*$ の長さ (並んだ記号の個数) を $|x|$ と書く．長さが 0 の文字列を**空語**といい，ε で表す．文字列 x の i 番目の要素を $x[i]$ と書く．また，i 番目から j 番目の連続する文字の並びを**部分文字列**といい，$x[i,j]$ と書く．便宜的に，$i > j$ のときは $x[i,j] = \varepsilon$ とする．長さが m の部分文字列は m **グラム**とも呼ばれる．特に，長さが 2 の場合を**バイグラム**と呼ぶ．

文脈自由文法 (CFG) $G = (\Sigma, V, \sigma, R)$ を考える．ここで，Σ は**終端アルファベット**，V は**非終端アルファベット** ($V \cap \Sigma = \emptyset$)，$\sigma \in V$ は**開始記号**である．終端アルファベット，非終端アルファベットの要素をそれぞれ，**終端記号**，**非終端記号**と呼ぶ．R は V から $(V \cup \Sigma)^*$ への関係であり，その要素は**生成規則**と呼ばれる．すなわち，ある $\alpha \in V$ と $\beta \in (V \cup \Sigma)^*$ に対して，$(\alpha, \beta) \in R$ である．またこのとき，$\alpha \to \beta$ と表現する．

CFG G は，開始記号 σ から R 中の生成規則を繰り返し適用することで文字列を導出する．非終端記号 $X \in V$ について，その導出過程を表す木構造を X の**構文木**と呼ぶ．

CFG G が次のような形の生成規則のみからなる場合，**チョムスキー標準形**という．

$$
\begin{aligned}
&X \to a && (a \in \Sigma), \text{または} \\
&X \to X_l X_r && (X_l, X_r \in V), \text{または} \\
&\sigma \to \varepsilon.
\end{aligned}
$$

任意の文脈自由文法は，それと等価なチョムスキー標準形の文法に書き換えることができる．G がチョムスキー標準形の場合，一つの非終端記号から導出される非終端記号は常に二つである．したがって，任意の非終端記号 X についてその構文木は二分木となる．

3.2 Re-Pair アルゴリズム

与えられたテキスト T に対して，Re-Pair は CFG $G = (\Sigma, V, \sigma, R)$ を一つ生成する．ここで，終端アルファベット $\Sigma = \{a_0, a_1, \cdots, a_{|\Sigma|-1}\}$ は，T を構成するアルファベットと同一である．非終端アルファベットを $V = \{\alpha_0, \alpha_1, \cdots, \alpha_{|V|-1}\}$ とすると，Re-Pair によって生成される CFG G は次のような生成規則からなる．

$$\sigma \to \alpha_{i_0}\alpha_{i_1}\cdots\alpha_{i_{m-1}} \quad (\forall i_k \in \{0, \cdots, |\Sigma| + |V| - 2\}),$$

$$\alpha_i \to \begin{cases} a_i & (0 \le i < |\Sigma| \text{ の場合}), \\ \alpha_j\alpha_k \quad (0 \le j, k < i) & (i \ge |\Sigma| \text{ の場合}). \end{cases}$$

すなわち，この G は，開始記号の生成規則以外の部分はチョムスキー標準形になっている．

Re-Pair で生成される CFG G について，非終端記号 α_i $(i \ge |\Sigma|)$ の生成規則の右辺 α_j, α_k はすべて異なる組み合わせである．任意の $\alpha_i \to \alpha_j\alpha_k (i \ge |\Sigma|)$ に対し，$L(\alpha_i) = \alpha_j$，$R(\alpha_i) = \alpha_k$ と定義する．G の任意の非終端記号 X から導出される文字列は唯一である．また，開始記号 σ からは唯一 T のみが導出される．X の構文木は形が唯一に決まるので，以降では特に混乱がない限り，非終端記号とその構文木を同じ記号で表す．X の構文木の高さを $h(X)$ と書き，同時にこれを X **の高さ**と呼ぶ．形式的には，整数 $0 \le i < |\Sigma|$ に対しては $h(\alpha_i) = 0$ とし，$i \ge |\Sigma|$ に対しては $h(\alpha_i) = \max\{h(L(\alpha_i)), h(R(\alpha_i))\} + 1$ と定義する．

次に，Re-Pair による，テキスト T から CFG G への変換処理について述べる．まず，T のアルファベット $\Sigma = \{a_0, a_1, \ldots, a_{|\Sigma|-1}\}$ を G の終端アルファベットとする．そして，生成規則 $\alpha_i = a_i$（$0 \le i < |\Sigma|$）を R に追加する．次に Re-Pair は，T 中に出現するバイグラムのうち最頻出のもの $\alpha_l\alpha_r$ を一つ選び，そのすべての出現をテキストの先頭から順に新しい記号 $X \notin \Sigma \cup V$ に置き換える．そして，X を非終端アルファベット V に，$X \to \alpha_l\alpha_r$ を生成規則として R に追加する．このとき新たに V に追加される非終端記号 X は，追加された順番で番号付されているものとする．したがって，$|V| = k$ のとき，$X = \alpha_k$ である．この置換手続きを，T 上のすべてのバイグラムの頻度が 1 になるまで繰り返す．上記の置換手続き後のテキスト T' を開始記号 σ から導出するように，生成規則 $\sigma \to T'$ を R に追加する．以上により，T を唯一に導出する CFG G を得る．

最終的には，G に適当な符号化を行い圧縮データを得る．生成規則 $\sigma \to T'$ が圧縮後のテキスト系列に対応し，それ以外の生成規則の集合がデータ圧縮のための**辞書**に対応していると見ることができる．よって，実際の符号化では，先に辞書に対して適切な符号化を行い，それを基に T' を符号化する方式が取られる．

Larsson と Moffat ら [5] によって，Re-Pair は，入力テキスト長 n に対し $O(n)$ 時間で CFG G を構築できることが示されている．その計算時間を達成するためには，入力テキストの全体を一旦オフラインで処理し，同一のバイグラムどうしを前後で連結させた双方向連結リストに変換する必要がある．さらに，任意のバイグラムの最初の出現位置へ $O(1)$ 時間でアクセスするためのハッシュテーブルと，バイグラムの頻度を管理するための優先度付きキューを必要とする．

4. 提案手法

本節では，我々が提案する *LT-RePair* **アルゴリズム**と，それと同じテキスト置換をオンライン的に実行する *SemiOnlineReplace* **アルゴリズム**について述べる．オフラインでの処理を前提とした Re-Pair で生成された辞書では，例え先にその辞書が与えられていたとしても同じ置き換え方をオンラインで実行することは困難である．我々の基本アイデアは，RePair で選択するバイグラムに，できるだけ左側が優先的に置換されていくような制約を付けることである．その制約を付けることで，バッファを用いたオンライン的な置換アルゴリズムを得ることができる．ちなみに，我々の予備的な実験において，この制約を課すことで圧縮率が犠牲になることはほとんどないことを確認している．

4.1 LT-RePair

提案する文法変換アルゴリズムは，基本的には Re-Pair と同じ手順で最頻バイグラム置換を繰り返す．ただし，選択されるバイグラム XY は，$h(X) \ge h(Y)$ となるもののみを許す．したがって，提案アルゴリズムによって生成される CFG G' は次のような生成規則からなる．

$$\sigma \to \alpha_{i_0}\alpha_{i_1}\cdots\alpha_{i_{m-1}} \quad (\forall i_k \in \{0, \cdots, |\Sigma| + |V| - 2\}),$$

$$\alpha_i \to \begin{cases} a_i & (0 \le i < |\Sigma| \text{ の場合}), \\ \alpha_j\alpha_k \quad (0 \le j, k < i \text{ かつ } h(\alpha_j) \ge h(\alpha_k)) & \\ & (i \ge |\Sigma| \text{ の場合}). \end{cases}$$

言い換えると，生成される G' の任意の非終端記号 X の構文木は，左の部分木が常に右の部分木よりも高さが等しいか大きい．よって，この文法変換を Left Tall Re-Pair（LT-RePair）と名付ける．

いま，この LT-RePair で作成された辞書 D の通りに，圧縮されていないテキスト $T = T[0, |T| - 1]$ を置き換えていくことを考える．議論の簡便のため，この入力テキスト T の各文字はそれと対応する非終端記号 α_i（$0 \le i \le |\Sigma|$）に置き換えられているものとする．また，位置 i のバイグラム $T[i, i+1]$ が置換された後のテキスト T' は，$i+1$ 以降の文字列が前方に詰められて索引付けされると考える．つまり，置換前に $T[i+2]$ だった文字は，置換後には $T'[i+1]$ としてアクセスされる．ここで，テキスト上の i 番目の位置

にあるバイグラム $T[i,i+1]$ に対応する非終端記号を $C(i)$ と書くことにする．この非終端記号 $C(i)$ は，D 中に追加された順番で番号付けされている．以降では，$C(i)$ とその D 中の索引番号とを同一視する．例えば，$T[i,i+1]=\texttt{ab}$ で，$\alpha_{256}\rightarrow\texttt{ab}$ が D 中に存在しているとすると，$C(i)=256$ である．バイグラム $T[i,i+1]$ に対応する非終端記号が D 中に存在しない場合には，便宜的に，$C(i)=\infty$ とする．テキスト中のバイグラム $T[i,i+1]$ は，$C(i)$ の番号が小さいほど優先的に置換される．ただし，その前後のテキストの状況によっては，必ずしも置換されるとは限らないことに注意する．

LT-RePair の置き換え方について，次の補題が成り立つ．

補題 1 ある整数 i（$0\leq i<|T|-1$）について，$h(T[i])<h(T[i+1])$ または $C(i)=\infty$ ならば，バイグラム $T[i,i+1]$ は LT-RePair によって置換されない．

証明 1 LT-RePair の生成規則の制約から明らか．

この補題 1 から，次の補題 2 が導ける．

補題 2 ある整数 i $(0<i<|T|-1)$ について，$C(i)=\infty$ の場合を考える．このとき，$k=\arg\min_{0\leq j<i}C(j)$ とすると，$C(k)\neq\infty$ ならば，バイグラム $T[k,k+1]$ は LT-RePair によって置換される．逆に，$C(k)=\infty$ ならば，$T[0,i]$ は LT-RePair によって置換されない．

証明 2 補題 1 より，$C(i)=\infty$ なので，バイグラム $T[i,i+1]$ は置換されない．また，$T[i+1,i+2]$ が置換されて文字 c に書き換わったとしても，$h(T[i])<h(c)$ となるので，補題 1 より，$T[i]$ と右隣の文字のバイグラムは決して置換されない．したがって，$T[0,i]$ 中で最も優先度が高いバイグラムの置換が確定する．ただし，$T[0,i]$ 中のすべてのバイグラムが辞書に登録されていないのならば，$T[0,i]$ は今後書き換わることはない．

上述の補題 2 の前半は，$C(k)$ の置き換えが LT-RePair の置き換えと矛盾しない場合について述べている．後半の命題は，T の先頭部分から長さ i までの部分 $T[0,i]$ が LT-RePair で置換済みであるかどうかを保証するものである．この補題により，バッファの先頭部分を処理済みとして追い出すことができるかどうかが分かる．

次の二つの補題は，補題 2 の条件以外でバイグラムの置換が確定する場合についてのものである．

補題 3 ある整数 i（$0<i<|T|-1$）について，$h(T[i-1])=h(T[i])=h(T[i+1])$ かつ $\arg\min_{0\leq j<i}C(j)=i-1$ かつ $C(i-1)\leq C(i)$ ならば，バイグラム $T[i-1,i]$ は置換される．

証明 3 $\arg\min_{0\leq j<i}C(j)=i-1$ かつ $C(i-1)\leq C(i)$ より，$T[0,i+1]$ 中で最も優先度が高いバイグラムは $T[i-1,i]$ である．また，$h(T[i])=h(T[i+1])$ より，$T[i+1,i+2]$ が置換され，$T[i+1]$ が文字 c に書き換わったとしても，$h(T[i])<h(c)$ となるので，$T[i]$ より右側の文字列は $T[i]$ に影響しない．したがって，バイグラム $T[i-1,i]$ の置換が確定する．

補題 4 ある整数 i（$0<i<|T|-1$）について，$h(T[0])\geq\cdots\geq h(T[i-1])>h(T[i])=h(T[i+1])$ の場合を考える．このとき，$k=\arg\min_{0\leq j<i}C(j)$ について，$C(k)<C(i)$ ならば，バイグラム $T[k,k+1]$ が置換される．

証明 4 補題 1 より，$C(i)=\infty$ なので，バイグラム $T[i,i+1]$ は置換されない．また，$T[i+1,i+2]$ が置換され，$T[i+1]$ が文字 c に書き換わったとしても，$h(T[i])<h(c)$ となるので，補題 1 より，$T[i]$ と右隣の文字のバイグラムは決して置換されない．したがって，$T[0,i]$ 中で最も優先度が高いバイグラムの置換が確定する．

4.2 SemiOnlineReplace

前述の LT-RePair と同じテキスト置換をオンライン的に実行するアルゴリズム，SemiOnlineReplace について述べる．SemiOnlineReplace では，テキストを先頭から 1 文字ずつバッファに詰め込んでいき，バッファ中で置き換えが確定するバイグラムが存在すれば順に置き換えていく．また，これ以上テキストを読み込んでも置き換えられることはないと確定したバッファの先頭部分はバッファから出力する．

4.2.1 アルゴリズム

アルゴリズムの流れを Algorithm 1 に示す．T は置き換え対象の長さ n のテキスト，B は一時的に置き換え途中のテキスト部分を保持するバッファである．ここで，バッファは双方向連結リストとして実現されているものとする．B 中の文字へのポインタを p とするとき，$h(p)$ は p が指し示す文字（非終端記号）の高さを表す．$p.prev$ は左隣の文字，$p.next$ は右隣の文字へのポインタであり，文字が存在しない場合は NIL を示す．$C(p)$ は p と $p.next$ のバイグラムが辞書 D に登録されている場合，その置き換え文字を返し，登録されていない場合は ∞ を返す．RMQ(p) は，区間最小クエリ処理を意味する．すなわち，B の先頭から p までの文字列中で，置き換え文字が最も小さくなるようなバイグラムの左文字へのポインタを返す．ただし，すべてのバイグラムが D に登録されていない場合は NIL を返す．UpdateST は，RMQ を実行するためのセグメント木の更新処理である．Replace(p) は，p と $p.next$ のバイグラムを D に登録されている置き換え文字に置換する処理である．Output(p) は，B の先頭から p までの文字列をバッファから追い出し，圧縮テキストとして出力する処理を指す．

4.2.2 アルゴリズムの正当性

前述した SemiOnlineReplace が，正しく LT-RePair と同じ置換処理を行うかどうかについて議論する．まず，前節の LT-RePair の補題から，以下の補題が成り立つ．

Algorithm 1 SemiOnlineReplace

```
 1: procedure MAIN
 2:     T := T[1, n]
 3:     T[n + 1] ← dummy
 4:     B := ∅
 5:     for i := 1, n + 1 do
 6:         B.append(T[i])
 7:         last_pos := B.tail
 8:         RecursiveReplace(B, last_pos.prev)
 9:     end for
10: end procedure
11: procedure RECURSIVEREPLACE(B, p)
12:     if p = NIL OR p.next = NIL then
13:         return
14:     end if
15:     if h(p) > h(p.next) then
16:         if h(p.prev) = h(p) AND C(p.prev) > C(p) then
17:             UpdateST(p.prev)
18:         else
19:             UpdateST(p)
20:         end if
21:         return
22:     else
23:         if p.prev = NIL OR h(p.prev) = h(p) then
24:             m ← p.prev
25:         else
26:             m ← RMQ(p.prev)
27:         end if
28:         while C(m) ≠ ∞ AND C(m) ≤ C(p) do
29:             m′ ← m.next
30:             Replace(m)
31:             RecursiveReplace(m.prev)
32:             RecursiveReplace(m)
33:             if m′ = p then
34:                 return
35:             end if
36:             m ← RMQ(p.prev)
37:         end while
38:         if C(m) = ∞ AND C(p) = ∞ then
39:             Output(p)
40:         end if
41:     end if
42: end procedure
```

補題 5　辞書 D と文字列 $T = T[0, |T| - 1]$ が与えられたとする．このとき，ある整数 i（$0 \leq i < |T| - 1$）について，$T[0, i]$ が LT-RePair で置き換えも出力も確定できない場合，$h(T[i])$ は i に関して単調減少している．

証明 5　$h(T[i])$ が i に関して単調減少していない，即ち $h(T[j]) < h(T[j + 1])$ となる j（$0 \leq j \leq i$）が存在すると仮定する．このとき，補題 2 より，$T[0, j]$ 中のバイグラムの置換か $T[0, j]$ の出力が確定するはずである．したがって，補題の対偶が真であるので，補題は真である．

補題 6　辞書 D と文字列 $T = T[0, |T| - 1]$ が与えられたとする．いま，ある整数 j（$0 < j < |T| - 1$）が存在し，任意の整数 $0 < i < j$ について $h(T[i-1]) = h(T[i]) = h(T[i+1])$ が満たされているとする．このとき，$T[0, j]$ が LT-RePair で置き換えも出力も確定できない場合，バイグラム $T[i-1, i]$ が $T[0, j]$ 中で最も優先度が高くなることはない．

証明 6　バイグラム $T[i - 1, i]$ が $T[0, j]$ 中で最も優先度が高いと仮定する．このとき，$h(T[i - 1]) = h(T[i]) = h(T[i + 1])$ と補題 3, 4 より，バイグラム $T[i - 1, i]$ の置き換えが確定する．したがって，補題の対偶が真なので，補題は真である．

したがって，上述の補題から次の定理が導かれる．

定理 1　辞書 D と文字列 $T = T[0, |T| - 1]$ が与えられたとする．ある整数 i（$0 < i < |T|$）について，$T[0, i]$ のどの部分も LT-RePair で置き換えも出力も確定できず，また $h(T[i - 1]) = h(T[i])$ であるとき，$T[0, i]$ で最も優先度が高いバイグラムは $T[i - 1, i]$ である．

証明 7　補題 5 と条件より，文字列 $T[0, i]$ が置き換えも出力も確定できないのならば，$h(T[0]) \geq \cdots \geq h(T[i - 1]) = h(T[i]$ である．また，任意の j（$0 < j < i$）について，$h(T[j - 1]) > h(T[j]) = h(T[j + 1])$ ならば，補題 4 と条件より，文字列 $T[0, j + 1]$ 中で最も優先度が高いバイグラムは $T[j, j + 1]$ である．以上の議論と補題 6 より，$T[i - 1, i]$ が $T[0, i]$ 中で最も優先度が高いバイグラムとなる．

以降では，上述した補題・定理を基に，アルゴリズムの流れについて解説する．

1-10 行目は，テキストを読み出しながら RECURSIVEREPLACE を再帰的に実行するメインループ部分である．まず，テキスト T と双方向連結リスト B の初期化を行う．次に，T の先頭から 1 文字ずつ B の末尾に追加しつつ，現時点のバッファに対して RECURSIVEREPLACE を実行する．

11 行目からは再帰処理である．この処理が呼び出された時点では，B の先頭から p までの文字列だけではバイグラムの置き換えもバッファの追い出しも確定できない状態である．そこで，新たに $p.next$ の文字を参照し，p と $p.next$ のバイグラムの情報から B から $p.next$ までの文字列でバイグラムの置き換えか出力が確定できないかを判断する．

12-14 行目は再帰処理の 1 つ目の基底ケースである．p か $p.next$ の少なくとも片方の文字が欠けているならば，return して次の文字を参照する．

15-21 行目は再帰処理の 2 つ目の基底ケースである．この場合（$h(p) > h(p.next)$ の場合），$C(p)$ がどんな値であろうと，補題 2-4 が適用できないので置き換えも出力も確定できない．したがって，return して次の文字を参照する．ただし，後で用いる RMQ のために，$h(p.prev) = h(p)$ ならば $min(C(p.prev), C(p))$ で，そうでないならば $C(p)$ でセグメント木を更新する．

22-41 行目は再帰ケースである．23-27 行目で，B の先頭から p までの文字列中の最も優先度が高いバイグラムの左文字の位置を m に代入する．ここで，定理 1 より，$h(p.prev) = h(p)$ ならば，$p.prev$ と p のバイグラムが最も優先度が高いことが確定する．$p.prev = NIL$ のときは，そ

もそもバイグラムが存在しないので，m に NIL を代入する．$h(p.prev) > h(p)$ のときは，RMQ を用いて計算する．

28-37 行目はバイグラムの置き換えを行う．$h(p) \leq h(p.next)$ より，while の条件式を満たす場合には，補題 2, 3 からバイグラムの置き換えが確定する．置き換えた後は m の文字が新しい文字に書き換わるので，m に影響する位置に関して左から順に再帰処理を実行する．このとき，選択した m の右隣が p であった場合，p の文字は置き換えによって消滅して $C(k)$ が変わるので再帰を抜ける．そうでない場合，バイグラムの置き換えでは p の文字は消滅せず，再帰処理に関しても m より右側の文字列が書き換わることはないので $C(k)$ は変化しない．さらに，$m' \neq p$ より，$p.prev$ も変化しないので，36 行目に到達した時点では $h(p.prev) > h(p)$ である．したがって，RMQ を用いて m を再度計算することで while ループを繰り返して置き換え可不可の判定ができる．

38-40 行目では，補題 2 より，B の先頭から p までの文字列の出力が確定するので出力を行う．

RMQ に関して，アルゴリズム全体を通して RMQ を用いているのは，26 と 36 行目である．RMQ を呼び出す時は，$h(B.head) \geq \cdots \geq h(p.prev) > h(p)$ となっていることがアルゴリズムの流れから確定する．また，定理 1 と補題 5 より，B の先頭から p までの文字列中で最優先となり得るバイグラムは，各高さの最も右側の文字を含むバイグラムのいずれかである．したがって，RMQ のためのセグメント木の更新は 15-21 行目だけでよく，セグメント木の要素数も $\hat{g}$ で済ませられる．

4.3 計算量解析

バイグラムの置き換えに対応する生成規則の数を g とする．すなわち，$g = |V| - |\Sigma|$ である．また，非終端記号の構文木の高さの最大を $\hat{g}$ と定義する．

まず，SemiOnlineRepalce の時間計算量について議論する．Algorithm 1 において，RecursiveReplace が呼び出される回数は $n + r$ 回である．ここで，r はバイグラムの置換が起こった総数である．Re-Pair の時間計算量解析と同様に，r は n で抑えられることに注意する．RecursiveReplace が 1 回呼び出される毎に，12-14 行目の基底ケースでは定数時間かかる．15-21 行目の基底ケースではセグメント木の更新に $O(\log \hat{g})$ 時間かかる．基底ケースではバイグラムの置き換えは発生しない．再帰ケースでバイグラムの置き換えが発生しないときは，26 行目の RMQ が 1 回実行されるので，$O(\log \hat{g})$ 時間かかる．バイグラムの置き換えが発生する場合では，バイグラムの置き換え毎に 26 行目と 36 行目にある RMQ が実行される．したがって，全体では $O(n \log \hat{g})$ 時間である．

次に，領域計算量について議論する．Algorithm 1 で用いるデータ構造は，置換と出力が確定していない文字列を保持する双方向連結リストのバッファ，バイグラムから置き換え文字を特定するためのハッシュ，最小区間クエリを行うためのセグメント木である．

バッファの中に入る最大の文字数を考える．バッファ中の文字列が置き換えも出力も確定していないとき，文字の高さはバッファの先頭から単調減少している．また，部分文字列の各文字の高さが等しく，置き換えも出力も確定しない場合，その部分文字列の中のバイグラムの置き換え文字はすべて辞書に登録されており，かつ先頭から順に常に小さくなっている．よって，置き換えも出力も確定していないとき，バッファ中のバイグラムはすべてユニークであり，辞書に登録されていないバイグラム $\alpha_l\alpha_b$ が存在するとすれば，$h(\alpha_l) > h(\alpha_r)$ である．したがって文法規則の構文木の高さの最大を $\hat{g}$ とすると，バッファに入る最大の文字数は $g + \hat{g}$ である．明らかに $\hat{g} < g$ であるので，バッファの使用領域は $O(g)$ である．

ハッシュについては，置き換え文字の数が g であるので，ハッシュのデータ構造は $O(g)$ サイズの領域で実現できる．

セグメント木の領域について考える．セグメント木は要素数 n に対して $O(n)$ サイズで実現できる．アルゴリズムより，セグメント木で管理するのはバッファ中の文字列で文字の高さが等しく連続してる部分の右端だけで良い．よって，要素数 $\hat{g}$ のセグメント木を構築すればよいので，セグメント木の使用領域は $O(\hat{g})$ である．

以上より，提案手法のアルゴリズムで用いるデータ構造の領域計算量は $O(g)$ である．

5. 実験

5.1 実験方法

提案手法の有用性を評価するため，提案アルゴリズム SemiOnlineReplace と元のオフラインなアルゴリズムとの比較実験を行った．ここで，比較したオフラインのアルゴリズムとは，Larsson と Moffat ら [4] の手法に基づいて，与えられた辞書で Re-Pair に則って置換を行うものである．以降は，これを Larsson&Moffat と記述する．

実験に使用したデータは，Web サイト Pizza&Chili Corpus (http://pizzachili.dcc.uchile.cl/index.html) より入手した 100MB の DNA の塩基配列シーケンスである．実験では，データの先頭 1MB に対して LT-RePair を適用して辞書を生成した後，残り 99MB をその辞書を用いて置換するという処理を行い，そのメモリ使用量と計算時間を測定した．

実験環境は，プロセッサ intel®Xeon (R) CPU E3-1225 V2@3.20GHz，メモリ 15.6GiB のマシンを用い，オペレーティングシステム Ubuntu 12.04 LTS (64bit) 上で測定を行った．各プログラムは C++で実装し，コンパイラは GCC (version 4.6.3) を用いた．

表 1 実行時間と使用メモリ量の比較

	実行時間 (s)	使用メモリ量 (MB)
Larsson&Moffat	25.20	1520
SemiOnlineReplace	25.78	22

5.2 結果と考察

実験結果を表 1 に示す．実験結果から，提案手法は実行時間をほとんど犠牲にしていないにも関わらず，使用メモリ量を劇的に少なく抑えられていることがわかる．

先に示したとおり，時間計算量は Larsson&Moffat が $O(n)$ であるのに対し，提案手法は $O(n \log \hat{g})$ である．これは，再帰関数の実行毎に区間最小クエリを行うという最悪時のケースを仮定したものである．実験結果からは，実際には区間最小クエリの実行回数はそれほど多く発生しなかったと推測できる．使用メモリ領域に関しては，Larsson&Moffat がテキスト長の十数倍ものデータ構造をメモリ上に保持するのに対し，提案手法は辞書に関するデータ構造以外はバッファ分のデータに抑えられていることがわかる．

6. おわりに

本稿では，Larsson と Moffat ら [4] によって提案されたオフラインの文法圧縮である Re-Pair アルゴリズムを基に，オンライン的に実行できる文法変換アルゴリズムを提案した．提案アルゴリズムは，Re-Pair の文法規則に左優先の制約を課すことで，元の置き換え方と同じ文法変換を，バッファを用いた半オンライン処理で実行する．入力テキスト長を n，生成規則の数を g，生成規則の構文木の高さの最大を $\hat{g}$ とすると，文法変換の処理に $O(n \log \hat{g})$ 時間かかり，使用バッファのサイズは高々 $O(g)$ で抑えられる．実際，実験により，基のオフラインな変換処理と比べて，ほぼ同等の処理速度を保ちつつも，劇的に消費メモリ量を抑えられることを示した．

今後の課題としては，元の Re-Pair と同様に $O(n)$ 時間で変換処理を行う改善手法の開発や，質の良い辞書を適応的に構築しながらオンラインで文法変換を行う新規な文法圧縮の開発などが挙げられる．

謝辞

本研究は JSPS 科研費 15K00002 および 24240021 の助成を受けたものです．

参考文献

[1] Gage, P.: A new algorithm for data compression, *The C Users Journal*, Vol. 12, No. 2 (1994).
[2] Kieffer, J. C., E.-H. Yang, G. N. and Cosman, P.: Universal Lossless Compression via Multilevel Pattern Matching, *IEEE Trans. Inform. Theory*, Vol. 46, No. 4, pp. 1227–1245 (2000).
[3] Kieffer, J. C. and Yang, E.-H.: Grammar-Based Codes: a New Class of Universal Lossless Source Codes, *IEEE Trans. on Inform. Theory*, Vol. 46, No. 3, pp. 737–754 (2000).
[4] Larsson, N. J. and Moffat, A.: Offline Dictionary-Based Compression, *Proceedings of the Data Compression Conference 1999 (DCC '99)*, IEEE Computer Society, pp. 296–305 (1999).
[5] Larsson, N. J. and Moffat, A.: Off-line dictionary-based compression, *Proceedings of the IEEE*, Vol. 88, No. 11, pp. 1722–1732 (2000).
[6] Maruyama, S., Tanaka, Y., Sakamoto, H. and Takeda, M.: Context-Sensitive Grammar Transform: Compression and Pattern Matching, *Proc. of 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, pp. 27–38 (2008).
[7] Maruyama, S., Tabei, Y., Sakamoto, H. and Sadakane, K.: Fully-online grammar compression, *Proceedings of the 20th international conference on String processing and information retrieval (SPIRE 2013)*, pp. 218–229 (2013).
[8] Nevill-Manning, C., Witten, I. and Maulsby, D.: Compression By Induction of Hierarchical Grammars, *Proc. of the Data Compression Conference 1994 (DCC '94)*, IEEE, pp. 244–253 (1994).
[9] Sakamoto, H., Kida, T. and Shimozono, S.: A Space-Saving Linear-Time Algorithm for Grammar-Based Compression, *String Processing and Information Retrieval*, Lecture Notes in Computer Science, Vol. 3246, Springer Berlin / Heidelberg, pp. 218–229 (2004).
[10] Sekine, K., Sasakawa, H., Yoshida, S. and Kida, T.: Variable-to-Fixed-Length Encoding for Large Texts Using Re-Pair Algorithm with Shared Dictionaries, *Proceedings of the Data Compression Conference 2013 (DCC 2013)*, p. 518 (2013).
[11] Sekine, K., Sasakawa, H., Yoshida, S. and Kida, T.: Adaptive Dictionary Sharing Method for Re-Pair Algorithm, *Data Compression Conference (DCC), 2014*, pp. 425–425 (online), DOI: 10.1109/DCC.2014.73 (2014).
[12] Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T. and Arikawa, S.: Speeding Up Pattern Matching by Text Compression., *Proc. of the 4th Italian Conference (CIAC2000)*, pp. 306–315 (2000).
[13] Wan, R. and Moffat, A.: Block merging for off-line compression, *Journal of American Society for Information Science and Technology*, Vol. 58, No. 1, pp. 3–14 (online), DOI: 10.1002/asi.v58:1 (2007).
[14] Yoshida, S. and Kida, T.: A Variable-length-to-fixed-length Coding Method Using a Re-Pair Algorithm, *IPSJ Transactions on Databases*, Vol. 6, No. 4, pp. 17–23 (2013).
[15] 関根渓, 笹川裕人, 吉田諭史, 喜田拓也：共有辞書を用いた効率の良い圧縮アルゴリズム, 電子情報通信学会技術研究報告. DE, データ工学, Vol. 112, No. 346, pp. 47–52 (オンライン), 入手先 ⟨http://ci.nii.ac.jp/naid/110009667169/⟩ (2012).
[16] 笹川裕人, 関根渓, 吉田諭史, 喜田拓也：簡潔索引を用いた VF 符号上の部分文字列抽出, 情報処理学会研究報告. AL, アルゴリズム研究会報告, Vol. 2014, No. 8, pp. 1–5 (オンライン), 入手先 ⟨http://ci.nii.ac.jp/naid/110009785568/⟩ (2014).