The Factor Automaton[1]

Milan Šimánek

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech technical University
Karlovo nám. 13, 121 35 Prague 2

e-mail: `simanek@fel.cvut.cz`

Abstract. The direct acyclic word graph (DAWG) is a good data structure representing a set of strings related to some word with very small space complexity. The famoust DAWG is the factor DAWG which is representing the set Fac(*text*) of all factors (substrings) of the string *text*. Bellow we call factor DAWG as DAWG. Finite automaton implementing this data structure is able to make out any substring of string *text* in time proportional only to length of the substring while its space complexity is linear to the length of the string *text*. We can define several operations on DAWG. Operations are usefull for fast derivating of the DAWG automaton from a similar one. This paper concern operation L-delete on factor graph DAWG and the relationship between deterministic and nondeterministic factor automaton.

Key words: DAWG, factor automaton, substring, pattern matching, fast searching

1 Introduction

The factor automaton is a finite automaton which accepts the set of all substrings of the string [1, chapter 6]. The set of all substrings (factors) of the string *text* is Fac(*text*).

This factor automaton can be formulated as a deterministic one or a nondeterministic one. The nondeterministic factor automaton is a good abstraction for formal description of its behaviour and of operations performed on it. On the other hand the deterministic one is used for implementation and practical use. This version is sometimes called direct acyclic word graph, $DAWG$, because it has no transition loop.

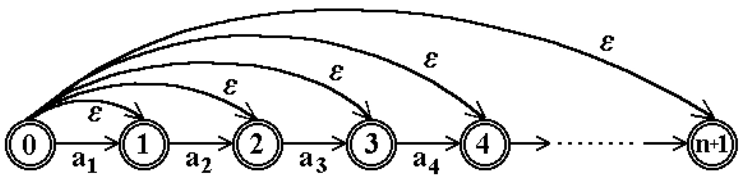
The main advantage of the $DAWG$ is very fast substrings searching while it keeps small memory requirments. Any matching string can be found in time equal to the length of the pattern looking for. The size of the factor automaton $DAWG(text)$ is linear with respect to the length of the string $text$. Total number of the nodes is less then double length of the input string $text$. The proof is in [1, Theorem 6.1].

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

2 Construction

2.1 Nondeterministic factor automaton

The nondeterministic factor automaton, which accepts all substrings of the string *text*, has $N + 1$ states and $2N - 1$ transitions, where N is the length of the string *text*. The structure of this automaton for string *text*= $a_1a_2a_3a_4...a_N$ is shown on the next picture.



2.2 Deterministic factor automaton

The deterministic factor automaton DAWG can be obtained from nondeterministic one [3] or we can construct it step by step using an incremental construction algorithm [1, 6.3 On-line construction]. Although we have a construction algorithm, in general, we cannot say anything about the structure of transitions except nonexistence of the circle and an estimate bounds of the number of states. The pattern matching using this automaton has optimal speed. The number of comparations (or other elementar operations) is linear to the length of the searching pattern.

2.3 Relation between deterministic and nondeterministic automata

It appears that every construction method produces equivalent (isomorphic) deterministic factor automaton. We can say the deterministic factor automaton is the best simulation of the nondeterministic one. In this simulation every state in deterministic automaton corresponds to a set of active states in nondeterministic automaton. This relationship can by very usefull for discovering and proving new algorithms for deterministic automata.

3 Operations on factor automaton

We can define a number of operations on factor automaton. Each operation modify given factor automaton representing string *text* to a new factor automaton representing another string *text'* while strings *text* and *text'* are very similar. It is important that both new and old factor automaton will be similar too and therefore performing the operation spend a little amount of time.

We will deal with this operations on a factor automaton:

operation	**action** *text'*
Append	the string *text'* will be longer by a character
Insert	inserts a character before the first character of the string *text'*
$R - delete$	*text'* is the string *text* without the last character
$L - delete$	*text'* is the string *text* without the first character
Replace	replaces one character in string *text* by another one

The algorithms for some operations have been yet discovered (*Append*, $R{-}delete$), but the algorithms of *Insert* and *Replace* are not known. This article concern about the algorithm of the $L - delete$ operation.

This operation modifies DAWG($a_0a_1a_2a_3...a_n$) to another factor automaton accepting all substrings of the string $a_1a_2a_3...a_n$ which is by a first character a_0 shorter then the original string $a_0a_1a_2a_3...a_n$. The algorithm is shown bellow.

The combination of operations *Append* and *L-delete* enables fast searching in the compression method known as LZ77 which use so called sliding window. Sliding window contains a part of source text with constatnt length. The window is moving through the text so at the begining it contains the first k characters of the text and at the end operating it contains the last k characters of the source text.

4 DAWG in details

To enable incremental construction of this factor automata (*append* operations) requires to keep a bit more information about the DAWG working on. In every step we should know the set of states (a state of finite automaton per a node of the DAWG), transitions between the states (representing edges of the DAWG), and the fail function. The fail function is used for creating and extending DAWG. We will need know which is the next character for each state for the *L-delete* operation.

Before we will show the algorithm we should make some denotation. *Next(q)* is a following character in source string *text* for each state q in the DAWG factor automaton. Concatenation of $Next(q_0) + Next(Next(q_0)) + ...$ gives the string *text* for DAWG(*text*). There is defined the fail function *Fail(q)* for each state q of DAWG automaton. If the automaton is in the state q_1 after reading substring uv and in the state q_2 after reading substring v which is the longest possible then $Fail(q_1){=}q_2$. Factor automaton being in state q_2 accepts each suffix which is accepted in state q_1. $Skip(q)$ is the set of states p_i which $Fail(p_i)$ is equal to state q. Function $Skip$ is the inverse function of function $Fail$: $p \in Skip(q)$ iff $q = Fail(p)$.

5 The algorithm of L-delete operation

Let main chain is a sequence of states q_0, $q_1{=}\delta(q_0, \mathrm{Next}(q_0))$, ..., $q_i{=}\delta(q_{i-1}, \mathrm{Next}(q_{i-1})$, ..., q_n. The idea of this algorithm is to disable passing only through a part of the main chain but to protect passing anyway through at least one skip transition.

This algorithm duplicates the starting part of main chain of states. One copy (original) of begin of main chain is used for processing these substrings which will pass through some skip transition later. Second copy (duplicated) is used for processing these states which have passed some skip transition before.

Not all main chains will be duplicated. The duplication process stops at the state where is obvious which shift transition will be pass. This stop state is determined by a value of Skip function. Assume last duplicated state is r. Next state to be duplicate is s. Let state $t = \delta(s, Next(s))$ is the next state after s. If the set of states Skip(t) is empty then duplication process stops, because no shift transition can be pass. If the set of states Skip(t) contain only one state, then duplication process stops too, because only one shift transition is possible and therefore it can be done immediately. Otherway if the number of states Skip(t) is greater then one then duplication process continue.

INPUT:	DAWG(aw)
OUTPUT:	DAWG(w)
LOCAL VARIABLES:	**a** – a character
	q_0, q_1, **r**, **s**, **t**, **d** – states
	q_0 – the initial state

```
a:=Next(q0)
q1:=δ(q0,a)
if |Skip(q1)| = 0 then
        δ(q0, a) := nil
        delete(q1)                              possible recursive delete
else if |Skip(q1)| = 1 then
        δ(q0, a) := Skip(q1)
        delete(q1)                              possible recursive delete
else
        r:=q0
        s:=q1
        loop
                a:=Next(s)
                t:=δ(s,a)
                if |Skip(t)| < 2 then break
                d:=duplicate(t)
                δ(r,a):=d
                Fail(t):=d
                r:=d
                s:=t
        endloop
        if |Skip(t)| = 1 then
                δ(s,a):=Skip(t)
        else
                δ(s,a):=nil
        endif
endif
```

6 Time and memory complexity

It seems that the time complexity of one *L-delete* operation is at least constant or in the worst case linear to length of the text *text*. The DAWG(*text*) for string *text* of length N has at most $2 * N$ states [1]. Therefore the time complexity of sequence of N *L-delete* operations is linear to N.

The number of states of DAWG(*text*) is limited by $2.N$ where N is number of characters in source string *text*. Moreover, DAWG(*text*) has less than $3.N$ edges. This is independent of the size of the alphabet [1, Theorem 6.1].

7 Conclusion

The power of operation *L-delete* grows up in conjunction with the operation *append*. We can apply k-times operation append which constructs the base DAWG for first k characters of the text. Then we will apply repeatively a couple of operations *L-delete* and *append*. We will get a moving window for fast searching in this part of the text. The speed of searching is independent of size of the searching window and depends only on the size of pattern looking for. The main application can be LZ77 compression algorithm. The part consuming the largest amount of time is just the algorithm searching for a pattern in a searching window. Using this searching algorithm should speed up compression.

References

[1] Crochemore, M., Rytter, W.: **Text Algorithms,** chapter 6, Subword graphs, Oxford University Press, 1994

[2] Chen, M. T., Seiferas, J.: **Efficient and elegant subword tree construction,** Combinatorial Algorithms on Words, NATO Advanced Science Institutes, Series F, vol. 12, Springer-Verlag, Berlin, 1985, 97–107

[3] Melichar, B.: **The construction of factor automaton**, Workshop 98, Czech Technical University, Prague 1998, 189–190