The Subsequence Graph of a Text

(Preliminary version)

Ricardo A. Baeza-Yates
Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1 *

Abstract

We define the directed acyclic subsequence graph of a text as the smallest deterministic partial finite automaton that recognizes all possible subsequences of that text. We define the size of the automaton as the size of the transition function and not the number of states. We show that it is possible to build this automaton using $O(n \log n)$ time and $O(n)$ space for a text of size n. With this structure, we can search a subsequence in logarithmic time. We extend this construction to the case of multiple strings obtaining a $O(n^2 \log n)$ time and $O(n^2)$ space algorithm, where n is the size of the set of strings. For the later case, we discuss its application to the longest common subsequence problem improving previous solutions.

1 Introduction

Given a text, a subsequence of that text is any string such that its symbols appear somewhere in the text in the same order. Subsequences arise in data processing and genetic applications, being the longest common subsequence problem (LCS) the most important problem. They are used in data processing to measure the differences between two files of data, and in genetic research to study the structure of long molecules (DNA).

The first interesting question to answer, is the membership problem. That is, if a given string is a subsequence of another string. This can be expressed as a regular expression (see [1] for the standard notation). For example, if the

*This work was supported by the Institute of Computer Research of the University of Waterloo and by the University of Chile.

subsequence is $x_1 x_2 \cdots x_r$, and t is the text, then the problem may be expressed as

$$t \in \theta^* x_1 \, \theta^* x_2 \, \theta^* \cdots \, \theta^* x_r \, \theta^* \; ?$$

where θ is the don't care symbol and $*$ the star operator or Kleene closure. Clearly, we can answer this question in linear time. However, we are interested in answer this question in optimal time, by allowing the text to be preprocessed.

A natural question is which is the size of the deterministic finite automaton that given a text, recognizes any possible subsequence of that text. We allow the automaton to be *partial*, that is, each state need not to have a transition on every symbol. As all the states of this automaton are accepting, it can be viewed as a directed acyclic graph, which we call the Directed Acyclic Subsequence Graph (DASG). This problem is analogous to the Directed Acyclic Word Graph (DAWG) in where we are interested in subsequences instead of subwords [3].

In section 2 we introduce the DASG, and in section 3 we show how to build it in $O(n \log n)$ time and space for arbitrary alphabets, and in $O(n \log |\Sigma|)$ time and space for finite alphabets, where Σ denotes the alphabet. With this structure, we can test membership in $O(|s| \log n)$ time for arbitrary alphabets and $O(|s|)$ time for finite alphabets, where s is the subsequence that we are testing. One interesting thing to point out is that the DAWG recognizes all possible $O(n^2)$ subwords using $O(n)$ space, while the DASG recognizes all possible 2^n subsequences using $O(n \log n)$ space. In section 4 we show that is possible to reduce the space required to $O(n)$, but having a $O(|s| \log n)$ searching time for any alphabet.

In section 5 we extend the DASG to the case of multiple strings and we use it to solve the longest common subsequence problem and variations of it [2]. Our algorithm improves upon previous solutions of this problem for more than two strings, running in time $O(n^2 \log n)$ using $O(n^2)$ space. Previous solutions to the general case used $O(n^L)$ time and space for L strings [7] by using dynamic programming, or $O(n^3)$ time and $O(n^2)$ space [6] using an approach similar to the one developed in this paper.

2 Building the DASG

We can define the DASG recursively in the size of the text. The DASG of a text of size n must recognize all possible subsequences of the last $n-1$ symbols of the text, and all possible subsequences that start with the first symbol. As a regular expression this is:

$$S_n = (\epsilon + t_1) S_{n-1} \quad \text{and} \quad S_0 = \epsilon$$

where ϵ is the empty word and $t = t_1 t_2 \cdots t_n$ is the text. The size of the regular expression S_n is linear on n, and so is the non-deterministic finite automaton equivalent to S_n. Suppose that all the symbols of the text are different. The "deterministic" version of S_n for this case is

$$S_n = \epsilon + t_1 S_{n-1} + t_2 S_{n-2} + \cdots + t_n S_0$$

Clearly, the size of S_n is $O(n^2)$. Figure 1 shows the DASG for the text *abcd.* This automaton has $n+1$ states (all of them are final states) and $n(n+1)/2$ transitions. The number of states is minimal because we have to recognize the complete text (the longest subsequence). The number of edges (given the minimal set of states) is also minimal, because in the position i of the text we have to recognize any subsequence starting with t_j for $j = i+1, ..., n$. It is not difficult to generalize this for the case of repeated symbols.

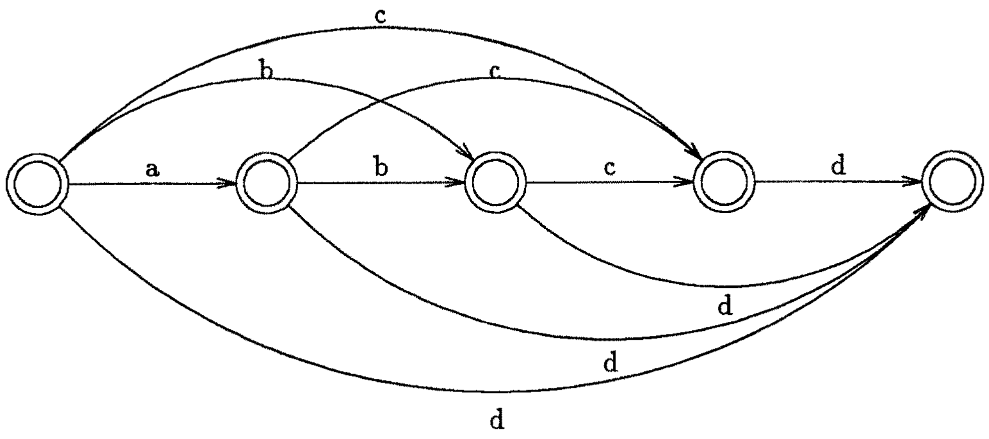

Figure 1: Minimal state DASG for the text *abcd.*

Definition: Let Σ be the alphabet. We define the effective size of Σ as $c = \min(|\Sigma|, n)$.

To build the DASG in $O(cn)$ time and space we use an incremental algorithm scanning the text from the *right to the left.* At each step we maintain a dictionary that contains all different symbols of the already scanned text, and the state in which the first skeleton transition labeled with that symbol appears. Hence, the algorithm is

1. Create state s_n and create an empty dictionary D.
2. For each symbol in the text t_i scanning from the right to the left do:
   (a) Create state s_{i-1}.

(b) Insert the pair $(t_i, i-1)$ in D. If t_i is already in D, its associated state is updated to $i-1$.

(c) For each symbol in D (d_j), append a transition labeled with d_j to state s_{k+1}, where k is the state associated to d_j in D.

Step (a) takes constant time. The insertion, step (b), can be performed in $\log c$ time, because the size of D is $O(c)$. For the same reason, step (c), the traversal of D takes $O(c)$ time. The cycle is performed n times. Then, the total time is $O(n(c + \log c))$. If we apply the same algorithm scanning the text from the left, we obtain the DASG of the reversed text. For this DASG, we can test the membership of a subsequence s using s reversed.

A membership query in this DASG takes $O(|s| \log c)$ time, where the $\log c$ term is the time to search for the appropriate transition in each state. Using a complete table for small alphabets, a $O(|s|)$ worst case time is achieved. For larger alphabets, we can obtain $O(|s|)$ average time by using hashing.

3 The Smallest Automaton

It is possible to reduce the time and space requirements? The answer is yes. The main problem is that the number of edges is $O(n^2)$ while the number of states is linear. Here we are not interested in the minimal set of states, we are interested in minimal space and that means a *minimal number of edges*. In other words, the *smallest transition function* for the automaton. To the best of our knowledge, this is first time that such concept is given.

Definition: The smallest deterministic partial finite automaton A that recognizes the regular language $L(r)$ defined by the regular expression r, is such that does not exist other automaton that recognizes $L(r)$ with less transitions than A.

We shall show that minimal number of states it is not, in general, equivalent to the smallest automaton. In [3] is claimed that the DAWG is the smallest automaton that recognizes all the subwords of a text. However, they show that is the smallest in the sense of minimal number of states. Intuitively, the DAWG may be the smallest automaton, because the number of states and the number of edges only differ in $n + O(1)$. In our problem, it is not the case, and we introduce a method that we call *encoding*, since it basically encodes the alphabet used.

To achieve the previous goal we will balance the number of states and the number of edges. For that we encode each symbol using $k < c$ digits. This means $\log_k c$ digits per symbol. Hence, our skeleton will have $O(n \log_k c)$ states, each one with at most k edges. Then, the total space is $O(nk \log_k c)$.

Intuitively, what happens is that the encoding permits to share transitions. We can see this by noting that the skeleton representing a symbol has k transitions times all the transitions of a skeleton one state short. That is

$$T_s = kT_{s-1}$$

and $T_1 = k$. But the length of the skeleton for each symbol is $\log_k c$. Thus, $T(\log_k c) = k^{\log_k c} = c$ different transitions per state. That is, the number of transitions per state in the $O(cn)$ DASG. Note that each transition in the previous version of the DASG, is simulated by the encoded DASG in $O(\log c)$ steps.

The optimal choice for k is 3. However, for practical obvious reasons we want an integer power of two. In that case, the best integer choices are 2 and 4. Thus, using $k = 2$ (typically most inputs are already encoded in binary) we have at least 2 edges per state and $n\lceil\log_2 c\rceil + 1$ states. Of these states, $n + 1$ are final. However, we do not have to distinguish them, because any input must be of length multiple of $\lceil\log_2 c\rceil$. This leads to the following theorem:

THEOREM **3.1** *The smallest deterministic partial finite automaton that recognizes all possible subsequences of a text of size n over an alphabet of effective size c, has at most* $n\lceil\log_2 c\rceil + 1$ *states and at most* $(2n - (\lceil\log_2 c\rceil + 1)/2)\lceil\log_2 c\rceil$ *transitions.*

Proof: It is only necessary to prove the result in the number of edges. Clearly, any state has at most 2 edges. However, the last state has no transitions and the previous $\lceil\log_2 c\rceil$ states only can have 1 transition because they represent the last symbol. For the same reason, the skeleton representing the symbol $n - i$ has at most i states with 2 transitions for any $i \leq \lceil\log_2 c\rceil$. ∎

These upper bounds can be slightly improved using $k = 3$. This result is optimal, because the length of the encoded text is $O(n \log c)$, and then we need at least $O(n \log_2 c)$ transitions to recognize the complete text (the longest subsequence).

Figure 2 shows the encoded version for the text *abcd*. This DASG does not have less transitions that the one presented in Figure 1. However, this only happens for small n or periodic strings (for example a^n).

Again, to construct this version of the DASG, we use an incremental algorithm scanning from the right to the left. Now, we need two auxiliary structures. One that given a symbol tell us its encoding (encoding dictionary/function) and another that given a prefix of a symbol code, returns the position of the first symbol (in the previously scanned text) with that prefix (analogous to the D dictionary of the previous algorithm). For the last data structure we use a binary

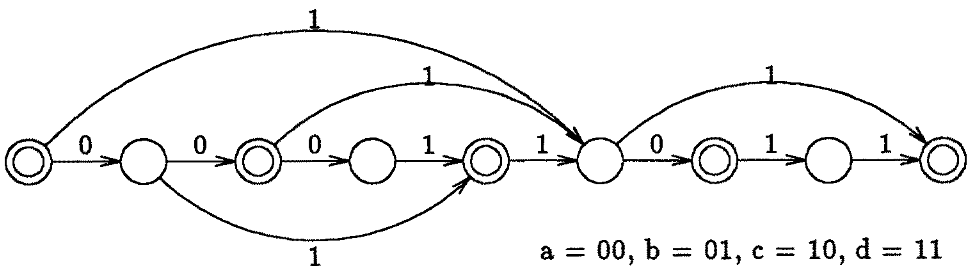

Figure 2: DASG for the text *abcd* (encoded).

trie (for example a Patricia tree [8]), where in each node we store the position (state) needed. Let b be $\lceil \log_2 c \rceil$. If c is not known in advance, we may use $c = n$ or we compute it using $O(n \log c)$ time. The detailed steps of the algorithm are:

1. Create state s_{nb+1} and create an empty binary trie D.
2. For each symbol in the text t_i scanning from the right to the left do:
   (a) Set the root as the actual position in D.
   (b) Create state $s_{(i-1)b}$.
   (c) Encode t_i.
   (d) For every bit x_j (0 or 1) in the encoding of t_i do:
      i. Create state $s_{(i-1)b+j}$ if $j < b$.
      ii. Append a transition labeled x_j between states $s_{(i-1)b+j-1}$ and $s_{(i-1)b+j}$.
      iii. If the $\overline{x}_j$ (complement of x_j) child of the current trie node exist, append a transition labeled $\overline{x}_j$ from state $s_{(i-1)b+j}$ to state k where k is the state stored in the child.
      iv. Set the x_j child of the current trie node as the new position in D and update its value (state) to $(i-1)b+j$. If the child does not exist, we create it.

All the steps in the internal loop takes constant time, and the internal loop is repeated nb times. Hence, the total time is $O(n \log c)$. The extra space is $O(c \log c)$ for the trie and $O(n \log c)$ for the encoding structure (if we do not have a function or table). This leads to the following theorem:

THEOREM **3.2** *It is possible to construct the DASG of a text of size n using $O(n \log n)$ worst case time and space for arbitrary alphabets, and using $O(n \log |\Sigma|)$ worst case time and space for finite alphabets.*

In practice the implementation is very simple. We need two words for each state (the 2 possible transitions), and we need $nb + 1$ states (contiguous space) for the whole automaton.

A membership test of a subsequence s is answered in $O(|s| \log_2 c)$ time (the time to encode the subsequence plus the the time to answer the query). Additionally to the DASG we may have to maintain a structure or a function to encode each symbol. This at most requires $O(n \log c)$ space. In practice this is not needed, since most inputs are already encoded in binary (e.g. ASCII). By keeping all the states visited during the search we can obtain where the subsequence started and where it finished.

The previous result proves the following (almost obvious) lemma:

Lemma 3.1 *The minimal state (partial) DFA and the minimal transition (partial) DFA are not equivalent.*

The lemma is also true for non-partial DFAs because the space complexities for our problem are the same for this case. The meaning behind this lemma is that to share part of a transition function in 2 "similar" states we need additional states. Encoding is one technique to share states. However, it is possible that the general problem of finding the smallest transition function is NP-complete based on related problems presented in [4,5]. Further research is being done in this problem and in local techniques to minimize space in finite automata.

The next lemma gives a necessary condition to have an encoding that may reduce the size of the automaton:

Lemma 3.2 *Given a minimal state partial DFA with s states, where s_0 of them do not have outgoing transitions, and t transitions, then encoding may reduce the size of the automaton only if $t > 2(s - s_0)$.*

Proof: If we apply encoding, each state is at least transformed in 2 states. That means that the number of transitions of the automaton of the encoded text is at least $2(s - s_0)$ transitions, because each new state must have at least one transition, $s - s_0$ of the original number of states also must have one transition and it is not necessary to encode symbols representing states without transitions. Hence, the new automaton may have less transitions if $t > 2(s - s_0)$. ∎

For example, any DAWG such that $t \leq 2s - 2$ ($s_0 = 1$ for this case) cannot be reduced using encoding. We have not found a single example where $t > 2s - 2$ for a DAWG. Based in the results presented in [3] we know that $t < 3s - 6$.

4 A Linear Space Representation

In section 3 we showed that we can transform the DASG of $O(n^2)$ transitions and $O(n)$ states, to a DASG with $O(n \log n)$ transitions and states. In this section we will describe how to simulate the $O(n^2)$ space DASG using only $O(n)$ space, but $\log n$ time per transition, independently of the alphabet size.

Instead of representing the transitions for each state, we will store all the states associated to the transitions of a given symbol. Let enumerate the states in the DASG defined in section 2 from 0 to n, or in other words, by using the position of each symbol in the text. For each symbol x we store, in order, all states s such that

$$\delta(i,x) = s$$

for any state i (in fact, $i < s$), where δ is the transition function. That is, we store all the positions in the text in where x appears. Let S_x be the ordered list of positions associated to x. To simulate $\delta(i,x)$, we look in S_x for the minimum state s in S_x such that $s > i$. Because the list is ordered, this takes $O(\log n)$ time (a sorted array suffices). To know where S_x is, we use an auxiliary index that tells us this information for each x.

Because there are n positions in the text, the space necessary for all the ordered lists is $O(n)$. The time necessary to construct this representation is $O(n \log n)$ to sort the lists, and $O(n \log c)$ to build the auxiliary index and to lookup all the symbols. This leads to the following theorem:

THEOREM **4.1** *It is possible to construct an implicit representation of the DASG of a text of size n using $O(n)$ space and $O(n \log n)$ worst case time, in where each transition is simulated in $O(\log n)$ steps.*

To test membership of a subsequence s, we need $O(|s| \log c)$ time to lookup each symbol, and $O(|s| \log n)$ time to simulate the transitions. That is, $O(|s| \log n)$ time, regardless of the alphabet size. Therefore, for finite alphabets we tradeoff space for search time. Table 1 shows a summary of the space and time complexities.

5 The DASG for a Set of Strings

Now we want to solve the following problem: Is a given string a subsequence of a string in a set of strings? Again, we can express the problem as a regular expression. To do this, we need first some additional notation.

Let S be a set of L strings, and s_i be the i^{th} string of the set. We assume that no string is a subsequence of any other string (this implies that at least there are

DASG	Space	Searching time	Building time
Section 2	nc	$\lvert s\rvert$	nc
Section 3	$n\log c$	$\lvert s\rvert \log c$	$n\log c$
Section 4	n	$\lvert s\rvert \log n$	$n\log n$

Table 1: Summary of time and space complexities

two different symbols in S). Let $n = \sum_{i=1}^{L} |s_i|$ be the total number of symbols. Let $T(S)$ be the set of distinct symbols in S $(2 \leq |T(S)| \leq c = \min(|\Sigma|, n))$.

Definition: We define (as in [6]) a *matched point* of S as a j-tuple of pairs $([i_1, p_1], [i_2, p_2] ..., [i_j, p_j])$ $(1 \leq j \leq L)$ which denotes a match of a symbol at positions p_1 in string s_{i_1}, p_2 in string s_{i_2}, ..., p_j in string s_{i_j}. A matched points is *maximal*, if the symbol matched does not appear in the $L - j$ remaining strings.

For example, all the maximal matched points for $S = \{aba, aab, bba\}$ are

$$([1,1],[2,1],[3,3]), ([1,1],[2,2],[3,3]), ([1,2],[2,3],[3,1]),$$
$$([1,2],[2,3],[3,2]), ([1,3],[2,1],[3,3]), \text{ and } ([1,3],[2,2],[3,3]).$$

Definition: We define the *initial maximal matched point* $(IM(S, x))$ in the set S for a given symbol x as the smallest maximal matched point (in a lexicographical sense) that matches x. That is, the maximal matched point with the smaller position p_i in each string that belongs to the matched point.

For the previous example, $IM(S, a)$ is $([1,1],[2,1],[3,3])$ and $IM(S, b)$ is $([1,2],[2,3],[3,1])$.

We denote by $R(S,$ *matched point*$)$ (*right set*) the set of non null substrings that are to the right (higher positions) of a matched point (we also eliminate any substring that is a subsequence of other substring). For the previous example, $R(S, IM(S, a)) = \{ba, ab\}$. Now, the regular expression that defines all possible common subsequences of S is recursively defined by

$$Subseq(S) = \sum_{t_i \in T(S)} t_i Subseq(R(S, IM(S, t_i)))$$

and $Subseq(\emptyset) = \epsilon$. This definition generates the subsequence automaton, and then allow us to count the number of states and edges needed by this automaton:

$$States(S) \leq 1 + \sum_{t_i \in T(S)} States(R(S, IM(S, t_i)))$$

and

$$Edges(S) \leq |T(S)| + \sum_{t_i \in T(S)} Edges(R(S, IM(S, t_i)))$$

Both results are not equalities, because identical right sets may appear (duplicated partial results). An example is given in Figure 3.

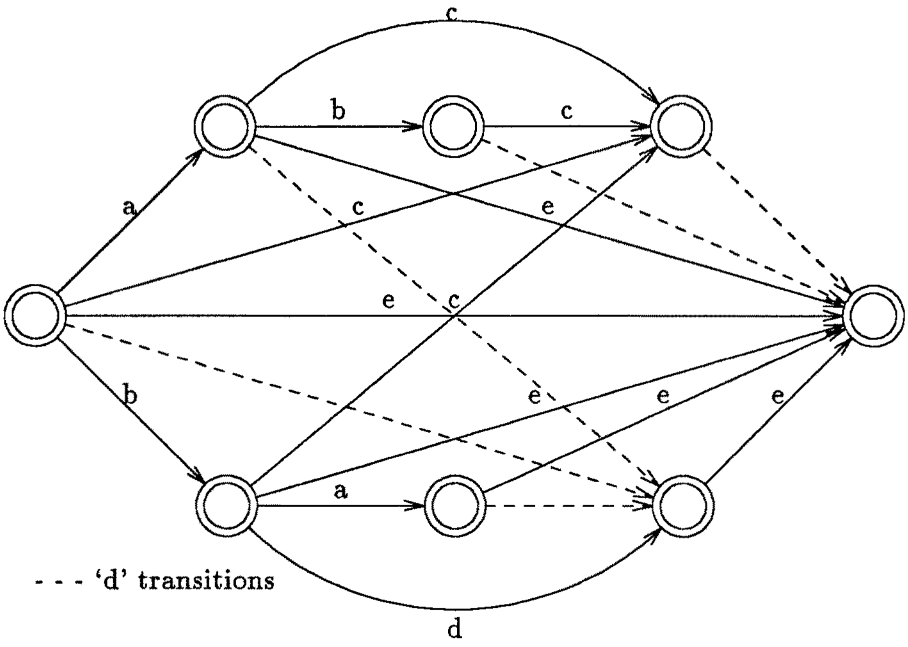

Figure 3: DASG for the strings *abcd* and *bade*.

THEOREM **5.1** *The DASG of a set of L strings of size n over an alphabet of effective size c has at most $n - L + 2$ states and at most $(N - L + 1)c$ edges.*

Proof: We use induction on the number L of strings. From section 2, the theorem is true for $L = 1$ ($n + 1$ states are necessary and sufficient).

Now, we will see what happens when we try to include a new string s in a DASG of a set S of L strings of size n. We will show that for each position in s (except one) we need to create at most one state. If we create a state for a transition labelled with s_j, we mark that position j in the string. We show that if position j has been marked, then there exists a state that recognizes $Subseq(s')$ and nothing else, where $s' = s_{j+1} \ldots s_{|s|}$. Note that the last position will be never marked, because $Subseq(\epsilon)$ exists already in the DASG of S (last final state or *sink* state).

Then, for each position j in s (the order is not important) we need a transition from the initial state labelled with that symbol (s_j). For the last position, if there is no transition from the initial state labelled with that symbol, we create a transition from the initial state to the sink state. For the other positions, we have three cases:

- A transition with that symbol does not exist and position $j+1$ has never been marked. In this case we create a new transition labelled with that symbol to a new state, and we mark that position. From, this state we apply this procedure recursively on the new state for the string s'. Note, that this new state will recognize only $Subseq(s')$.
- A transition does not exist, but position $j+1$ has been marked. Therefore, there exist a state that recognizes only $Subseq(s')$, and we create a transition labelled with s_j to that state.
- A transition with that symbol already exists. Hence, we use that transition, and we follow it. Now, from that state, we apply this procedure recursively.

Then, in the worst case, $|s|-1$ states are created (less, if we have common suffixes). Hence, the size of the new DASG is at most

$$States(S \cup \{s\}) \leq States(S) + |s| - 1$$

Using the inductive hypothesis, we have

$$States(S \cup \{s\}) \leq n - L + 2 + |s| - 1 = (n + |s|) - (L+1) + 2$$

as claimed.

The bound in the number of edges is obtained using the fact that $N - L + 1$ states have transitions, and that the number of transitions per state is bounded by c. ∎

The bound is tight on the number of states, because if all the symbols are different, $n - L + 2$ states and $O(n^2/L)$ transitions are needed.

For this case, it is not possible to use the encoding technique of the previous section. If not, it would be possible to solve the LCS problem ($L = 2$) in $O(n \log n)$ comparisons for an arbitrary alphabet. This is a contradiction with the $O(n^2)$ lower bound in the comparison model presented in [2]. Therefore, the size of the DASG must be $O(n^2)$ for this case. In fact, the encoding technique fails for this case, because now we have more than one skeleton.

The only structure that resembles our automaton is the *ICS* tree of Hsu and Du [6], which is used to solve the LCS problem for a set of strings. In that case, only matched points between all the strings are considered.

Now, we will present the main ideas behind the algorithm that builds the DASG for this case. The algorithm must find, very efficiently, all possible different symbols at any state. For this, we first build the DASG (first version given) for each individual string. With this, for each string, we can find all different symbols after a given position. The algorithm recursively generates states until all possible symbols belongs to one string (or there are no more symbols left in that position). After that, the individual DASG is used. To keep track of how much of this DASG we have used we have a list of L positions D that indicates from where the DASG of each string is already available.

To find if a right set has been already generated we need two structures. First, a structure that maps common suffixes to one representative. For this we use a suffix tree of the strings (using $O(n)$ space and time). Second, to remember all the right sets (partial results) we use a dictionary that given a right set tell us where it is or if does not exist.

The algorithm is:

1. Create the last state F.
2. Create the right set remember dictionary, inserting the empty right set and its associated state F.
3. Initialize $D_j = |s_j|$ for $j = 1, ..., L$.
4. Set up the table of representatives.
5. Create the DASG for each string ($DASG_j$) using F as common last state.
6. Call *Merge* with pairs $(j, 1)$ for $j = 1, ..., L$. *Merge* will return the initial state.
7. Remove the first $D_j - 1$ states of each $DASG_j$.

The procedure *Merge* does almost all the work, merging the strings from position i_j for all j in the set of pairs P. Namely:

1. If $|P| = 0$ then return state F (no symbols left).
2. Else look up on the right set remember structure. If it is there, return the appropriate state.
3. Else if $|P| = 1$ then we can use the individual DASG (or a copy of it). Let j be the position different to 0. Return state i_j of $DASG_j$ and if $i_j < D_j$, set i_j as the new value of D_j.

4. Otherwise
   (a) Create a new state N
   (b) Insert P in the right set remember structure, and its associated state N.
   (c) Look for all different symbols in state i_j of $DASG_j$ for all values of j in P. For each new symbol x, create a transition labeled by x between state N and the state returned from $Merge$ called with pairs Q, where Q is defined as the pairs in P with all the positions updated to $\delta(i_j, x)$, where δ stands for the transition function. If $\delta(i_j, x)$ does not exist, or is the state F, we remove the pair corresponding to string j.
   (d) Return state N.

Step (b) takes time $O(\log States(S)) = O(L \log n)$ and we need a bit of care in step (c). To look for all different symbols defined by the set of positions P, we assume that there is a lexicographical order between the symbols, and that the edges of each $DASG_j$ are ordered. Then, we can look all the edges in time proportional to the number of edges to obtain all possible different symbols. If d is the number of different symbols, then at most Ld edges are inspected. Because d edges will be then generated, the time used is proportional to L for each new state created. At the same time that we inspect the edges, we build the new set of pairs Q. The size of the stack is at most $L \max_i(|s_i|) = O(Ln)$.

The time for each call to merge in the worst case is then $O(L \log n)$. There are as many calls to merge as matches between the strings (or edges created by *Merge*). The construction of the individual DASGs takes time $O(\sum_i |s_i|^2)$ and the construction of the table of representatives takes time and space $O(n)$. Hence, the total time is $O(L|Edges(S)| \log n + n^2/L)$. This leads to the following theorem:

THEOREM **5.2** *It is possible to construct the DASG for a set of L strings using* $O(Ln^2 \log n)$ *worst case time and* $O(n^2)$ *space for arbitrary alphabets, or using* $O(L|\Sigma| n \log n)$ *worst case time and* $O((L + |\Sigma|)n)$ *space for finite alphabets.*

The time to test membership of a subsequence s is in the worst case $O(|s| \log c)$, and $O(|s|)$ for small alphabets by using an array in each state. For arbitrary alphabets, we can achieve $O(|s|)$ average time by using a hashing table.

5.1 An Application: The Longest Common Subsequence

Additionally to fast searching of subsequences in a text or a set of strings, we can also use the DASG to solve the longest common subsequence problem, and some of its variants. For that purpose, we append to each edge (transition) the number of strings that are represented by that edge. Then, to know which is the longest common subsequence problem between $k \leq n$ strings, we search for the longest sequence of edges belonging to k or more strings. The LCS of all the strings is when $k = n$. This LCS can also be computed while the automaton is being built. In all these cases, we find all common subsequences in optimal time.

Hence, the DASG can be used to solve the LCS problem and many variants of it in time $O(n^2 \log n)$ using $O(Ln^2)$ space. This improves over the solution presented in [6] that uses $O(nc + Lc|P|)$ time and $O(nc + |P|)$ space, where $|P|$ is the total number of matched points between all the strings. Because $|P|$ may be as big as $O(n^2)$, this algorithm runs in $O(n^3)$ worst case time for arbitrary alphabets ($O(|\Sigma|n^2)$ for finite alphabets) using $O(n^2)$ space.

6 Concluding Remarks

We define the DASG of a text, giving different algorithms to build it. If the alphabet is known and finite, the DASG presented in section 3 uses only $O(n \log |\Sigma|)$ space and preprocessing time, and $O(|s| \log |\Sigma|)$ searching time for a subsequence s. To achieve this, we have introduced encoding as a technique to reduce the number of transitions in an automaton.

For arbitrary alphabets, the implicit representation of section 4 uses only $O(n)$ space, but $O(|s| \log n)$ searching time.

We used the number of transitions to measure the size of an automaton, and this problem shows that a minimal state automaton is in general not a minimal space automaton.

Remains as open problems the uniqueness of the minimal DASG and the complexity of transition function minimization in a DFA for a general case.

A related problem, is to search for a sequence of substrings. Using a Patricia tree [8], where each internal node have an ordered list of all the positions associated to the corresponding prefix, we can solve this problem in logarithmic time, using $O(n \log n)$ space on average, for a text of size n.

We extended the definition of the DASG to a set of strings, and we use it to solve the LCS problem between those strings, and several variations of it using $O(n^2 \log n)$ time and $O(n^2)$ space improving previous solutions for the case of more than two strings. Other application of this DASG is related to subset membership problems.

Acknowledgements

We wish to thanks the helpful comments of Gaston Gonnet.

References

[1] Aho, A., Hopcroft, J. and Ullman, J. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] Aho, A., Hirschberg, D. and Ullman, J. "Bounds on the Complexity of the Longest Common Subsequence Problem", *JACM* 23 (1976), 1-12.

[3] Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., and Seiferas, J. "The Smallest Automaton Recognizing the Subwords of a Text", *Theoretical Computer Science*, 40 (1985), 31-55.

[4] Garey, M. and Johnson, D. *Computers and Intractability, A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[5] Hopcroft, J. and Ullman, J. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1969.

[6] Hsu, W. and Du, M. "Computing a longest common subsequence for a set of strings", *BIT* 24 (1984), 45-59.

[7] Itoga, S. "The string merging problem", *BIT* 21 (1981), 20-30.

[8] Morrison, D. "PATRICIA-Practical algorithm to retrieve information coded in alphanumeric", *JACM 15*, 4 (Oct 1968), 514-534.