Directed Acyclic Subsequence Graph[1]

Zdeněk Troníček and Bořivoj Melichar

Department of Computer Science and Engineering
Czech Technical University
Karlovo náměstí 13, 121 35 Prague 2, Czech Republic
phone: ++420 2 2435 7287, fax: ++420 2 298098

e-mail: {tronicek,melichar}@fel.cvut.cz

Abstract. Directed Acyclic Subsequence Graph is an automaton, which accepts all subsequences of the given string. We introduce a left-to-right algorithm for incremental construction of DASG. The algorithm requires $\mathcal{O}(z)$ extra space and $\mathcal{O}(nz \log z)$ time for arbitrary alphabet ($\mathcal{O}(nz)$ for fixed alphabet), where $z = min(|\Sigma|, n)$. The number of transitions can be reduced by encoding input symbols using k digits, where $k < min(|\Sigma|, n)$. We introduce a left-to-right algorithm for incremental construction of DASG for $k = 2$. We show the extension of the algorithm for the set of strings and its application for the longest common subsequence problem.

Key words: Directed Acyclic Subsequence Graph, finite automaton, searching subsequences

1 Introduction

A subsequence of a string is any string obtained by deleteing zero or more symbols from the given string. Directed Acyclic Subsequenc Graph (DASG) is an automaton, which accepts all subsequences of the given text. It was introduced in [2] (preliminary version was published in [1]). DASG is analogous to Directed Acyclic Word Graph (DAWG) [3] using subsequences instead of substrings.

Let us suppose an alphabet Σ and a text $T = t_1 t_2 \dots t_n$ over this alphabet. DASG for the text T is an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an input alphabet, δ is a transition function, q_0 is the initial state and F is a set of final states. States are denoted by numbers in this article.

In [2], there is described a right-to-left algorithm for construction of DASG and encoding for reducing the number of transitions. In section 3 we introduce an incremental left-to-right algorithm for construction of DASG, and in section 4 its modification for encoded DASG. In section 5 we show the extension of the algorithm for a set of strings and its application for the longest common subsequence problem.

[1]This research has been supported by GAČR grant No. 201/98/1155

2 Motivation

Let $Sub(T)$ denotes the set of all subsequences of the text $T = t_1 t_2 \dots t_n$. The set $Sub(T)$ can be described recursively by the regular expression (ε is empty subsequence):
$Sub_0 = \varepsilon$
$Sub_i = Sub_{i-1}(\varepsilon + t_i)$
$Sub(T) = Sub_n$
For the set Sub_n then holds:

$$Sub_n = Sub_{n-1}(\varepsilon + t_n) = Sub_{n-2}(\varepsilon + t_{n-1})(\varepsilon + t_n) = (\varepsilon + t_1)(\varepsilon + t_2)\dots(\varepsilon + t_n)$$

This expression gives us the direction for construction of the nondeterministic version of DASG. The example of such nondeterministic finite automaton (NFA) is in Fig. 1. It also holds:

$$Sub_n = Sub_{n-1} + Sub_{n-1}t_n = \varepsilon + Sub_0 t_1 + Sub_1 t_2 + \dots + Sub_{n-2}t_{n-1} + Sub_{n-1}t_n$$

The last expression can be used for construction of the nondeterministic DASG without ε-transitions (the example is in Fig. 2). If all the symbols of T are different, we obtain directly the deterministic finite automaton (DFA). The example of deterministic DASG is in Fig. 3.

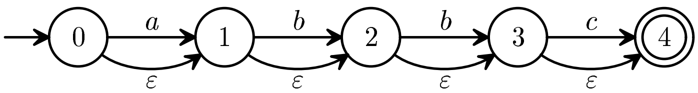


Figure 1: NFA accepting all subsequences of $T = abbc$ (with ε-transitions).

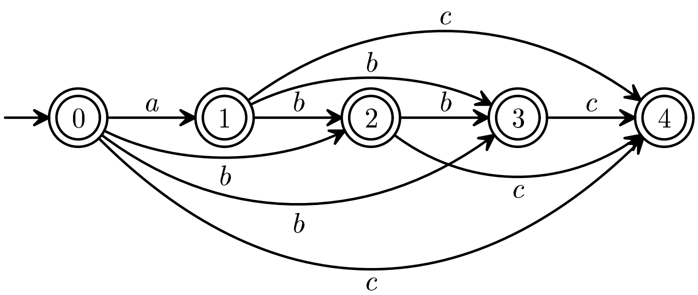


Figure 2: NFA accepting all subsequences of $T = abbc$ (without ε-transitions).

3 Construction of DASG

Let us suppose an alphabet Σ and a text $T = t_1 t_2 \dots t_n$ over this alphabet. For each symbol a of the alphabet Σ we will maintain the value f_a, which is the smallest number of the state not having an output transition labeled with a. We start with an automaton with the only state 0. Each state of the automaton is final.

Lemma 1: The automaton constructed by Algorithm 1 has $(n + 1)$ states.

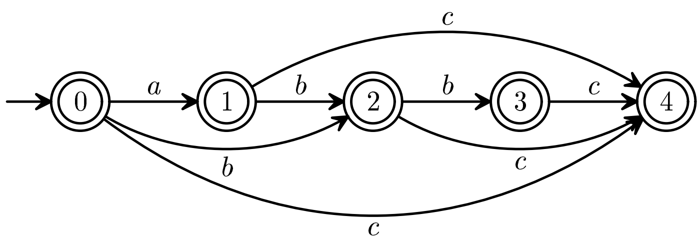


Figure 3: DASG for the string $T = abbc$.

for each $a \in \Sigma$ **do**
  $f_a \leftarrow 0$
end for
for $k \leftarrow 1$ to n **do**
  add state k
  for $s \leftarrow f_{t_k}$ to $(k-1)$ **do**
    add a transition labeled t_k from the state s to the state k
  end for
  $f_{t_k} \leftarrow k$
end for

Figure 4: Algorithm 1 (incremental construction of DASG)

Proof: We start with the automaton with one state. The main cycle of the algorithm is performed n times. During each step of the cycle we add just one new state.

Lemma 2: The automaton constructed by Algorithm 1 accepts just all subsequences of T.

Proof: We prove the lemma in two steps.
1. If S is a subsequence of the string T then S is accepted by the automaton (induction by the length of S):
Step 1: $|S| = 1, S = s_1$. If s_1 occurs in T then state 0 of the automaton has transition labeled with s_1 and the automaton accepts S.
Step 2: A string $S_k = s_1 s_2 \dots s_k$ is a subsequence of T and is accepted by the automaton. Let us create a new string $S_{k+1} = s_1 s_2 \dots s_k s_{k+1}$ by adding a symbol s_{k+1} to the end of S_k. There exists a sequence $i_1, i_2, \dots, i_k$ such that $s_1 = t_{i_1}, s_2 = t_{i_2}, \dots, s_k = t_{i_k}$ (the automaton will finish in state i_k after accepting S_k). If there exists i_{k+1} such that $i_k < i_{k+1} \leq n$ and $s_{k+1} = t_{i_{k+1}}$, then state i_k has transition labeled with s_{k+1} and the automaton accepts S_{k+1}.
2. If S is accepted by the automaton then S is a subsequence of T (induction by the length of S):
Step 1: $|S| = 1, S = s_1$. If S is accepted by the automaton then state 0 has the transition labeled with s_1. State 0 has transition labeled with s_1 only if there exists $j, 1 \leq j \leq n$ such that $s_1 = t_j$.
Step 2: A string $S_k = s_1 s_2 \dots s_k$ is accepted by the automaton and there exists a sequence $i_1, i_2, \dots, i_k$ such that $s_1 = t_{i_1}, s_2 = t_{i_2}, \dots, s_k = t_{i_k}$. We create a new string $S_{k+1} = s_1 s_2 \dots s_k s_{k+1}$ by adding a symbol s_{k+1} to the end of S_k. The automaton will finish in state i_k after accepting S_k. If state i_k has transition labeled s_{k+1} then there exists $i_{k+1}, i_k < i_{k+1} \leq n$ such that $s_{k+1} = t_{i_{k+1}}$. $\triangle$

3.1 Number of transitions

Definition 1: Let Σ be an alphabet and $T = t_1t_2 \dots t_n$ a string over this alphabet. Let Σ_e denotes the set of all symbols, which are contained in T. We define the effective size of Σ as $z = |\Sigma_e|$.

The minimum number of transitions is n (if and only if all the symbols of T are the same).
For each state k, the maximum number of its output transitions is:

$$max\_out\_deg_k = min(z, n-k)$$

It results from that the first $(n+1-z)$ states may have at most z output transitions and for the last z states the maximum number of output transitions decreases to 0. Then, the maximum total number of transitions is:

$$(n+1-z)z + (z-1) + \dots + 2 + 1 + 0 = (n+1-z)z + \frac{z(z-1)}{2} = \frac{2zn + z - z^2}{2}$$

DASG has this number of transitions if and only if the last z symbols of T are different.

3.2 Complexity

The main cycle of the Algorithm 1 (lines 4–10) is performed n times. The lines 5 and 9 take constant time. At lines 6–8 are added all the transitions. Therefore, the total time complexity of lines 6–8 is $\mathcal{O}(nz)$.

For fixed alphabet we suppose that adding or looking up the transition takes constant time. Then, the algorithm requires time $\mathcal{O}(n + nz)$ in the worst case. The time of subsequence test is $\mathcal{O}(m)$ in the worst case.

For arbitrary alphabet we suppose that adding or looking up the transition takes time $\mathcal{O}(\log z)$. Then, all the complexities must be multiplied by factor $\log z$. In this case, we use a balanced tree for values $f_a, a \in \Sigma$.

In both cases the algorithm requires $\mathcal{O}(z)$ extra space.

4 Encoding of input symbols

Encoding as a method for reducing the number of transitions was introduced in [2]. The method use $k < z$ digits for encoding the input symbols, where z is the effective size of the alphabet. The number of states grows at most to $n\lceil \log_k z \rceil + 1$, but the number of transitions usually decreases (see [2] for details). Fig. 5 shows the encoded version of DASG for the text $T = abbc$ (in this case encoding does not reduce the number of transitions). The symbols are coded this way: $a = 00$, $b = 01$, $c = 10$.

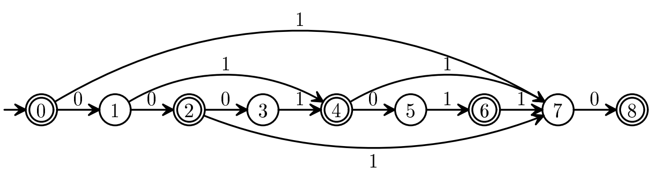


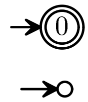
Figure 5: Encoded DASG for the string $T = abbc$.

Let us suppose an alphabet Σ and a text $T = t_1 t_2 \dots t_n$ over this alphabet. Each symbol a of Σ we encode using digits 0 and 1. For that we need at least $c = \lceil \log_2 z \rceil$ digits. The algorithm is incremental. When we add a new symbol encoded as $e_1 e_2 \dots e_c$, we need to ensure, that all previous final states have an output path labeled with $e_1 e_2 \dots e_c$. We use a binary tree as an auxiliary structure. The tree is built during the construction of the automaton. Each inner node of the tree has two lists (for 0 and for 1), which contents states, where ends the path labeled with the same symbols as the path in the tree, starting at any final state and is the longest possible. So, if a state s is in the list l_e in the node with the path $p_1 p_2 \dots p_q$ from the root, there exists a path in the DASG from any final state to state s, which is labeled $p_1 p_2 \dots p_q$ and state s has no output transition labeled e. We start with an automaton with the only state 0. States (tc) for $t = 0, 1, \dots, n$ are final. At the beginning, the tree has only the initial node.

$l_0^\varepsilon \leftarrow \{0\}$, $l_1^\varepsilon \leftarrow \{0\}$
for $k \leftarrow 0$ to $(n-1)$ **do**
  encode the symbol $t_{(k+1)}$ as $e_1 e_2 \dots e_c$
  set the root as the actual node in the tree
  for $b \leftarrow 1$ to c **do**
    add state $(ck + b)$
    for each state s in the list l_{e_b} in the actual node of the tree **do**
      add a transition labeled e_b between states s and $(ck + b)$
      remove s from the list l_{e_b}
    end for
    go to the child of the actual node of the tree through the transition labeled e_b and set the child as the actual node (if the child does not exist, create it and set both lists of new node empty)
    if $b < c$ **then**
      add the state $(ck + b)$ to the both lists in the actual node
    end if
  end for
  mark the state $(ck + c)$ as a final state and add it to the both lists in the root of the tree
end for

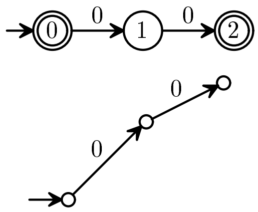
Figure 6: Algorithm 2 (incremental construction of encoded DASG)

The algorithm is demonstrated in Fig. 7–11. The lists maintained in the node with the path p from the root are denoted as l_0^p and l_1^p, the symbols are coded this way: $a = 00$, $b = 01$, $c = 10$.



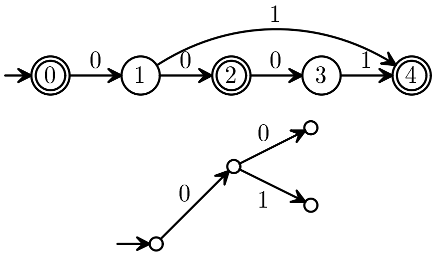
$l_0^\varepsilon = \{0\}$, $l_1^\varepsilon = \{0\}$

Figure 7: Encoded DASG for the string $T = \varepsilon$.



$l_0^\varepsilon = \{2\}$, $l_1^\varepsilon = \{0, 2\}$
$l_0^0 = \{\}$, $l_1^0 = \{1\}$

Figure 8: Encoded DASG for the string $T = a$.



$l_0^\varepsilon = \{4\}$, $l_1^\varepsilon = \{0, 2, 4\}$
$l_0^0 = \{3\}$, $l_1^0 = \{\}$

Figure 9: Encoded DASG for the string $T = ab$.

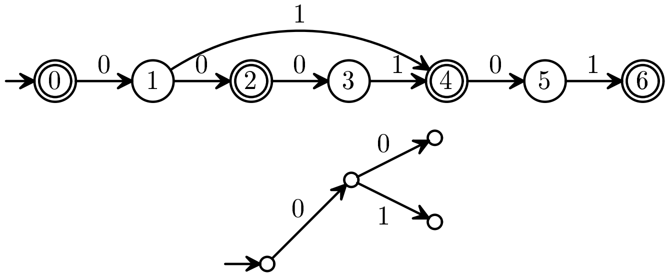
4.1 Number of transitions

The number of states grows to $1 + nc = 1 + n\lceil \log_2 n \rceil$. The maximum number of transitions is $c(2n - \frac{1}{2}(c + 1)) = \lceil \log_2 z \rceil (2n - \frac{1}{2}(\lceil \log_2 z \rceil + 1))$.

4.2 Complexity

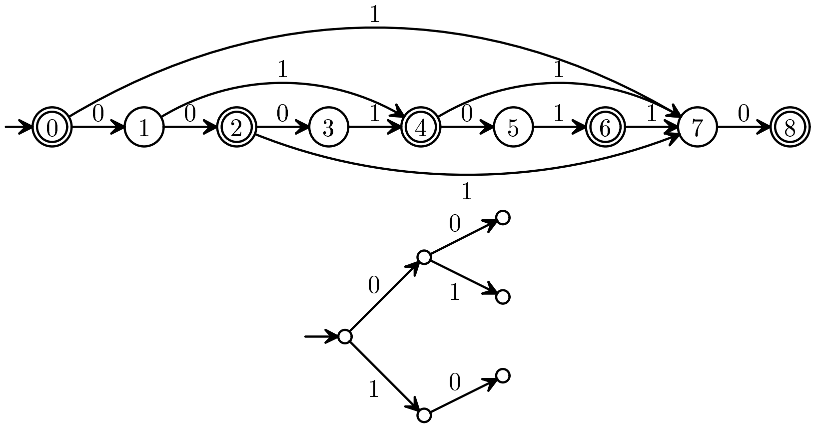
The main cycle (line 2–15) is performed n times. Lines 1,3,4,6,11,12,13 and 16 require constant time. The cycle on line 5 is performed $\mathcal{O}(\log_2 z)$ times. The total number of transitions is $\mathcal{O}(n \log_2 z)$. Therefore, the total time complexity of lines 7–10 is $\mathcal{O}(n \log_2 z)$. Hence, the total time complexity is $\mathcal{O}(n \log_2 z)$.

The algorithm needs $\mathcal{O}(z + n \log_2 z)$ extra space for the tree and for the lists in its nodes. The subsequence test requires $\mathcal{O}(m \log_2 z)$ time in the worst case.



$l_0^\varepsilon = \{6\}$, $l_1^\varepsilon = \{0, 2, 4, 6\}$
$l_0^0 = \{3, 5\}$, $l_1^0 = \{\}$

Figure 10: Encoded DASG for the string $T = abb$.



$l_0^\varepsilon = \{6,8\}$, $l_1^\varepsilon = \{8\}$
$l_0^0 = \{3,5\}$, $l_1^0 = \{\}$
$l_0^1 = \{\}$, $l_1^1 = \{7\}$

Figure 11: Encoded DASG for the string $T = abbc$.

5 DASG for a set of strings

Let us suppose an alphabet Σ and strings $T_1, T_2, \ldots, T_w$ over this alphabet. We extend Algorithm 1 to a set strings $\{T_1, T_2, \ldots, T_w\}$. Let $L = \sum_{i=1}^{w} length(T_i)$.

The construction of DASG has two steps:

- Construction of inverted trie for reversed strings.
- Construction of an automaton.

Construction of inverted trie: Inverted trie arises from trie by reversing the transitions and can be constructed during the construction of trie (each node of trie will have one inverted transition). Final nodes of trie are initial nodes of inverted trie. Inverted trie is used as an auxiliary structure and served for finding common suffixes of the strings.

Construction of an automaton: For each string $T_i, 1 \leq i \leq w$ we will maintain:

- lists $l_a^i, a \in \Sigma$ of states, which have no output transition for the symbol a
- actual position act_i in inverted trie
- actual position $last_i$ in the automaton

In each node of inverted trie we save the number of corresponding state in the automaton. Let v denotes this number, let γ denotes the transition function in trie, and let δ denotes the transition function in the automaton. For each transition of the automaton we have to remember, which strings it belongs to. This set is denoted E. We start with the automaton with the only state 0. Each state of the automaton is final. Set is ordered set with two defined operation: $first$ return the first string in the set and $next$ return the successor of the string. The total number of states after each step is in the variable $states$.

The algorithm is demonstrated in Fig. 15–18.

```
for i ← 1 to w do
   for each a ∈ Σ do
      l_a^i ← {0}
   end for
   act_i ← final state for the string T_i in trie
   last_i ← 0
end for
for each node i in trie do
   v(i) ← 0
end for
Set ← {T_1, T_2, ..., T_w}
states ← 1
c ← 1
p ← 1
for k ← 1 to L do
   M ← ∅
   symbol ← p-th symbol of T_c
   act_c ← γ(act_c, symbol)
   if δ(last_c, symbol) ≠ ∅ then
      new_state ← δ(last_c, symbol)
   else if v(act_c) > 0 then
      new_state ← v(act_c)
   else
      add state states
      new_state ← states
      v(act_c) ← states
      states ← states + 1
   end if
   last_c ← new_state
   for each s ∈ l^c_symbol do
      if δ(s, symbol) ≠ ∅ then
         M ← M ∪ {δ(s, symbol)}
         E(s, symbol) ← E(s, symbol) ∪ {c}
      else
         δ(s, symbol) ← new_state
         E(s, symbol) ← {c}
      end if
   end for
   l^c_symbol ← ∅
   M ← M ∪ {new_state}
   for each a ∈ A do
      l_a^c ← l_a^c ∪ M
   end for
   if p = length(T_c) then
      Set ← Set \ {T_c}
   end if
   if next(Set, c) is defined then
      d is defined as follows: next(Set, c) = T_d
   else
      d is defined as follows: first(Set, c) = T_d
      p ← p + 1
   end if
   c ← d
end for
```

Figure 12: Algorithm 3 (extension of DASG for a set of strings $\{T_1, T_2, \ldots, T_w\}$)

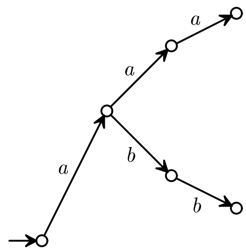


Figure 13: Trie for the reversed strings aaa and bba.

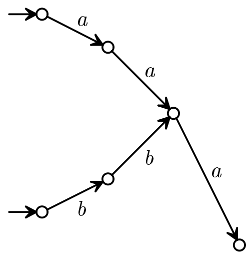
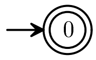
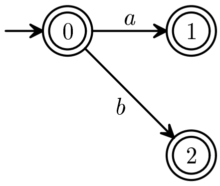


Figure 14: Inverted trie for the reversed strings aaa and bba.



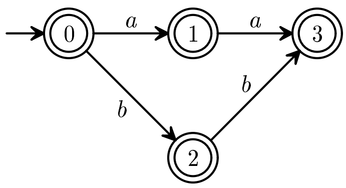
$l_a^1 = \{0\}$, $l_b^1 = \{0\}$
$l_a^2 = \{0\}$, $l_b^2 = \{0\}$

Figure 15: Extension of DASG for the $Set = \{\varepsilon\}$.



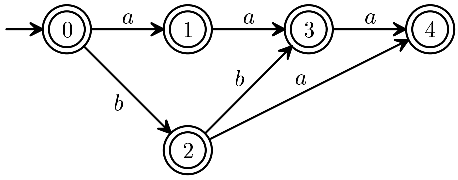
$l_a^1 = \{1\}$, $l_b^1 = \{0, 1\}$
$l_a^2 = \{0, 2\}$, $l_b^2 = \{2\}$
$E(0, a) = \{1\}$, $E(0, b) = \{2\}$

Figure 16: Extension of DASG for the $Set = \{a, b\}$.



$l_a^1 = \{3\}$, $l_b^1 = \{0, 1, 3\}$
$l_a^2 = \{0, 2, 3\}$, $l_b^2 = \{3\}$
$E(0, a) = \{1\}$, $E(0, b) = \{2\}$
$E(1, a) = \{1\}$, $E(2, b) = \{2\}$

Figure 17: Extension of DASG for the $Set = \{aa, bb\}$.



$l_a^1 = \{4\}$, $l_b^1 = \{0, 1, 3, 4\}$
$l_a^2 = \{1, 4\}$, $l_b^2 = \{1, 3, 4\}$
$E(0, a) = \{1\}$, $E(0, b) = \{2\}$
$E(1, a) = \{1\}$, $E(2, b) = \{2\}$
$E(3, a) = \{1\}$, $E(2, a) = \{2\}$

Figure 18: Extension of DASG for the $Set = \{aaa, bba\}$.

5.1 Number of states

For each symbol of the string, except for the last, a new state can be added. Hence, the DASG constructed in Algorithm 3 has at most $1+\sum_{i=1}^{w}(length(T_i)-1)+1 = 2+L-w$ states. DASG has this number of states if no two strings have any common nonempty prefix and suffix.

Each state can have at most z output transitions. Therefore, the total number of transitions is $\mathcal{O}(Lz)$.

5.2 Complexity

Construction of inverted trie requires $\mathcal{O}(L)$ time and $\mathcal{O}(L)$ extra space. For the time analysis of the Algorithm 3 is important the time complexity of set operations. We use four of them: *insert a member*, *delete a member*, *assign a value* and *union*. Suppose, that we use a balanced tree for the representation of the set M (another possibilities are a member function or a list). Then, the operations *assign* and *union* require $\mathcal{O}(|M|)$ time and the other operations require $\mathcal{O}(\log |M|)$ time.

Lines 1–7 require $\mathcal{O}(wz)$ time, lines 8–10 require $\mathcal{O}(L)$ time, line 11 requires $\mathcal{O}(w)$ time. The main cycle (lines 15–53) is performed L times. Line 16 requires $\mathcal{O}(L)$ time, and lines 18–20 require $\mathcal{O}(\log z)$ time. The cycle on the lines 30–38 is performed at most L times. Its time complexity is $\mathcal{O}(L \log L)$ (line 32 requires $\mathcal{O}(\log L)$ time). Line 40 requires $\mathcal{O}(\log L)$ time, lines 41–43 require $\mathcal{O}(Lz)$ time. Hence, the total time complexity is $\mathcal{O}(L^2 + L^2 \log L + L \log L + L^2 z) \approx \mathcal{O}(L^2 \log L)$ for arbitrary alphabet.

We need $\mathcal{O}(L)$ space for trie, $\mathcal{O}(L)$ space for the set M, and $\mathcal{O}(Lwz)$ space for the lists l_a^c. Hence, the total required extra space is $\mathcal{O}(Lwz)$.

5.3 Application: the longest common subsequence problem

The longest common subsequence (LCS) problem is known problem with applications in many areas. There are efficient algorithms that solve the LCS problem for two strings, for example [4].

Let us define the following problem (as in [2]): What is the longest common subsequence between any $k \leq w$ strings in a set S of w strings?

To solve this problem, we construct DASG for the set S and append to each transition $\delta(q, a)$ the number of strings in the set E (denoted as $num(q, a)$) and to each state q the number of its input edges (denoted as c_q) and the number of its input edges with $num(q,a)$ greater or equal k (denoted as ck_q). We do not need the set E in this case. Then, we traverse DASG. During the traversing we use LIFO (Last-In-First-Out) memory as an auxiliary structure (denoted as $Stack$). Dot (.) denotes concatenation. The longest sequence of input symbols from the initial state to the state q is stored in cs_q.

```
lcs ← ε
for each state q do
    cs_q ← ε
end for
Stack ← 0
while Stack is not empty do
    q ← Pop
    if length(cs_q) > length(lcs) then
        lcs ← cs_q
    end if
    for each a ∈ Σ such that δ(q, a) ≠ ∅ do
        r ← δ(q, a)
        c_r ← c_r − 1
        if c_r = 0 then
            Push(r)
        end if
        if num(q, a) ≥ k then
            ck_r ← ck_r − 1
            if ck_r = 0 then
                cs_r ← cs_q.a
            end if
        end if
    end for
end while
```

Figure 19: Algorithm 4 (the longest common subsequence)

The traversion of DASG requires $\mathcal{O}(Lz)$ time. For common subsequences cs_q we need $\mathcal{O}(Ly)$ space, where $y = max\{length(T_i)\}$. Hence, the general longest common subsequence problem of w strings requires $\mathcal{O}(L^2 \log L + Lz)$ time for arbitrary alphabet. It is a better solution than presented in [2].

6 Conclusion

In section 3, we introduced a left-to-right algorithm for construction of DASG. It requires $\mathcal{O}(nz \log z)$ time and $\mathcal{O}(z)$ extra space for arbitrary alphabet. The subsequence test takes $\mathcal{O}(m \log z)$ time.

In section 4, we showed the modification of that algorithm for encoded DASG. The modified algorithm requires $\mathcal{O}(n \log z)$ time and $\mathcal{O}(z + n \log z)$ extra space. The subsequence test takes $\mathcal{O}(m \log z)$ time.

In section 5, we extended DASG for a set of strings and used it to solve the general longest common subsequence problem. Construction of DASG takes $\mathcal{O}(L^2 \log L)$ time and $\mathcal{O}(Lwz)$ extra space. The traversion of DASG requires $\mathcal{O}(Lz)$ time. Hence, the solution of the general longest common subsequence problem requires $\mathcal{O}(L^2 \log L)$ total time and $\mathcal{O}(Lwz + Ly)$ space, where $y = max\{length(T_i)\}$.

References

[1] Baeza-Yates, R. A.: The Subsequence Graph of a Text. TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Sotware Development, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'89), Lecture Notes in Computer Science 351, Barcelona, Spain, March 1989, pages 104–118.

[2] Baeza-Yates, R. A.: Searching subsequences. Theoretical Computer Science 78 (1991), pages 363–378.

[3] Crochemore, M., Rytter, W.: Text algorithms. Oxford University Press, 1994.

[4] Hunt, J., Szymanski, T.: A fast algorithm for computing longest common subsequences. Communication of ACM 20, 1977, pages 350–353.

[5] Hirschberg, D. S.: A linear space algorithm for computing maximal common subsequences. Communication of ACM, 18(6), 1975, pages 341–343.