Vol.2015-AL-154 No.2
2015/9/28

制約付き Re-Pair に基づいた適応型ブロック伸長法によるデータ圧縮アルゴリズム

正木 拓也[1]　喜田 拓也[1]

概要：
Re-Pair アルゴリズムは，入力テキスト長に対して線形時間で動作し，優れた圧縮率を達成することのできる文法圧縮アルゴリズムである．ただし，その動作はオフライン的であるため，テキスト全体を一度にメモリ上に読み込む必要がある．この問題に対し，従来，入力テキストを固定長のブロックに分割し，ブロック毎に Re-Pair を適用する手法が取られている．その手法で良好な圧縮率を達成するには，あらかじめ適切なブロック長を与える必要がある．我々はこれまでに，Re-Pair によって生成される文法にある種の条件を設けることで，Re-Pair アルゴリズムと同等のテキスト置換を与えられた辞書を用いてオンライン的に実行することのできるアルゴリズムを得ている．本稿では，そのオンライン置換アルゴリズムを基に，ブロック長を適応的に伸長させながら圧縮する手法を提案する．

1. はじめに

Larsson と Moffat ら [1] 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 (バイグラム) 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 [2]．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 [7].

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 [6] 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 [4, 5, 8] 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 [2] は，FOLCA と呼ばれる手法を提案している．FOLCA は，Sakamoto ら [3] 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

[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

を可能としている．他の文法圧縮同様に，繰り返し部分の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 [9]．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．

これまでに我々は，Re-Pair の文法規則の生成方法にある制約を課した変種である *LT-RePair* を提案し，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**を示した [10]．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文法変換を行うことができる．実際，圧縮率や圧縮速度などの性能をほぼ犠牲にすることなく，メモリ消費量を劇的に抑えることができた．ただし，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た．

本稿では，先に提案した LT-RePair と SemiOnlineReplace アルゴリズムを用いて，大規模なテキストに対して適応的にブロック分けしつつ圧縮を実行するアルゴリズムを提案する．提案アルゴリズムは，入力テキストの前方から逐次的にオフライン的な置換アルゴリズムとオンライン的な置換アルゴリズムを交互に適用しながら，辞書構築と文法変換を実行する．それにより，メモリ消費量を抑えながら，可能な限り長大なブロック長を自動的に選択することができる．分割された各ブロックは，ブロック毎に適当な符号化が適用される．元の Re-Pair アルゴリズムの文献 [1] では可変長符号化を用いる手法が示されているが，圧縮後のデータの取り扱いの簡便さからは固定長の符号化を用いる方法 [7] を用いることもできる．

2. 準備

本節では，文献 [10] にしたがって二つのアルゴリズム，LT-RePair と SemiOnlineReplace について簡潔に述べる．

2.1 LT-RePair

先に述べたとおり，LT-RePair は元の Re-Pair に制約を付けた変種である．基本的には Re-Pair と同様，テキスト中に出現する最頻のバイグラムを新たな非終端記号に置き換える．この置換処理を最頻なバイグラムがなくなるまで繰り返すことで文法変換を行う．ただし，置換のために選択されるバイグラム XY は，$h(X) \geq h(Y)$ となるもの (LeftTall) のみを許す．ここで，$h(X)$ とは非終端記号 X の構文木の高さである．したがって，LT-RePair 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．

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

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

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常に高さが等しいか大きい．文献 [1] で示されているように，Re-Pair は入力テキスト長 n に対して $O(n)$ 時間で実行できる手法が知られている．LT-RePair も同様に $O(n)$ 時間で実行可能である．ただし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し，元のテキストの十倍以上のメモリを消費する．

2.2 SemiOnlineReplace

SemiOnlineReplace は，置換のための辞書 D（文法規則の集合）が与えられたとき，その辞書 D での LT-RePair と同じ置換を，テキストを逐次的に読み込みながら実行するアルゴリズムである．SemiOnlineReplace の大まかな流れは次のとおりである．まず，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．アルゴリズムの詳細を 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）の構文木の高さを表す．$p.prev$ 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．

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，SemiOnlineReplace による文法変換の処理は，全体で $O(n \log \hat{g})$ 時間かかり，使用バッファのサイズは $O(g)$ かかることが示されている [10]．アルゴリズム内部で区間最小クエリ処理を行っているた

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
```

め，LT-RePair の置換処理よりも時間計算量が大きくなっている．

3. 提案手法

本節では，提案手法である LT-RePair に基づいた適応型ブロック伸長法 (*AdaptiveBlockExpand*) について述べる．テキストの先頭部分（初期ブロック）のみに LT-RePair を実行して辞書を構築し，残りのテキスト全体に SemiOnlineReplace を適用させることで，初期ブロック長と辞書サイズに線形サイズな使用メモリ領域でテキスト全体を圧縮することが可能となった．しかし，初期ブロックのみで構築された辞書では圧縮率が芳しくない．そこで提案手法では，初期ブロック領域を効率良く利用し，実質的にブロックを伸長する．伸長することで，より長いテキスト構造を把握した辞書で圧縮することが可能となる．また，初期ブロックを限界まで利用した後はそこでブロックを打ち切り，新しい初期ブロックから開始することで，適用的にブロック分けを行う．

提案手法の流れを以下に示す．

(1) テキスト T を長さ m だけ読み込み，初期ブロック F に詰め込む．
(2) F 中の LeftTall な最頻バイグラムを新しい記号に置換し，その規則を辞書 D に追加する．
(3) SemiOnlineReplace で残りのテキストを置換しながら F の空きスペースに詰め込む．
(4) F 中の全てのバイグラムがユニークならば，F と D を出力する．そうでないならば，(2) に戻る．
(5) T を全て読み込めていないならば，(1) に戻る．

ここで，m は初期ブロックの長さである．また，固定長符号化を行う場合は (4) の条件を $|\Sigma| + |D| = 2^l$ との論理和に変更する．ただし，l は指定する固定長符号語長である．

結果として，ブロックを実質的に伸長しながら，LT-RePair が終了もしくは固定長符号語を使い切ったところでブロックを打ち切る．初期ブロックの短さに対して，圧縮後のブロックは長いテキストをまとめている．圧縮ブロックの数は少なくて済み，固定長符号化の場合はどのブロックにおいても符号語長が等しくなる．

3.1 アルゴリズム

アルゴリズムの詳細を Algorithm 2 に示す．T は圧縮対象の長さ n のテキストである．F は長さ m の初期ブロックであり，連結リストで実現されているものとする．D はバイグラムの置き換え規則を登録する辞書である．GetMaxPair(F) は F 中の LeftTall な最頻バイグラムを返し，全てのバイグラムがユニークな場合は NIL を返す．AddRule(D, p) は D にバイグラム p から非終端記号への置き換え規則を登録する．Output(F, D) は圧縮テキスト F と辞書 D を出力する．Replace(F, p) は F 中存在する全ての p を非終端記号に置換し，その出現数を返す．SemiOnlineReplace($F[m - cb, m], T[f, n]$) は空きスペース $F[m - cb, m]$ に残りテキスト $T[f, n]$ を SemiOnlineReplace で逐次的に読み込みながら置換して詰め込み，読み込んだ文字数を返す．F の空きに全て詰め込み終わった場合は T の読み込みを中断する．

アルゴリズムの流れについて解説する．2-3 行目は初期化処理であり，f はまだ読み込んでいないテキストの開始位置を示す．4 行目の while 文は元テキストの読み込みが全て終わっているかどうか判定し，終わっていないならば新たな初期ブロックから処理を続ける．5-8 行目は LT-Repair と SemiOnlineReplace を実行するための初期化処理である．T 中の位置 f から m 文字分読み込んで F に詰め込み，初期

Algorithm 2 AdaptiveBlockExtend

```
 1: procedure MAIN
 2:     T := T[1, n]
 3:     f := 1
 4:     while f ≤ n do
 5:         D := ∅
 6:         F[1, m] = T[f, f + m − 1]
 7:         f := f + m
 8:         p := GetMaxPair(F)
 9:         while p ≠ NIL do
10:             AddRule(D, p)
11:             cb := Replace(F, p)
12:             ct := SemiOnlineReplace(F[m − cb, m], T[f, n], p)
13:             f := f + ct
14:             p := GetMaxPair(F)
15:         end while
16:         Output(F, D)
17:     end while
18: end procedure
```

ブロックを作成する．読み込んだことで残りテキストの開始位置が変わるので，f を $f+m$ に変更する．9-15 行目の while 文では条件判定し，LT-RePair と SemiOnlineReplace を続けるかどうか決定する．p =NIL ならば最頻のバイグラムが存在しないので，これ以上 LT-RePair を実行できない．cb には置換したバイグラムの数を代入することで，F の空いたスペースが判明する．そのスペースを引数に SemiOnlineReplace を実行し，ct には読み込んだ文字数を代入する．その後，ct を元に位置 f を更新する．この while 文を抜けた後は F と D が圧縮テキストと辞書になっているので，これらを出力し，まだテキスト全体を読み込んでいないのならば，4 行目に戻り，次の初期ブロックを新たに作成して処理を続ける．

3.2 計算量解析

各ブロックで処理過程で出現したバイグラムの種類数のうち最大のものを，b' とする．各ブロックで辞書に登録された文法規則の数のうち最大のものをを g' とする．また，各ブロックでの非終端記号の構文木の高さの最大のうち最大のものを $\hat{g}'$ と定義する．

まず，時間計算量について議論する．Algorithm2 において，AddRule は生成規則を登録するだけなので 1 回当たり定数時間，GetMaxPair もハッシュを利用することで 1 回当たり定数時間で実行できる．全体で Replace によって置換されるバイグラムは高々 n 個である．Replace ではポインタを辿ることによってバイグラム 1 つにつき定数時間で置換することができる．よって，Replace による処理時間は全体で $O(n)$ 時間である．全体の SemiOnlineReplace によって読み込まれる総計のテキスト長は高々 n であり，非終端記号の構文木の高さの最大は，どの地点でも高々 $\hat{g}'$ である．よって，SemiOnlineReplace による処理時間は全体で $O(n \log \hat{g}')$ 時間である．Output は圧縮テキストと辞書を出力するだけなので，全体で $O(n)$ 時間である．5-8 行目の初期化処理は全て定数時間であり，4 行目の while ループも高々 n 回で抑えられるので，これらの処理は全体で $O(n)$ 時間である．9-15 行目の while 文では，1 回ループ処理処理を行う毎に少なくとも 2 つのバイグラムが置換される．よって，この while 文のループ回数はアルゴリズム全体でも高々 n 回である．while 文内の Replace と SemiOnlineReplace 以外の処理は定数時間なので，これらの処理は全体で $O(n)$ 時間である．以上をまとめると，アルゴリズム全体で $O(n \log \hat{g}')$ 時間である．

次に，領域計算量について議論する．提案手法は LT-RePair と SemiOnlineReplace の組み合わせであり，全体の使用メモリ領域はこれら 2 つの手法の使用メモリ領域の総和である．ただし，提案手法は実質的に初期ブロックを拡張してさらにテキストを読み込んでいるため，LT-RePair の使用メモリ領域は $O(m)$ では抑えられない．LT-RePair ではデータ構造として，読み込んだテキストを保持するシーケンス，最頻のバイグラムを取得するための優先度付きキュー，バイグラムからシーケンスや優先度付きキューの登録位置を特定するためのハッシュの 3 つを使用する．ここで，提案手法の各ブロックにおける LT-RePair について考える．テキスト保持のシーケンスは，初期ブロックを使いまわすので $O(m)$ 領域である．優先度付きキューとハッシュでは，出現したバイグラムの種類数に線形なメモリ領域を使用する．よって，各ブロックでデータ構造を再利用するとして，どちらも $O(b')$ 領域である．したがって，提案手法中の LT-RePair では $O(m+b')$ 領域を使用する．各ブロックでの SemiOnlineReplace は辞書中の文法規則数に線形なメモリ領域を使用する．各ブロックでそのメモリ領域を再利用することにより，常に SemiOnlineReplace での使用メモリ領域は $O(g')$ で抑えられる．以上をまとめると，提案手法の使用メモリ領域は $O(m+g'+b')$ である．また，g' は常に b' より小さいので，結局のところ $O(m+b')$ となる．

4. 予備実験

4.1 実験方法

提案手法の有用性を評価するため，提案手法と Re-Pair アルゴリズムとの予備的な比較実験を行った．

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，提案手法と Re-Pair ぞれぞれの圧縮率，メモリ使用量，実行時間を測定した．また，提案手法は可変長符号とし，初期ブロック長を 1MB と 5MB の 2 つの場合について測定した．

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

表 1 実行時間と使用メモリ量の比較

	圧縮率 (%)	メモリ (MB)	時間 (s)
Re-Pair	31.7	1403	22.8
提案手法 (5MB)	33.0	424	78.0
提案手法 (1MB)	33.8	142	82.9

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．

4.2 結果と考察

実験結果を表 1 に示す．実験結果から，提案手法の初期ブロック長が小さい程使用メモリ領域が少なくなるが，圧縮率と実行時間は悪化していることがわかる．提案手法を Re-Pair を比べると，圧縮率の悪化は微々たるものであるが，使用メモリ領域は大幅に改善されている．ただし，実行時間に関しては悪化している．また，初期ブロック長が 5MB の時はブロック数が 3 個，1MB の時はブロック数が 14 個であった．

圧縮率に関して，ブロック数が少ない程ブロック間で共通して存在する冗長な辞書の要素が少なくなるため，良くなったのだと推測できる．実行時間に関しては，SemiOnlineReplace の処理時間分，Re-Pair と比べて悪化している．使用メモリ領域に関しては，$O(m+b')$ の b' の分だけ，初期ブロックの小ささの割りにそれ程改善されなかったと思われる．現在のところ，プログラムの実装は十分に最適化されていない．今後，LT-RePair のデータ構造についての見直しを行う必要がある．

5. おわりに

本稿では，Larsson と Moffat ら [1] 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，LT-RePair と SemiOnlineReplace [10] を用いて大規模なテキストに対して適応的にブロック分けしつつ圧縮を実行す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，初期ブロック長を指定することで圧縮率と使用メモリ領域を調整可能な文法圧縮法となった．初期ブロック長を m，各ブロックでの生成規則の数のち最大を g'，各ブロックでの出現したバイグラムの種類数のうち最大を b'，各ブロックでの生成規則の構文木の高さの最大のうちの最大を $\hat{g}'$ とすると，文法変換の処理に $O(n \log \hat{g}')$ 時間かかり，使用メモリ領域は高々 $O(m+b')$ で抑えられる．また，実験で Re-Pair アルゴリズムと比較し，圧縮率の悪化は少ないながらも，使用メモリ領域を大幅に削減できることを示した．

今後の課題としては，実験で実装したプログラムを改善し，アルゴリズム本来の性能を示すことと，提案手法中の LT-RePair を改善することにより，領域計算量を落とすことが挙げられる．

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] 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).

[2] 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).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 関根渓, 笹川裕人, 吉田論史, 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).

[9] 笹川裕人, 関根渓, 吉田論史, 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).

[10] 正木拓也, 喜田拓也：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, 電子情報通信学会技術研究報告, Vol. 115, No. 84, pp. 37–43 (2015).