Directed Acyclic Subsequence Graph\textsuperscript{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 $O(z)$ extra space and $O(nz \log z)$ time for arbitrary alphabet ($O(nz)$ 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 deleting zero or more symbols from the given string. Directed Acyclic Subsequence 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 $\Sigma$ and a text $T = t_1 t_2 \ldots t_n$ over this alphabet. DASG for the text $T$ is an automaton $A = (Q, \Sigma, \delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ is an input alphabet, $\delta$ 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.

\textsuperscript{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 $T = t_1t_2 \ldots t_n$. The set $Sub(T)$ can be described recursively by the regular expression ($\varepsilon$ 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:

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

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:

$Sub_n = Sub_{n-1} + Sub_{n-1}t_n = \varepsilon + Sub_0t_1 + Sub_1t_2 + \ldots + Sub_{n-2}t_{n-1} + Sub_{n-1}t_n$

The last expression can be used for construction of the nondeterministic DASG without $\varepsilon$-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 $\varepsilon$-transitions).](image1)

![Figure 2: NFA accepting all subsequences of $T = abbc$ (without $\varepsilon$-transitions).](image2)

3 Construction of DASG

Let us suppose an alphabet $\Sigma$ and a text $T = t_1t_2 \ldots t_n$ over this alphabet. For each symbol $a$ of the alphabet $\Sigma$ 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 $T = abbc$.

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 sub-sequences 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 $S_k = s_1s_2 \ldots s_k$ is a subsequence of $T$ and is accepted by the automaton. Let us create a new string $S_{k+1} = s_1s_2 \ldots s_k s_{k+1}$ by adding a symbol $s_{k+1}$ to the end of $S_k$. There exists a sequence $i_1, i_2, \ldots, i_k$ such that $s_1 = t_{i_1}, s_2 = t_{i_2}, \ldots, s_k = t_{i_k}$ (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 $S_k = s_1s_2 \ldots s_k$ is accepted by the automaton and there exists a sequence $i_1, i_2, \ldots, i_k$ such that $s_1 = t_{i_1}, s_2 = t_{i_2}, \ldots, s_k = t_{i_k}$. We create a new string $S_{k+1} = s_1s_2 \ldots s_k s_{k+1}$ 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 $\Sigma$ be an alphabet and $T = t_1 t_2 \ldots t_n$ a string over this alphabet. Let $\Sigma_e$ denotes the set of all symbols, which are contained in $T$. We define the effective size of $\Sigma$ 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:

$$\max_{\text{out-deg}}k = \min(z, n - k)$$

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:

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

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 $O(nz)$.

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

For arbitrary alphabet we suppose that adding or looking up the transition takes time $O(\log z)$. 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 $O(z)$ 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 $n[\log_z z] + 1$, 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 $T = abbc$.](image-url)
Let us suppose an alphabet $\Sigma$ and a text $T = t_1t_2\ldots t_n$ over this alphabet. Each symbol $a$ of $\Sigma$ we encode using digits 0 and 1. For that we need at least $c = \lfloor \log_2 z \rfloor$ digits. The algorithm is incremental. When we add a new symbol encoded as $e_1e_2\ldots e_c$, we need to ensure, that all previous final states have an output path labeled with $e_1e_2\ldots e_c$. 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 contains 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 $p_1p_2\ldots p_q$ from the root, there exists a path in the DASG from any final state to state $s$, which is labeled $p_1p_2\ldots p_q$ and state $s$ has no output transition labeled $e$. We start with an automaton with the only state 0. States $(tc)$ for $t = 0, 1, \ldots, n$ are final. At the beginning, the tree has only the initial node.

1: $l_0^e \leftarrow \{0\}, \ l_1^e \leftarrow \{0\}$
2: for $k \leftarrow 0$ to $(n - 1)$ do
3:   encode the symbol $t_{(k+1)}$ as $e_1e_2\ldots e_c$
4:   set the root as the actual node in the tree
5: for $b \leftarrow 1$ to $c$ do
6:   add state $(ek+b)$
7:   for each state $s$ in the list $l_{en}$ in the actual node of the tree do
8:     add a transition labeled $e_b$ between states $s$ and $(ek+b)$
9:     remove $s$ from the list $l_{en}$
10: end for
11: 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)
12: if $b < c$ then
13:   add the state $(ek+b)$ to the both lists in the actual node
14: end if
15: end for
16: mark the state $(ek+c)$ as a final state and add it to the both lists in the root of the tree
17: 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 $l_p^0$ and $l_p^1$, the symbols are coded this way: $a = 00$, $b = 01$, $c = 10$.

$\rightarrow 0$

$\rightarrow 0$

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

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

111
4.1 Number of transitions

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

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 \(O(\log_2 z)\) times. The total number of transitions is \(O(n \log_2 z)\). Therefore, the total time complexity of lines 7–10 is \(O(n \log_2 z)\). Hence, the total time complexity is \(O(n \log_2 z)\).

The algorithm needs \(O(z + n \log_2 z)\) extra space for the tree and for the lists in its nodes. The subsequence test requires \(O(m \log_2 z)\) time in the worst case.
5 DASG for a set of strings

Let us suppose an alphabet $\Sigma$ 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 $L = \sum_{i=1}^{w} \text{length}(T_i)$.

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 $\ell_i^a$, $a \in \Sigma$ 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 $\gamma$ denotes the transition function in trie, and let $\delta$ 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: $\text{first}$ return the first string in the set and $\text{next}$ return the successor of the string. The total number of states after each step is in the variable $\text{states}$.

The algorithm is demonstrated in Fig. 15–18.
for $i \gets 1$ to $w$ do
  for each $a \in \Sigma$ do
    $i_a \gets \{0\}$
  end for
  $act_i \gets$ final state for the string $T_i$ in trie
  $last_i \gets 0$
end for
for each node $i$ in trie do
  $v(i) \gets 0$
end for

Set $T_1, T_2, \ldots, T_w$

for $k \gets 1$ to $L$ do
  $M \gets \emptyset$
  $symbol \gets p$-th symbol of $T_c$
  $act_c \gets \gamma(act_c, symbol)$
  if $\delta(last_c, symbol) \neq \emptyset$ then
    new_state $\gets \delta(last_c, symbol)$
  else if $v(act_c) > 0$ then
    new_state $\gets v(act_c)$
  else
    add state states
    new_state $\gets$ states
    $v(act_c) \gets$ states
  end if
  $last_c \gets$ new_state
  for each $s \in \mathcal{E}_c$ do
    if $\delta(s, symbol) \neq \emptyset$ then
      $M \leftarrow M \cup \{\delta(s, symbol)\}$
    end if
  end for
  $\mathcal{E}(s, symbol) \leftarrow new_state$
  $\mathcal{E}(s, symbol) \leftarrow \{c\}$
end if
$\mathcal{E}_c \leftarrow \emptyset$
$M \leftarrow M \cup \{new_state\}$
for each $a \in A$ do
  $V_a \leftarrow V_a \cup M$
end for
if $p = \text{length}(T_c)$ then
  $Set \leftarrow Set \setminus \{T_i\}$
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$(Set, c) = T_d$
  $p \leftarrow p + 1$
end if
$c \leftarrow d$
end for

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

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

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

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

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

Figure 17: Extension of DASG for the Set = \{aa, bb\}.
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 $1 + \sum_{i=1}^{w}(\text{length}(T_i) - 1) + 1 = 2 + L - w$ 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 $O(Lz)$.

5.2 Complexity

Construction of inverted trie requires $O(L)$ time and $O(L)$ 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 $O(|M|)$ time and the other operations require $O(\log |M|)$ time.

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

We need $O(L)$ space for trie, $O(L)$ space for the set $M$, and $O(Lwz)$ space for the lists $l^c_i$. Hence, the total required extra space is $O(Lwz)$.

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 $\text{num}(q, a)$ 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 $cs_q$.

```plaintext
1: lcs ← $\varepsilon$
2: for each state $q$ do
3:   $cs_q$ ← $\varepsilon$
4: end for
5: Stack ← 0
6: while Stack is not empty do
7:   $q$ ← Pop
8:   if length($cs_q$) > length(lcs) then
9:     lcs ← $cs_q$
10: end if
11:   for each $a \in \Sigma$ such that $\delta(q, a) \neq \emptyset$ do
12:     $r$ ← $\delta(q, a)$
13:     $c_r$ ← $c_r$ − 1
14:     if $c_r$ = 0 then
15:       Push($r$)
16:     end if
17:     if $\text{num}(q, a) \geq k$ then
18:       $ck_r$ ← $ck_r$ − 1
19:       if $ck_r$ = 0 then
20:         $cs_r$ ← $cs_q, a$
21:       end if
22:     end if
23:   end for
24: end while
```

Figure 19: Algorithm 4 (the longest common subsequence)

The traversal of DASG requires $O(Lz)$ time. For common subsequences $cs_q$ we need $O(Ly)$ space, where $y = \max \{\text{length}(T_i)\}$. Hence, the general longest common subsequence problem of $w$ strings requires $O(L^2 \log L + Lz)$ 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 $O(nz \log z)$ time and $O(z)$ extra space for arbitrary alphabet. The subsequence test takes $O(m \log z)$ time.

In section 4, we showed the modification of that algorithm for encoded DASG. The modified algorithm requires $O(n \log z)$ time and $O(z + n \log z)$ extra space. The subsequence test takes $O(m \log z)$ 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 $O(L^2 \log L)$ time and $O(Lwz)$ extra space. The traversal of DASG requires $O(Lz)$ time. Hence, the
solution of the general longest common subsequence problem requires \( O(L^2 \log L) \) total time and \( O(Lw_z + Ly) \) space, where \( y = \max\{\text{length}(T_i)\} \).

References


