LA_002

単語幅を制約した接尾辞木の効率のよい構築アルゴリズム
Efficient Algorithm of Constructing Suffix Tree with Word Count Limitation

上村 卓史* 喜田 拓也* 有村 博紀*
Takashi Uemura Takuya Kida Hiroki Arimura

1. はじめに

情報爆発時代の今日において，膨大な量のデータを効率よく管理する技術はきわめて重要である．ウェブ上には多くの文書やデータベースが存在しており，それらの中から利用者が欲する情報をすばやく取り出す検索技術の開発は急務である．

接尾辞木は文字列データ（テキスト）の任意の部分文字列をコンパクトに索引付けするデータ構造である．対象とするテキストの長さを n とすると，その大きさは $O(n)$ 領域である．この接尾辞木を用いれば，長さ m の検索文字列（パターン）に対する照合を $O(m)$ 時間で行うことができる．接尾辞木をテキスト長の線形時間で構築する効率のよいアルゴリズム [6, 9] は古くから知られており，今日でも様々な拡張・応用が提案されている．

先に述べたとおり，接尾辞木はすべての部分文字列を $O(n)$ 領域で扱える優れたデータ構造ではあるが，実際の応用においては n 自身が巨大であり，非常に大きな領域を必要とする．したがって，これと同等な機能を持ち，かつより小さなデータ構造が望ましい．接尾辞木の応用という観点からは，ある程度以上に長い部分文字列が必要になる場面は少ない．例えばキーワード検索の場合，数語程度の長さの部分文字列に対して索引付けされていれば概ね十分である．また，処理済のテキストから次にくる文字を予測することで高い圧縮率を実現する PPM 法 [2] では，過去の部分文字列すべてではなく最近現れたものだけを保持しておくことで十分な圧縮性能と圧縮速度を実現している．

本稿では，与えられたテキストから k 個の単語幅の部分文字列のみを保持する新たなデータ構造（k 単語幅制約付き接尾辞木）を提案し，それがテキスト長の線形時間で構築できることを示す．ここで，与えられるテキスト T は，区切り文字 $\#$ によって単語の列 $T = w_1\#w_2\#\cdots\#w_N$ に分割されていると仮定する．このとき，与えられたテキストと整数 $k(1 \leq k \leq N)$ に対して，求めるデータ構造は，任意の $i(1 \leq i \leq N-k+1)$ について文字列 $w_i\#w_{i+1}\#\ldots\#w_{i+k-1}$ のすべての部分文字列を表現する．提案アルゴリズムは Ukkonen による接尾辞木構築アルゴリズム [9] に基づいており，長さ n の入力テキストに対して線形時間・領域でこのデータ構造を構築できる．そのアイデアは，Ukkonen のアルゴリズムにおいてはラベルの文字列が自動的に延長される葉ノードを，単語の切れ目ごとに適切に延長を打ち切ることにある．

関連研究 Ziv-Lempel 系圧縮法の実装での利用は接尾辞木の重要な応用の一つである．しかし，特に LZ77 圧縮系ではスライド窓方式によって参照できる過去のテキストが制限されており，そのままの接尾辞木では対応できない．この問題を克服する手法 [8, 3, 5] が既にいくつか提案されている．Na ら [7] は，入力テキストの幅 k 以下の部分文字列すべてを表す接尾辞木として切落し接尾辞木（*truncated suffix tree*）を定義し，それをテキスト長に比例した時間で構築するアルゴリズムを提案した．我々の提案アルゴリズムは，彼らのアイデアの延長線上にあるが，表現される部分文字列の長さが一定ではないため，接尾辞木の拡張を止めるタイミングが各葉ノードで異なる．

*北海道大学大学院情報科学研究科, Hokkaido University

2. 準備

Σ を有限のアルファベットとし，Σ 上の文字列全体の集合を Σ^* で表す．文字列 $x \in \Sigma^*$ の長さを $|x|$ で表す．特に長さが 0 の文字列を空語といい ε で表す．すなわち，$|\varepsilon| = 0$ である．また，$\Sigma^+ = \Sigma^* - \{\varepsilon\}$ と定義する．

文字列 $x_1, x_2 \in \Sigma^*$ の連結を $x_1 \cdot x_2$ で表す．ただし，特に混乱が生じないかぎりはこれを省略し，x_1x_2 と表す．

ある文字列 $w \in \Sigma^*$ に対して，$w = xyz$ となる $x, y, z \in \Sigma^*$ が存在するとき，x, y, z をそれぞれ w の**接頭辞**，**部分文字列**，**接尾辞**と呼ぶ．w の i 番目の文字を $w[i]$ と表し，w の i 番目から j 番目までの部分文字列を $w[i \ldots j]$ で表す．$i > j$ の場合，便宜上 $w[i \ldots j] = \varepsilon$ と定義する．また，文字列 $w \in \Sigma^*$ の接尾辞全体の集合を $Suf(w)$ と表し，w の部分文字列全体の集合を $Fac(w)$ で表す．

2.1 接尾辞木

接尾辞木は文字列 T の全ての部分文字列 $Fac(T)$ を表現するデータ構造である．文字列 T の接尾辞木は $ST(T) = (V \cup \{\perp\}, root, E, suf)$ の四つ組みで表される．ここで，V はノードの集合であり，$\perp \notin V$ とする．E は辺の集合であり，$E \subseteq V^2$ である．接尾辞木において，このグラフ (V, E) は $root \in V$ を根とする木を成している．すなわち，任意の $s \in V$ は $root$ からの唯一のパスが存在する．ここで，ノード $s, t \in V$ について，$(s, t) \in E$ のとき，s を t の**親**，t を s の**子**と呼ぶ．また，子を持つノードを**内部ノード**，子を持たないノードを**葉ノード**と呼ぶことにする．さらに，$root$ からノード $s \in V$ へのパス上にあるノードを s の**祖先**と呼ぶ．

E の各辺にはラベル文字列が与えられ，T 上の位置の組 (j, k) で表される．すなわち，このときのラベル文字列は $T[j \ldots k]$ である．子は親を 1 つだけ持つから，$s \in V$ へ向かう辺のラベル文字列を $label(s)$ で表すことにする．任意の $s \in V$ について，そのすべての祖先のラベル文字列を $root$ から順に連結したものを $\bar{s}$ と書き，これをノード s が**表す文字列**と呼ぶ．すなわち，$root, a_1, a_2, \ldots, s$ をノード s の祖先の列とすると，$\bar{s} = label(root) \cdot label(a_1) \cdot label(a_2) \cdots label(s)$ である．

与えられたテキスト T に対して，接尾辞木 $ST(T)$ 上の各葉ノードは T の接尾辞と一対一で対応し，$root$ を除く内部ノードは必ず子を二つ以上持つ．ただしここで，$T[n] = \$$ は $T[1..n-1]$ には含まれない文字（**終端記号**と呼ばれる）とする．すなわち，葉ノードは $|T|$ 個存在

し，全体では $2|T|-1$ 個の以下ノードを持つ．また，各 $s \in V$ の任意の二つの子は互いに異なる文字から始まるラベル文字列を持つ．よって，各ノード $s \in V$ は T のある特定の部分文字列と一対一に対応する．$ST(T)$ 上に対応するノードが存在しないその他の部分文字列については仮想ノードとして扱われる．これと区別するため，以降は V の要素を実ノードと呼ぶ．

接尾辞木上の任意のノード $s \in V$ と文字 $a \in \Sigma$ に対し，$\overline{s} \cdot a$ を表すノードを $child(s,a)$ と表す．$child(s,a)$ は仮想ノードの場合もある．接尾辞木上に対応するノードが無い場合は未定義とする．

関数 $suf : V \to V$ は，任意のノード $s \in V$ に対して $\overline{s}$ の先頭一文字を取り除いた接尾辞を表すノードを返す関数である．すなわち，ノード $s,t \in V$ と文字 $a \in \Sigma$ に対して $\overline{s} = a \cdot \overline{t}$ であるとき，$suf(s) = t$ である．実ノードについては，これはノード s から t への辺 (s,t) と見ることもでき，この辺を**接尾辞リンク**と呼ぶ．葉でない任意の実ノード $s \in V$ について，接尾辞リンクをたどって $root$ へいたるパスがただ一つ存在する．このパスを s の**接尾辞パス**と呼ぶ．

2.2 Ukkonen の接尾辞木構築アルゴリズム

Ukkonen[9] による接尾辞木の構築アルゴリズムを以下に概説する．このアルゴリズムは，与えられたテキスト $T = T[1 \ldots n]$ を先頭から一文字ずつ読み込みながら各時点 $i (1 \leq i \leq n)$ における $ST(T[1 \ldots i])$ を構築する．その構築は，任意の時点 i について，文字列 $T[1 \ldots i]$ の接尾辞（に対応するノード）を長いものから順に $ST(T[1 \ldots i-1])$ に追加していくことで行われる．ただし，構築の途中においては終端記号がないため，$T[1 \ldots i]$ の接尾辞に対応するノードは必ずしも $ST(T[1 \ldots i])$ の葉ノードではないことに注意する．このような接尾辞木は**暗黙的接尾辞木**と呼ばれる．

接尾辞木の拡張は接尾辞を表す葉ノードを木に追加することで行われる．この追加は長い接尾辞に対応するノードから順に行われる．現在葉ノードを新しく作ろうとしているノードを**作業ノード**と呼び，ϕ と表す．$i = 0$ においては $\phi = root$ とする．$child(\phi, T[i])$ が木に存在しなければ，$T[i]$ から始まるラベル文字列を持つ葉ノードを作る．このとき，ϕ が仮想ノードならば実ノードにする．また，接尾辞リンクも作る．ϕ が仮想ノードであった場合，この接尾辞リンクは明示的には保持されていないが，最も近い実ノードの祖先が持つ接尾辞リンクを利用することで ϕ の接尾辞 $suf(\phi)$ を手繰ることができる．

葉ノード v と辺を作るとき，ラベルの表記に特別な記号 $*$ を導入し，$label(v) = (i, *)$ とする．$*$ は現在の文字列長 i と等しいと解釈する．つまり，葉ノードは一度作られると自動的に T のある接尾辞を表すことになる．これで $\overline{\phi} \cdot T[i]$ という接尾辞を追加し終わったので，今度は現在より 1 文字短い接尾辞を追加するために ϕ を $suf(\phi)$ に更新する．そうして $child(\phi, T[i])$ が存在しないならば葉ノードを作っていき，$child(\phi, T[i])$ が見つかったところで更新を終了する．次の文字による拡張は $child(\phi, T[i])$ から行う．

このとき，$\overline{\phi} \cdot T[i]$ が T に現れたならば $Suf(\overline{\phi}) \cdot T[i]$ も必ず同じ終端位置で現れているので，接尾辞パス上のより短い接尾辞はすべて木に表されているということに注意する．

ここではその詳細については省略するが，これらの更新作業は全体をならしてみると追加するノードの数に比例した時間で完了できる．こうして得られた暗黙的接尾辞木 $ST(T[1 \ldots n])$ に終端記号 \$ を付け加えることで，最終的な接尾辞木が得られる．

3. k 単語幅制約付き接尾辞木

Σ に含まれない記号 $\#$ を考え，これを区切り文字と呼ぶことにする．文字列 $x \in (\Sigma \cup \{\#\})^*$ は，ある適当な文字列の並び $w_1, w_2, \ldots, w_N$ $(^\forall i, w_i \in \Sigma^*)$ によって $x = w_1 \# w_2 \# \cdots \# w_N$ と書き表すことができる．このとき，各 w_i を**単語**といい，文字列 w は $\#$ で区切られた N **単語の文字列**という．

いま，与えられるテキスト $T = T[1 \ldots n]$ を $\#$ で区切られた N 単語の文字列とする．ただし，議論の簡便のため，以降では任意の i について $w_i \in \Sigma^+$ とし，$\#$ は連続して出現しないと仮定する．すなわち，これは連続する $\#$ は区切り記号 1 個分とみなすことに対応する．このことは実際の応用において適切な仮定である．例えば，英語の文章において空白と改行の連続をひとつの区切りとみなすことは，十分に自然な仮定である．日本語の文章には単語間の区切り記号は存在しないが，あらかじめ適切な形態素解析ツールでもって区切られていると考えればよい．このとき，与えられたテキスト $T = w_1 \# w_2 \# \cdots \# w_N$ と整数 k $(1 \leq k \leq N)$ に対する **k 単語幅制約付き接尾辞木** (k-$WST(T)$) とは，任意の i $(1 \leq i \leq N-k+1)$ についての $Fac(w_i \# w_{i+1} \# \ldots \# w_{i+k-1})$ を表現するトライをパス圧縮した木である（図 1，2）。形式的には，k-$WST(T)$ は以下の条件を満たす根付き木である．

1. k-$WST(T)$ の各辺は，空語でない T の部分文字列でラベル付けされている．
2. k-$WST(T)$ 中の根以外の任意の内部ノード v は少なくとも二つ以上の子を持ち，v から子への辺はそれぞれ異なる文字で始まる文字列でラベル付けされる．
3. k-$WST(T)$ の各葉ノード v は，$\overline{v}$ はある文字列 $s \in Suf(w_i \# w_{i+1} \# \ldots \# w_{i+k-1})(1 \leq i \leq N-k+1)$ と一対一に対応する．

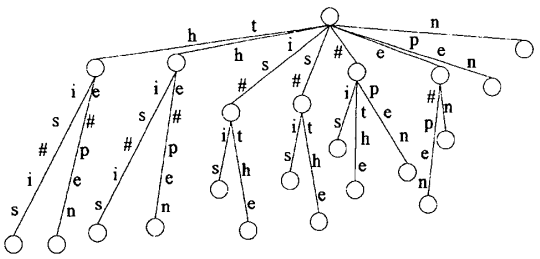

図 1: 2-$WST(T =$ "this#is#the#pen")

以下では，k-$WST(T)$ を構築するアルゴリズムを示し，さらにそれが $O(n)$ 時間領域で構築できることを示す．すなわち，本稿での結論は次のとおりである．

定理 1. 長さ n のテキスト $T = w_1 \# w_2 \# \cdots \# w_N$ $(\# \notin \Sigma$ かつ $^\forall i, w_i \in \Sigma^+)$ と整数 k $(1 \leq k \leq N)$ が与えられたとき，k 単語幅制約付き接尾辞木は $O(n)$ 時間領域で構築できる．

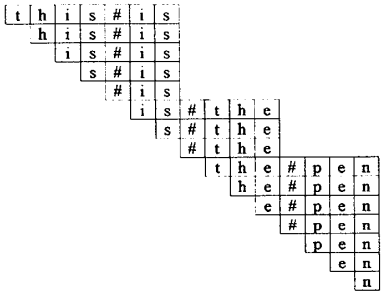

図 2: 2-$WST(T$ = "this#is#the#pen") で表現される接尾辞

3.1 構築アルゴリズム

アルゴリズムを図3に示す．重要なのは，更新の際にちょうど $k+1$ 単語となる直前で葉ノードの拡張を止めることである．この拡張を停止する操作は，葉ノードのラベルの終端 $*$ を $i-1$ に変えることで行われる．この操作を葉ノードを閉じるという．このとき，閉じた葉ノードのラベルには仮想的な終端記号 $\natural \notin \Sigma$ が付加されると考える．このようにして，$k+1$ 単語以上の長さの接尾辞が木に追加されることを防止することができる．

木の更新手続き 今 k-$WST(T[1\ldots i-1])$ が得られており，$T[i]$ に対して更新処理を行うとする．$\overline{\phi}$ の T での開始位置を $lpos_\phi$ とする．したがって，$\overline{\phi} = T[lpos_\phi \ldots i-1]$ である．また $\overline{\phi}$ が含む # の数を num_ϕ とし，$\overline{\phi}$ の単語数を管理する．$T[i] = \#$ で木を進むときは num_ϕ を 1 増やす．また，$suf(\phi)$ へ移動するとき，$lpos_\phi$ を 1 増やし，$T[lpos_\phi] = \#$ ならば num_ϕ を 1 減らす．

木の更新は基本的に Ukkonen のアルゴリズムと同様である．ただしここで，図4のように，ある k 単語の文字列が別の k 単語の文字列の接頭辞である場合は特別な処理が必要となる．このとき，短いほうの接尾辞も葉ノードでなければならない．その際葉のラベル文字列は ε となるが，仮想的に終端記号 $\natural$ を表していると考える．

葉ノードを閉じる処理 今，k-$WST(T[1\ldots i-1])$ に対して $T[i] = \#$ による拡張を行う場合を考える．このとき，ちょうど $k-1$ 個の # を含む接尾辞を表すすべての葉ノード v について，拡張を止める必要がある．このとき，最長の接尾辞を表す葉ノードから順に処理することで，該当するすべての葉ノードを閉じることができる．

今注目する葉ノードを ψ とする．更新手続きの場合と同様に，$\overline{\psi}$ 中の # の個数 num_ψ と，$\overline{\psi}$ の T での開始位置 $lpos_\psi$ を管理する．そして，$num_\psi = k-1$ である間，ψ のラベルの終端を $i-1$ として葉ノードを閉じていく．$num_\psi < k-1$ となった時点で $\overline{\psi} \cdot T[i]$ を表すノードが次回の ψ となる．

3.2 アルゴリズムの計算量

補題 1. 木の更新処理は全体で $O(n)$ 時間で行える．

証明. $i = 1, \ldots, n$ において，更新手続きは最大で n 個の葉ノードを作る．必要のあるときに新しく内部ノードを作り，さらに新しい葉ノードへの辺を作る作業は，定数時間で行える．1 度の更新手続きは最大で n 個の葉ノードを作る可能性があるが，最終的な葉ノードの個数は $O(n)$ 個であるから，これは全体では $O(n)$ 時間で処理できる． □

---

procedure $update(\phi, num_\phi, lpos_\phi, i)$;
method:
  $c = T[i]$; $oldr = root$;
  while ($\overline{\phi}c$ が木に表されていない) {
    if ($c \neq \#$ **or** $\overline{\phi}$ を表す閉じた葉が存在しない) {
      if (ϕ が仮想ノード)
        ϕ を実ノードにする;
      ϕ の子 q, $label(q) = (i, *)$ を作る;
      if ($oldr \neq root$)
        $suf(oldr) = \phi$;
      $oldr = \phi$;
    }
    $\phi = suf(\phi)$;
    if ($T[lpos_\phi] = \#$)
      $num_\phi = num_\phi - 1$;
    $lpos_\phi = lpos_\phi + 1$;
  }
  if ($oldr \neq root$)
    $suf(oldr) = \phi$;
  $\phi = child(\phi, c)$;
  if ($c = \#$)
    $num_\phi = num_\phi + 1$;
  return $(\phi, num_\phi, lpos_\phi)$;
end.

procedure $close(\psi, num_\psi, lpos_\psi, i)$;
method:
  if ($T[i] = \#$) {
    while ($num_\psi = k-1$) {
      ψ のラベルの終端を $i-1$ とする;
      if ($T[lpos_\psi] = \#$)
        $num_\psi = num_\psi - 1$;
      $lpos_\psi = lpos_\psi + 1$; $\psi = suf(\psi)$;
    }
    $num_\psi = num_\psi + 1$;
  }
  $\psi = child(\psi, T[i])$;
  return $(\psi, num_\psi, lpos_\psi)$;
end.

Algorithm *constructing_k-WST*;
input: $k, T[1\ldots n]$;
output: k-$WST(T)$;
method:
  木の根 $root$，および補助ノード $\perp$ を作り，
    辺 $(\perp, root)$ を張る;
  foreach $c \in \Sigma \cup \{\#, \$\}$ **do**
    c でラベル付けされた辺 $(\perp, root)$ を作る;
  $\phi = root$; $num_\phi = 0$; $lpos_\phi = 1$;
  $\psi = root$; $num_\psi = 0$; $lpos_\psi = 1$;
  for $(i = 1\ldots n)$ {
    $(\phi, num_\phi, lpos_\phi) = update(\phi, num_\phi, lpos_\phi, i)$;
    $(\psi, num_\psi, lpos_\psi) = close(\psi, num_\psi, lpos_\psi, i)$;
  }
  k-$WST(T)$ を出力する.
end.

図 3: k 単語幅制約付き接尾辞木の構築アルゴリズム

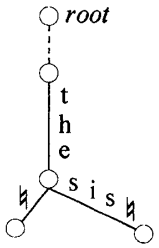

図 4: 仮想的な終端記号

補題 2. 葉ノードを閉じる処理は全体で $O(n)$ 時間で行える.

証明. 全ての時点 i について，$T[i]$ が入力されると，処理の最後に ψ は $child(\psi, T[i])$ に移動するので，ψ は全体で $O(n)$ 回移動する．このとき，ψ の移動と，num_ψ および $lpos_\psi$ の更新作業は共に定数時間である．また，ψ が $suf(\psi)$ へ移動するのは，現在の ψ を閉じるときである．葉ノードを閉じる作業と $num_\psi, lpos_\psi$ の更新は定数時間で行うことができる．葉ノードの個数は $O(n)$ 個であるから，合計では $O(n)$ 時間で実行できる． □

3.3 出現頻度の管理

接尾辞木では，T の任意の部分文字列の出現頻度を求めることが可能である．接尾辞木 $ST(T)$ の各葉ノード v は長さの異なる $|T|$ 個の接尾辞と一対一で対応しているので，それぞれの接尾辞 $\overline{v}$ の出現頻度は 1 である．また，内部ノード u の表す文字列 $\overline{u}$ は，その子の表す文字列の接頭辞である．したがって，u の任意の子 u' について $\overline{u'}$ が T 中に n 回現れたならば，$\overline{u}$ も同じ開始位置で n 回出現する．よって，$\overline{u}$ の出現頻度は u のすべての子の出現頻度の和で求められる．

しかし，k 単語幅限定接尾辞木 k-$WST(T)$ においては，閉じた葉ノード v は必ずしも接尾辞を表さないため，$\overline{v}$ は T 中に再び現れる場合があり，その出現頻度は 1 とは限らない．そこで，葉ノード $v \in V$ に対して頻度情報 $f(v)$ を付加する．$\overline{v}$ が T 中に再び現れたとき，接尾辞木では作られていた葉ノードの分として，$f(v)$ を 1 カウントする．あとは接尾辞木の場合と同様に，k-$WST(T)$ を深さ優先で探索することで再帰的に計算できる．

4. 実験結果

テストデータとして，情報検索用のデータである Reuters-21578 を用いた．ここでは元のデータから SGML タグを取り除き，英文のみを連結した約 18.8MB のテキストデータを用意した．このデータに対し，$k = 1, 2, \ldots, 6$ および通常の接尾辞木を構築し，先頭からの入力文字列長に対する木のノード数の変化を測定した．結果を図 5 に示す．k の値が小さい場合，接尾辞木に比べてきわめてノード数が少なくなることがわかる．

5. おわりに

本稿では，文字列の k 単語以下の部分文字列を効率よく表す k 単語幅制限付き接尾辞木を提案し，その構築が $O(n)$ でできることを示した．また，実際にアルゴリズムを実装し実験したところ，k が小さいほど大幅にノード数が抑制されることが分かった．実験結果では，単語幅の制約を 3 にした場合のノード数は接尾辞木の約半分であった．これは，同じメモリで約 2 倍のテキストに対する索引を構築できることを意味する．

単語の先頭から始まる接尾辞のみを木に追加する手法 [1, 4] が提案されている．それらを組み合わせることでさらにノードを省略できるかは，今後の課題である．

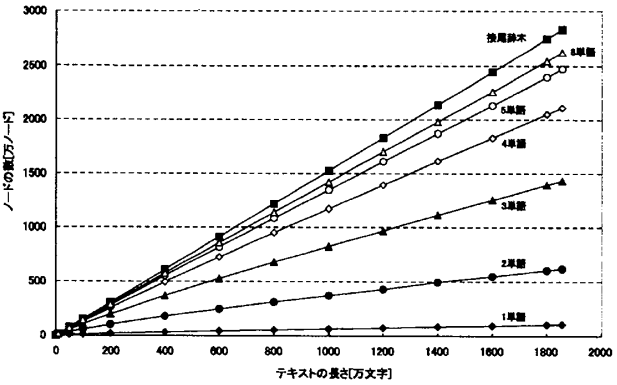

図 5: 入力テキスト長に対するノード数の変化

参考文献

[1] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[2] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[3] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Comm. ACM*, 32(4):490–505, 1989.

[4] S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *In proc. of 17th Ann. Symp. on Combinatorial Pattern Matching*, 2006. (to appear).

[5] N. J. Larsson. Extended application of suffix trees to data compression. In *Proc. of Data Compression Conference*, pp. 190–199, 1996.

[6] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[7] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304:87–101, 2003.

[8] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, 1981.

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