

Indexing factors in DNA/RNA Sequences [★]

Tomáš Flouri^{1,3}, Costas Iliopoulos^{2,4}, M. Sohel Rahman^{2,4}, Ladislav Vagner^{1,3},
and Michal Voráček^{1,3}

¹ Department of Computer Science & Engineering
Czech Technical University in Prague
Czech Republic

² Algorithm Design Group
Department of Computer Science
King's College London

Strand, London WC2R 2LS, England

³ {flourt1, xvagner, voracem}@fel.cvut.cz

⁴ {csi, sohel}@dcs.kcl.ac.uk

Abstract. In this paper, we present the Truncated Generalized Suffix Automaton (*TGSA*) and present an efficient on-line algorithm for its construction. *TGSA* is a novel type of finite automaton suitable for indexing DNA and RNA sequences, where the text is degenerate i.e. contains sets of characters. *TGSA* indexes the so called *k*-factors, the factors of the degenerate text with length not exceeding a given constant *k*. The presented algorithm works in $\mathcal{O}(n^2)$ time, where *n* is the length of the input DNA/RNA sequence. The resulting *TGSA* has at most linear number of states with respect to the length of the text. *TGSA* enables us to find the list $occ(u)$ of all occurrences of a given pattern *u* in degenerate text \tilde{x} in time $|u| + |occ(u)|$.

1 Introduction

Degenerate strings are special strings having a set of symbols instead of one at any particular position, e.g. $\tilde{x} = [a][c, g][g][a, c, t][a, c]$. A normal (non-degenerate) string can be seen as a special kind of degenerate string with singleton set at each position. Degenerate strings are extensively used in molecular biology to express polymorphisms in DNA/RNA sequences, e.g. the polymorphism of protein coding regions caused by redundancy of the genetic code or polymorphism in binding site sequences of a family of genes. Another reason to use degenerate strings as a model for the DNA/RNA sequences is the limited quality of automatically-obtained sequences caused by the intrinsic limitations of the equipments in the biology labs. In practice, the sets of nucleotides of DNA sequences are represented by single symbols of the extended IUB/IUPAC alphabet, where there exists a special symbol for each possible combination of nucleotides. For example, using IUB/IUPAC alphabet, \tilde{x} is represented as $\tilde{x} = ASGHM$.

[★] Supported by the Ministry of Education, Youth and Sports under research program MSM 6840770014 and the Czech Science Foundation as project No. 201/06/1039.

Indexing of short factors is a widely used and useful technique in stringology and bioinformatics. Use of k -factors (factors of length k) can be seen in solving diverse text algorithmic problems ranging from different string matching tasks [6] to motif finding [7] and alignment problems [8]. One can further mention the use of k -factors in FASTA [12] and BLAST [11]. In order to efficiently use the k -factors we need an efficient data structure to index them. It may be noted here that, in the literature, there exists several recent works on indexing different kinds of k -factors [10, 9].

An index over a fixed text \tilde{x} can be defined as an abstract data type based on the set of all factors of \tilde{x} , denoted by $Fact(\tilde{x})$. Such data type is equipped with some operations that allow it to answer the following 2 queries. Firstly, given an arbitrary string u , an index can answer the question whether $u \in Fact(\tilde{x})$ (the existence query). Secondly, if $u \in Fact(\tilde{x})$, then it can find the list of all occurrences of u in \tilde{x} . In the case of exact string matching in *normal* string, there exist classical data structures for indexing, such as suffix trees, suffix arrays, DAWGs.

In [1], the *Generalized Factor Automaton (GFA)* was presented, which, to the best of our knowledge, is the first data structure serving as a full index of a given degenerate string. The precise number of states of *GFA* is not yet known, but, experiments showed that it tends to grow super-quadratically with respect to the length of the degenerate string [1]. As a result, it is not suitable for indexing very long texts. Later, in [3], the *Truncated Generalized Factor Automaton (TGFA)* was presented along with an algorithm for its construction based on the so-called “partial determinization”. Essentially, *TGFA* is a modification of *GFA* that indexes only factors with length not exceeding a given constant k and it has at most linear number of states. The algorithm in [3] is based on the classical subset construction technique [5] and it inherits its space and time complexity. The space complexity is a bottleneck of this algorithm when indexing very long text since we need to determinize the corresponding large NFA. As a result, we lose the advantage of the much reduced size of *TGFA* due to its indirect construction from *GFA* using the partial determinization process and thereby limiting its applicability only for texts not exceeding KBs in length in practice.

In this paper, we present an on-line algorithm to construct *Truncated Generalized Suffix Automaton (TGSA)* ¹. This algorithm is space and time efficient, and, therefore, it enables us to construct *TGSA* for degenerate strings of length of tens of MBs. Notably, in [2], algorithms based on *GSA*¹ for searching regularities in degenerate strings were presented. These algorithms can be easily adapted for *TGSA* [4].

The rest of the paper is organized as follows. In Section 2, we present the notations and definitions. In Section 3, we present our main results. In particular, we present the algorithms for the construction of *GSA* and *TGSA*. Finally, we briefly conclude in Section 4. Due to the space constraints, some proofs, results

of some experiments and an example of *TGSA* construction are presented in the appendices.

2 Preliminaries

An *alphabet* Σ is a nonempty finite set of symbols. A *string* over a given alphabet is a finite sequence of symbols. The *empty string* is denoted by ε . The set of all strings over an alphabet Σ (including empty string ε) is denoted by Σ^* . A string $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$ is said to be *degenerate*, if it is built over the potential $2^{|\Sigma|} - 1$ non-empty sets of letters belonging to Σ . We say that $\tilde{x}_i \in \{\mathcal{P}(\Sigma) \setminus \{\emptyset\}\} = \mathcal{P}^+(\Sigma)$ and $|\tilde{x}_i|$ denotes the cardinality of \tilde{x}_i . In what follows, the set containing the letters a and c will be denoted by $[a, c]$ and the singleton $[c]$ will be simply denoted by c for the ease of reading. Also, we use the following convention: we use normal letters like x to denote normal strings. The same letter may be used to denote a degenerate string if used in the following way: \tilde{x} .

The *Language represented by degenerate string* \tilde{x} is the set $\mathcal{L}(\tilde{x}) = \{u \mid u = u_1u_2\dots u_n, u_j \in \tilde{x}_j, 1 \leq j \leq n, u \in \Sigma^*\}$. A string u is said to be an *element* of degenerate string \tilde{x} , denoted $u \in \tilde{x}$, if it is an element of language represented by \tilde{x} .

We define the concatenation operation on the set of degenerate strings in the usual way: if \tilde{x} and \tilde{y} are degenerate strings over Σ , then the concatenation of these strings is $\tilde{x}\tilde{y}$. The length $|\tilde{x}|$ of degenerate string \tilde{x} is the number of its sets (of letters). Note that, there may exist singleton sets. A string u is a *factor* (resp. *suffix*) of a degenerate string \tilde{x} if $\tilde{x} = \tilde{u}\tilde{v}\tilde{w}$ and $u \in \tilde{v}$ (resp. $u \in \tilde{w}$) and the set of all factors (resp. suffixes) of \tilde{x} is denoted $Fact(\tilde{x})$ (resp. $Suff(\tilde{x})$).

A string u is a *k-factor* of \tilde{x} if $u \in Fact(\tilde{x})$ and $|u| = k$ and the set of all *k-factors* of \tilde{x} is denoted $Fact_k(\tilde{x})$. A string v is a *at-most-k-factor* of \tilde{x} if $v \in Fact(\tilde{x})$ and $|v| \leq k$ and the set of all at-most-k-factors of \tilde{x} is denoted $Fact_k^-(\tilde{x})$. Similarly, we define *k-suffix*, *at-most-k-suffix*, $Fact_k(\tilde{x})$ and $Fact_k^-(\tilde{x})$.

A (normal) string $u = u_1\dots u_\ell$ of length ℓ is said to *occur* in a degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$ at position $i, 1 \leq i \leq n$ if and only if $u_j \in \tilde{x}_{i-\ell+j}$ for all $1 \leq j \leq \ell$. The list of all occurrences of u in \tilde{x} is denoted by $occ_{\tilde{x}}(u)$.

Since our algorithms are based on a finite automata approach, we give brief definitions to related concepts below. A nondeterministic finite automaton M is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where: Q is a finite set of states, Σ is an input alphabet, δ is a mapping $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$ called a state transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of final states. A deterministic finite automaton M is a special case of nondeterministic finite automaton such that transition mapping is a function $\delta : Q \times \Sigma \mapsto Q$ and there is only one initial state $q_0 \in Q$. The number of states (resp. transitions) of M we denote by $|M|_Q$ (resp. $|M|_\delta$). The *left language* of state q of finite automaton M , denoted $\underline{\mathcal{L}}_M(q)$, is a set of strings for which there exists a sequence of transitions

¹ Note that *TGFA* and *TGSA* resp. (*GFA* and *GSA*) has the same transition diagram (i.e. number of states) and they differ only in sets of the final states.

from the initial state to state q . The *left language* of finite automaton M , denoted $\overleftarrow{\mathcal{L}}_M$, is a union of left languages of all its states. *Language accepted* by finite automaton M , denoted $\mathcal{L}(M)$, is a set of words for which there exists sequence of transitions from the initial state to some of the final states. The *depth* of state q of acyclic finite automaton M is the length of the shortest path from the initial state to q . d -subset D of a state p of DFA M from NFA M' is the set of states of M' such that for each $q \in D$ it holds $\overleftarrow{\mathcal{L}}_{M'}(q) = \overleftarrow{\mathcal{L}}_M(p)$. A trie $TrieS$ constructed for set of string S is deterministic finite automaton accepting S having the transition diagram in the form of a rooted $|\Sigma|$ -ary tree. Note that a trie for S has maximum possible number of states among all deterministic finite automata accepting S .

We conclude this section by presenting the following example.

Example 1. Suppose we are given a degenerate string $\tilde{x} = a[a, c][a, c]ca[a, c]a[a, c]ca$. Then, according to definition, $Fact_2(\tilde{x}) = \{\varepsilon, a, c, ac, ca, cc, aa\}$, $Fact_2(\tilde{x}) = \{ac, ca, cc, aa\}$ and $occ_{\tilde{x}}(ca) = \{3, 5, 7, 10\}$.

3 Truncated Generalized Factor Automaton

In this section, we first introduce the concept of the *TGSA*. Next, we present an on-line algorithm for the construction of *GSA* followed by an on-line algorithm for the construction of *TGSA*. Notably, the *TGSA* construction algorithm is an extension of the *GSA* construction algorithm.

3.1 Generalized Suffix Automaton

Definition 1 (Generalized Suffix Automaton). *Given a degenerate string \tilde{x} , Generalized Suffix Automaton $GSA(\tilde{x})$ is a minimal deterministic finite automaton accepting the set $Suff(\tilde{x})$.*

GSA can be constructed easily by constructing a nondeterministic *GSA* and then determinizing it [1]. The nondeterministic *GSA* of $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$, has $n + 1$ states, q_0, q_1, \dots, q_n . For each symbol $s \in \tilde{x}_i, 1 \leq i \leq n$, there exists one transition each from q_0 and q_{i-1} to the state q_i . The only final state is q_n .

If we use natural labelling of the states (a number of a state corresponds to a position in the text) then elements of a d -subset of a state corresponds to end-positions of factors from the left language of that state. Thus, *GSA* can be used as a full index of a degenerate string. If, during the determinization of nondeterministic *GSA*, we stop the expansion of states in depth equal to a given constant k (partial determinization, [3]), then we obtain automaton accepting at least the set of all at-most- k suffixes. The resulting automaton is *TGSA*. More formally, *TGSA* can be defined as follows:

Definition 2 (Truncated Generalized Suffix Automaton). *Suppose we are given a degenerate string \tilde{x} and a positive integer k . Assume that $GSA(\tilde{x}) = M = (Q, \Sigma, \delta, q_0, F)$. Then, the Truncated Generalized Suffix Automaton for \tilde{x}*

and k is a deterministic finite automaton $TGSA_k(\tilde{x}) = M_T = (Q_T, \Sigma_T, \delta, q_0, F_T)$, where $Q_T \subseteq Q$, $F_T \subseteq F$ and $\delta_T \subseteq \delta$, such that it holds:

1. $Fact_k^-(\tilde{x}) \subseteq \overleftarrow{\mathcal{L}}(M_T) \subseteq Fact(\tilde{x})$,
2. $Suff_k^-(\tilde{x}) \subseteq \mathcal{L}(M_T) \subseteq Suff(\tilde{x})$,
3. $|M_T|_Q \leq |Trie(Fact_k^-(\tilde{x}))|_Q$,
4. for M_T , the value $|\overleftarrow{\mathcal{L}}(M_T) \setminus Fact_k^-(\tilde{x})|$ is minimal among all automata satisfying Conditions 1–3.

Remark 1. The number of states of $Trie(Fact_k^-(\tilde{x}))$ is given by the size of set $Fact_k^-(\tilde{x})$. The maximum number of all strings over an alphabet Σ of length at most k is limited by value of $\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$. This implies that the number of states of $Trie(Fact_k^-(\tilde{x}))$ and hence the size of $TGSA_k(\tilde{x})$ never exceeds this value regardless the size of \tilde{x} . Next, the number of not-unique factors of length at most k grows in linear manner with respect to length of \tilde{x} and hence the number of unique factors and hence the size of $TGSA_k(\tilde{x})$ cannot grow faster till it reaches the mentioned value $\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$.

Since the construction of $TGSA$ using partial determinization is costly, we, in this paper, develop a new space and time efficient online algorithm to construct $TGSA$ directly. The algorithm is presented in the following sections.

3.2 On-line construction of generalized suffix automaton

In this section we present the online construction algorithm for GSA . In the subsequent section, we show how to extend this algorithm to construct $TGSA$.

The algorithm reads the degenerate string from left to right, one symbol-set at a time. After processing each symbol set \tilde{x}_i ($1 \leq i \leq n$), a suffix automaton $M_i = (Q, \Sigma, \delta, q_0, F)$ accepting the set $Suff(\tilde{x}[1, i])$, is constructed. Each suffix $u \in Suff(\tilde{x}[1, i+1]) \setminus \{\varepsilon\}$ can be written in the form $u = vx$ where $v \in Suff(\tilde{x}[1, i])$ and $x \in \tilde{x}_{i+1}$. This means that the construction of suffix automaton $M_{i+1} = (Q', \Sigma, \delta, q_0, F')$, accepting the set $Suff(\tilde{x}[1, i+1])$, resides on traversing the set of final states F of suffix automaton M_i . Since the language $\overleftarrow{\mathcal{L}}_{M_i}(p)$, accepted by states $p \in F$, is the set $Suff(\tilde{x}[1, i])$, we can construct suffix automaton M_{i+1} , accepting the set $Suff(\tilde{x}[1, i+1])$, by creating new transitions $\delta(p, x) = p'$, where p' is a newly created state ($p' \notin Q \wedge p' \in F'$) and $x \in \tilde{x}_{i+1}$. All newly created transitions lead to the same state p' which has a d -subset $\{i+1\}$ (it indicates the position in the text where the given suffix ends). In case a transition $\delta(p, x) = w$ already exists, it means that the automaton M_i already has a transition $\delta^i(q_0, u) = w$, for some $u \in Suff(\tilde{x}[1, i+1])$ and $w \in Q$. In other words, u was already indexed in M_i as a factor of degenerate string $\tilde{x}[1, i]$. In this case, it is necessary to check whether all $v \in \overleftarrow{\mathcal{L}}_{M_i}(w)$ belong to the set $Suff(\tilde{x}[1, i+1])$. Considering that $v = ux$, where $x \in \tilde{x}_{i+1}$, if all u are part of the set $Suff(\tilde{x}[1, i])$, then all v are part of the set $Suff(\tilde{x}[1, i+1])$ and thus, it is only needed to extend the d -subset of w with the element $\{i+1\}$,

```

procedure: CONSTRUCT
1 begin
2    $Q \leftarrow \emptyset$ 
3    $F \leftarrow \text{NEW-LIST}()$ 
4    $q_0 \leftarrow \text{NEW-STATE}(\{0\})$ 
5    $Q \leftarrow Q \cup \{q_0\}$ 
6    $\text{ADD}(F, q_0)$ 
7   for  $i \leftarrow 1$  to  $n$  do
8      $p' \leftarrow \text{NEW-STATE}(\{i\})$ 
9      $F \leftarrow \text{UPDATE}()$ 
10     $Q \leftarrow Q \cup \{p'\}$ 
11     $M_i \leftarrow (Q, \Sigma, \delta, q_0, F)$ 
12  end for
13 end

```

Algorithm 1: On-line construction of a deterministic suffix automaton accepting a degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2 \dots \tilde{x}_n$.

```

procedure: UPDATE
1 begin
2    $\text{ADD}(F', p')$ 
3   while not  $\text{EMPTY}(F)$  do
4      $p \leftarrow \text{DEQUEUE}(F)$  traverse all final states
5     for each  $x \in \tilde{x}_i$  do
6       if  $\exists \delta(p, x)$  then
7          $\text{EXTEND}()$ 
8       else
9          $\delta(p, x) \leftarrow p'$ 
10      end if
11    end for
12  end while
13   $\text{ADD}(F', q_0)$ 
14  return  $F'$ 
15 end

```

Algorithm 2: Procedure *update* traverses the states in the suffix path and updates their transitions.

which denotes the new found position in the text. In order to check whether all u are part of $\text{Suff}(\tilde{x}[1, i])$, it is necessary to traverse all final states of M_i , which actually represent the set $\text{Suff}(\tilde{x}[1, i])$ and count the x -transitions leading to state w . If the counted number is equal to the number of incoming transitions of state w , then all v are part of $\text{Suff}(\tilde{x}[1, i + 1])$. If there exists at least one $u \notin \text{Suff}(\tilde{x}[1, i])$ (in other words, there exists one incoming transition of state w , which does not originate from a final state of M_i), then the language $\overleftarrow{\mathcal{L}}_{M_i}(w)$ is formed by factors (not only suffixes) of $\tilde{x}[1, i + 1]$. Thus, it is required to “split” the state in two states, one representing the factors which are not suffixes and

one representing the suffixes. This is carried out by “cloning” state w (we will denote the new state as $clone(w)$) and changing all transitions $\delta(p, x) = w$, where $p \in F$, to $\delta(p, x) = clone(w)$. By the term “cloning” of state w , we mean the creation of a new state $clone(w)$ having the same d -subset as w and the creation of transitions $\delta(clone(w), a) = \delta(w, a)$ for all $a \in \Sigma$. The complete procedure is formally presented in Algorithms 1-3.

```

procedure: EXTEND
1 begin
2    $w \leftarrow \delta(p, x)$ 
3    $in \leftarrow \text{INCOMING}(w)$            number of incoming transitions of w
4    $out \leftarrow \text{OUTGOING-SP}(w)$     transitions from states in F leading to w
5   if  $in = out$  then
6      $w \leftarrow w \cup \{i\}$            if equal extend the d-subset
7      $\text{ADD}(F', w)$                      add w to final states of  $M_{i+1}$ 
8   else
9      $s \leftarrow \text{NEW-STATE}(w \cup \{i\})$  if not equal, split the state
10    for each  $q \in F$  and  $\delta(q, x) = w$  do  $\delta(q, x) \leftarrow s$ 
11    for each  $a \in \Sigma$  do  $\delta(s, a) \leftarrow \delta(w, a)$ 
12     $\text{ADD}(F', s)$                      add s to final states of  $M_{i+1}$ 
13  end if
14 end

```

Algorithm 3: Procedure *extend* adds states to the new suffix path by extending the d -subsets of existing states or creating new ones.

```

procedure: EXTEND-2
1 begin
2    $w \leftarrow \delta(p, x)$ 
3   if  $\nexists clone(w)$  or  $clone(w) \cap \{i\} = \emptyset$  then
4      $clone(w) \leftarrow \text{NEW-STATE}(w \cup \{i\})$ 
5     for each  $a \in \Sigma$  do  $\delta(clone(w), a) \leftarrow \delta(w, a)$ 
6      $\text{ADD}(F', clone(w))$ 
7   end if
8    $\delta(p, x) \leftarrow clone(w)$ 
9   if  $\text{INCOMING}(w) = 0$  then  $\text{REMOVE}(w)$ 
10 end

```

Algorithm 4: Optimized EXTEND procedure.

In Algorithm 4, we present a simpler version of Algorithm 3. In this case, instead of counting x -transitions and then deciding whether to split state w ,

the split is performed always. While traversing the rest of the final states, if a transition is found to lead to w , it is redirected to *clone* (w). In case w is left with no incoming transitions, it is removed. Notably, usage of Algorithm 3 or 4 does not change the asymptotic time complexity of the algorithm.

Lemma 1. *Given a degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$, Algorithm 1 correctly constructs $GSA(\tilde{x})$ in time $\mathcal{O}(n^2\sigma\varphi)$, where φ is the number of states of the resulting automaton and σ is the size of used alphabet.*

A sketch of the proof is provided in Appendix B

3.3 On-line construction of truncated generalized suffix automaton

As before, the algorithm for constructing a *TGSA* reads the degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$ from left to right, one symbol set at a time. After processing each symbol set \tilde{x}_i ($1 \leq i \leq n$), a *TGSA* $M_i = (Q, \Sigma, \delta, q_0, F)$ accepting at least all k -suffixes of $\tilde{x}[1, i]$ is created. The *TGSA* M_{i+1} is constructed in a similar fashion as presented in Section 3.2 (Algorithms 1-4). The main difference is that instead of only traversing the final states of M_i and creating or updating transitions where necessary, it is required to traverse all states of M_i in a *breadth-first-search* (BFS) manner. This is because when transforming M_i to M_{i+1} , certain states of M_i can change depth in the new M_{i+1} to a value higher than k , which (states) must be eliminated. The BFS is carried out in order to mark all visited states that have a depth less than k in the resulting M_{i+1} . For each final state of M_i , having a depth less than k , the same steps described in Section 3.2 (Algorithms 1-4) are applied. All states having a depth greater than k are removed, resulting in a new M_{i+1} having at most as many states as $TrieSuff_k(\tilde{x})$. In step i , to check whether a given state is a final state of M_{i-1} , we need only check whether element $i-1$ is part of its d -subset. This can be done in constant time if the d -subset is implemented as a linked-list of arrays and storing the size of each state's d -subset. Element $i-1$ will then be either on the last position or one position before the last one since the d -subset is always sorted (this is a consequence of reading the degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$ from left to right). The whole process is presented in Algorithms 5-10.

Lemma 2. *Given a degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$ and positive integer k , Algorithm 5 correctly constructs $TGSA_{\tilde{x}}(k)$ in $\mathcal{O}(n^2\sigma^{k+2})$ time, where σ is the size of used alphabet.*

A sketch of the proof is provided in Appendix B.

Corollary 1. *Given a degenerate DNA/RNA sequence $\tilde{x} = \tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$ and a positive integer k ($n \gg k$), Algorithm 5 correctly constructs $TGSA_{\tilde{x}}(k)$ in $\mathcal{O}(n^2)$ time.*

Proof. This follows immediately because σ is constant for DNA/RNA sequences.

Remark 2. We remark that in practical setting we are interested in indexing strings of size in 10's of MBs and the patterns involved are typically small and hence the assumption that $n \gg k$ is a valid assumption.

```

procedure: CONSTRUCT
input:  $\tilde{x} = \tilde{x}_1\tilde{x}_2 \dots \tilde{x}_n$  - a degenerate string over alphabet  $\Sigma$ ,  $k$  - maximum
length of factors to index
output:  $M \leftarrow (Q, \Sigma, \delta, q_0, F)$  - a TGSA over  $\tilde{x} = \tilde{x}_1\tilde{x}_2 \dots \tilde{x}_n$ 

1 begin
2    $Q \leftarrow \emptyset$ 
3    $q_0 \leftarrow \text{NEW-STATE}(\{0\})$ 
4    $Q \leftarrow Q \cup \{q_0\}$ 
5    $D \leftarrow \text{NEW-QUEUE}()$ 
6   for  $i \leftarrow 1$  to  $n$  do
7      $F \leftarrow \{q_0\}$ 
8      $p' \leftarrow \text{NEW-STATE}(\{i\})$ 
9      $Q \leftarrow Q \cup \{p'\}$ 
10     $\text{ENQUEUE}(D, q_0)$ 
11     $\text{status}(q_0) \leftarrow \text{OPEN}$ 
12     $\text{used} \leftarrow \text{false}$ 
13    while not  $\text{EMPTY}(D)$  do
14       $p \leftarrow \text{DEQUEUE}(D)$ 
15      if  $\text{level}(p) < k$  then
16        if  $i - 1 \in p$  or  $p = q_0$  then  $\text{UPDATE}()$ 
17         $\text{ENQUEUE-SUCCESSORS}(D, p, p')$ 
18      else
19         $\text{ELIMINATE-REDUNDANT}(p)$ 
20      end if
21    end while
22    if  $\text{used} = \text{false}$  then  $\text{REMOVE}(p')$            Remove  $p'$  if not used
23     $M_i \leftarrow (Q, \Sigma, \delta, q_0, F)$ 
24    for each  $q \in Q$  do  $\text{status}(q) \leftarrow \text{FRESH}$ 
25  end for
26 end

```

Algorithm 5: On-line construction of a *TGSA* accepting all k -suffixes of a degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2 \dots \tilde{x}_n$.

Existence Query As is mentioned before, *TGSA* has at most linear number of states. However, with each state there is an associated list namely d -subset. Since each such list can have upto n items, the space requirements becomes higher. But if we need only answer the existence queries, we don't need the d -subset lists and in that case we can get a very efficient version of *TGSA*.

4 Conclusion

In this paper, we have presented efficient on-line algorithm to construct *Truncated Generalized Suffix Automaton (TGSA)*, a novel type of finite automaton serving as a k -factor-index for a degenerate strings. The *TGSA* algorithm works in $\mathcal{O}(n^2)$ time for DNA/RNA sequences and the resulting automaton has at

```

procedure: UPDATE()
1 begin
2   for each  $x \in \tilde{x}_i$  do
3     if  $\exists \delta(p, x)$  then
4       EXTEND( $\delta(p, x), p, x$ )
5     else
6        $\delta(p, x) \leftarrow p'$ 
7        $level(p') \leftarrow \text{MIN}(level(p'), level(p) + 1)$ 
8       if  $used = false$  then  $F \leftarrow F \cup \{p'\}$ 
9        $used \leftarrow true$  p' was used
10    end if
11  end for
12 end

```

Algorithm 6: As in Algorithm 2, procedure *update* updates the appropriate transitions of final states of automaton M_{i-1} .

```

procedure: EXTEND( $w, p, x$ )
input:  $w$  - destination state,  $p$  - source state,  $x$  - transition symbol
1 begin
2   if  $\nexists clone(w)$  or  $clone(w) \cap \{i\} = \emptyset$  then
3      $clone(w) \leftarrow \text{NEW-STATE}(w \cup \{i\})$ 
4      $status(clone(w)) \leftarrow FRESH$ 
5     for each  $a \in \Sigma$  do  $\delta(clone(w), a) \leftarrow \delta(w, a)$ 
6      $F \leftarrow F \cup \{clone(w)\}$ 
7   end if
8    $\delta(p, x) \leftarrow clone(w)$ 
9    $level(clone(w)) \leftarrow \text{MIN}(level(clone(w)), level(p) + 1)$ 
10  if INCOMING( $w$ ) = 0 then REMOVE( $w$ )
11 end

```

Algorithm 7: Optimized EXTEND procedure.

```

procedure: ELIMINATE-REDUNDANT( $p$ )
input:  $p$  - the state whose non-visited successors are to be eliminated
1 begin
2   for each  $u \leftarrow \delta(p, a), a \in \Sigma$  do
3     if  $status(u) = FRESH$  then
4        $\delta(p, a) \leftarrow nil$ 
5       if INCOMING( $u$ ) = 0 then REMOVE( $u$ )
6     end if
7   end for
8 end

```

Algorithm 8: Elimination of redundant states (states with a $level > k$).

```

procedure: ENQUEUE-SUCCESSORS( $D, p, p'$ )
input:  $D$  - BFS Queue,  $p$  - state whose successors are to be enqueued,  $p'$  -
        new state
1 begin
2   for each  $\delta(p, a) \neq p', a \in \Sigma$  do
3     if  $status(\delta(p, a)) = FRESH$  then
4       ENQUEUE( $D, \delta(p, a)$ )
5        $level(\delta(p, a)) \leftarrow level(p) + 1$ 
6        $status(\delta(p, a)) \leftarrow OPEN$ 
7     end if
8   end for
9 end

```

Algorithm 9: Part of the *breadth-first-search* algorithm which enqueues all successors of a state.

```

procedure: REMOVE( $p$ )
input:  $p$  - the state to remove
1 begin
2   for each  $u \leftarrow \delta(p, a), a \in \Sigma$  do
3     if  $status(u) = FRESH$  then
4        $\delta(p, a) \leftarrow nil$ 
5       if  $INCOMING(u) = 0$  then REMOVE( $u$ )
6     end if
7   end for
8    $Q \leftarrow Q \setminus \{p\}$ 
9   DELETE-STATE( $p$ )
10 end

```

Algorithm 10: Removal of state p and all its, unreachable from other states, successors.

most linear number of states. The already known algorithms for searching regularities in degenerate strings designed originally for *GSA* can be easily modified to work with *TGSA* and hence, *TGSA* can be used for searching regularities in degenerate strings as well.

References

1. M. Voráček, B. Melichar, M. Christodoulakis, *Generalized and weighted strings: Repetitions and pattern matching*, In String Algorithmics, 225–248, KCL Publications, King’s College London, (2004).
2. M. Voráček, B. Melichar. *Searching for regularities in generalized strings using finite automata*, In Proceedings of International Conference on Numerical Analysis and Applied Mathematics, Willey-VCH, (2005).

3. M. Voráček, V. Vagner, T. Flouri *Indexing Degenerate Strings*, In Proceedings of International Conference on Computational Methods in Science and Engineering, Melville, New York: American Institute of Physics, 2007.
4. T. Flouri, *Indexing Degenerate Strings*, Master Thesis, Czech Technical University in Prague, 2008.
5. Hopcroft J.E., Ullman J. D., *Introduction to automata, languages and computation* Addison-Wesley, Reading, MA (1979)
6. G. Navarro and E. Sutinen and J. Tanninen and J.,Tarhio: *Indexing Text with Approximate q-Grams*, Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, Montréal, Canada, LNCS 1848, 350–363, Springer-Verlag, Berlin(2000)
7. C. S. Iliopoulos and J. McHugh and P. Peterlongo and N. Pisanti and W. Rytter and M. Sagot *A First Approach to Finding Common Motifs with Gaps*, International Journal of Foundations of Computer Science, (2004).
8. B. Ma, J. Tromp, and M. Li., *Patternhunter: faster and more sensitive homology search*, Bioinformatics, 18(3):440–445, March (2002).
9. C. S. Iliopoulos and M. Sohel Rahman. *Indexing factors with gaps*. Algorithmica. To Appear (DOI: 10.1007/s00453-007-9141-3).
10. P. Peterlongo, J. Allali and M.-F. Sagot. *The gapped-factor tree*. In: J. Holub and J. Zdrek (eds.) Stringology, pp. 182-196. Czech Technical University, Prague (2006).
11. S.F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, *Basic local alignment search tool.*, J. Mol. Biol. 215:403–410, (1990)
12. D .J. Lipman , W.R. Pearson (1988) *Improved Tools for Biological Sequence Comparison*, PNAS 85: 2444–2448 (1998)

A Experiments

As we have presented in Section 3.1, the number of states of $TGSA_{\bar{x}}(k)$ can grow in the worst case in linear manner but it will never exceeds the value $\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$. We performed a series of experiments to investigate the size of a $TGSA$ in a practical manner.

The tests were conducted on degenerate strings with uniform distribution of symbol sets. Due to hardware limitations the degenerate strings were comprised by the following symbol sets. 84% are symbol sets with cardinality 1, 12% with cardinality 2, 3% with cardinality 3 and 1% with cardinality 4. For a given text size and truncation level k , we randomly generated 50 different degenerate strings and constructed a deterministic $TGSA$ for it. For each text size and truncation level, the number of states was summed up and the average number of states from the 50 instances was calculated. The experiment was performed on degenerate strings with sizes from 100 000 to 1 000 000 symbol sets with a step of 20 000 symbol sets.

Figure A depicts the graph showing the results of the experiment. From the graph it is obvious that the size of the $TGSA$ grows in a linear fashion in respect to the text size, while keeping the distribution and the percentage of cardinalities of symbol sets unchanged. This is actually expected since by keeping the same distribution and extending the text size, the number of new, unique factors grows. As we extend the text, the number of unique factors gradually falls.

At a certain text size, the automaton stops growing as can be seen in Figure A, for $k = 10$. This is due to the fact that at some point, all possible factors of length up to k are already in the text. Therefore, at that point, the automaton has indexed all possible k -factors. It has the form of a complete $|\Sigma|$ -ary prefix tree and only extends the d -subsets of its states.

B Proofs

Proof (Lemma 1). The loop in procedure *construct* (Algorithm 1) requires n cycles for reading the input data from left to right. φ steps are required for traversing the suffix path at each of the n cycles of *construct*. Procedure *extend* can then be called at most $\sigma\varphi$ times. Using the optimized *extend* procedure we need at most n steps for copying the d -subset of a state during the process of cloning. Checking whether a cloned state was created in the i -th cycle ($clone(w) \cap \{i\}$) can be done in constant time since the d -subset can be implemented as a list, where access to the last element is immediate.

Assuming $n \gg |\Sigma|$ and that the function $incoming(w)$, returning the number of transitions leading to w , is of constant complexity (since it can be implemented as a counter for each state being increased (resp. decreased) when a transition leading to w is created (resp. deleted)), the asymptotic time complexity of the presented algorithm is $\mathcal{O}(n^2\sigma\varphi)$.

Proof (Lemma 2). A $TGSA$, for a given k , has at most (when it takes the form of a complete $|\Sigma|$ -ary prefix tree) $\sum_{i=0}^k |\Sigma|^i$ states, which is always less than $2|\Sigma|^k$,

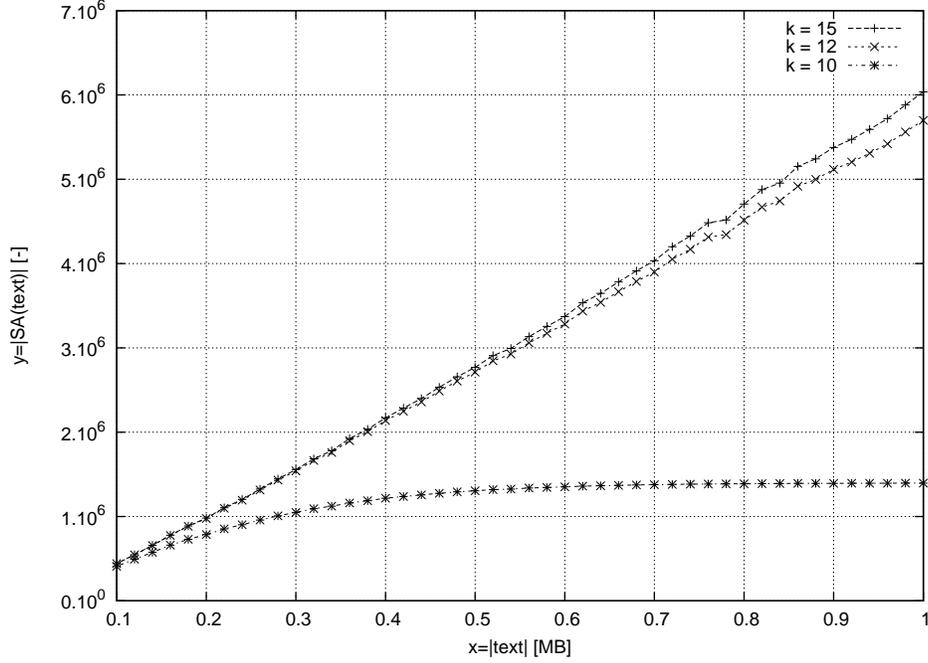


Fig. 1. Average size of *TGSA* with respect to the length of the preprocessed degenerate string and the level k of truncation

assuming that $|\Sigma| > 1$. Asymptotically, *TGSA* has at most $\mathcal{O}(|\Sigma|^k)$ (or k , in the case of $|\Sigma| = 1$) states. We deduce the complexity of the algorithm as follows (we assume that $|\Sigma| > 1$). The loop of procedure *construct* requires n cycles for reading each symbol set. During each cycle, all states with a level less or equal to k must be traversed in the *breadth-first-search* (BFS) way. This requires $\mathcal{O}(n|\Sigma|^k)$ steps. Procedure *update*, which is called from *construct*, calls at most σ times procedure *extend* which, in its turn, requires at most n steps for copying the d -subset of a state during the process of cloning. Assuming that automaton M_i constructed in step i has φ states, the maximum theoretical number of new states that can be created when constructing M_{i+1} is $|\Sigma|\varphi$, resulting in a total of at most $\mathcal{O}(n|\Sigma|^{k+1})$ states. Procedure *remove* recursively removes all states that were not visited during BFS (states with level greater than k). Thus, the total time complexity is $\mathcal{O}(n^2|\Sigma|^{k+1}|\Sigma|) = \mathcal{O}(n^2|\Sigma|^{k+2})$. As all states with level greater than k are removed at each step, the resulting automaton has at most as many states as $\text{TrieSuff}_k(\tilde{x})$ and thus, fulfills Definition 2.

C Examples

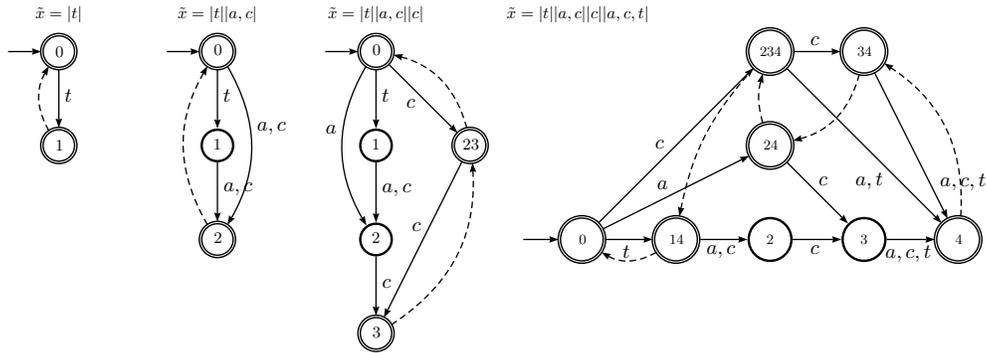


Fig. 2. The first 4 steps of the on-line construction of a SA accepting degenerate string $\tilde{x} = [t][a, c][c][a, c, t] \dots$. Transitions in dashed style of each automaton M_i denote the states that need to be traversed for creating the automaton M_{i+1} .

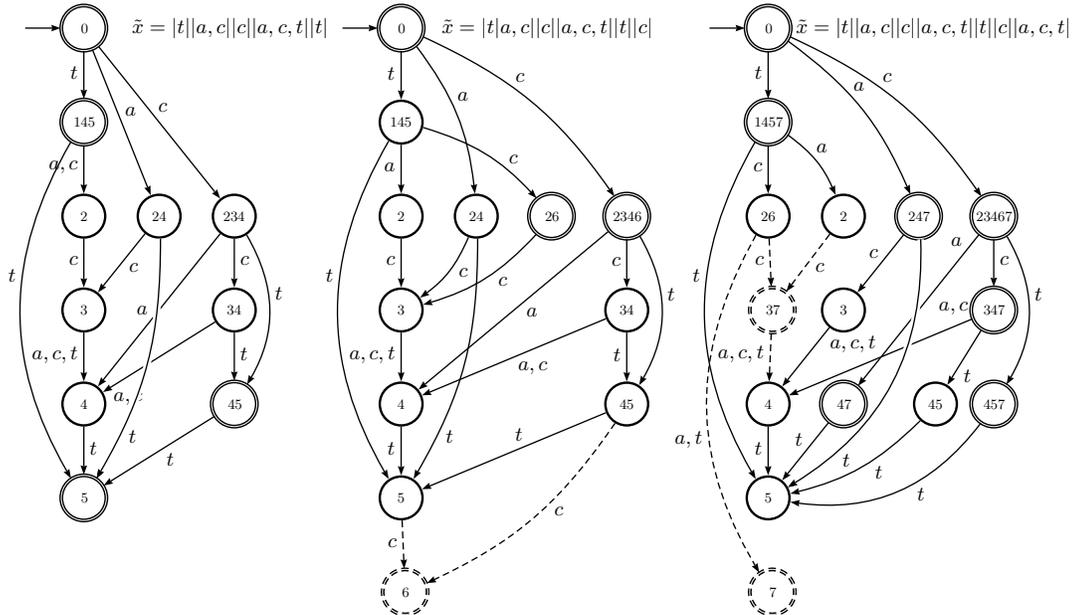


Fig. 3. Steps 5-7 of the on-line construction of TGSA ($k = 2$) for degenerate string $\tilde{x} = [t][a, c][c][a, c, t][t][c][a, c, t]$. Dashed states along with dashed transitions are to be deleted at the end of each step since they have a depth higher than k . The first 4 steps of the algorithm are the same as Figure 2.