A NOTE ON LINEAR TIME SIMULATION
OF DETERMINISTIC
TWO-WAY PUSHDOWN AUTOMATA

by

Neil D. Jones

DAIMI PB-75
April 1977
Keywords:
    two-way pushdown automata
    linear time
    string matching
    pattern matching
    random access machines

INTRODUCTION

In [1], Cook showed that any deterministic two-way pushdown automaton could be simulated by a uniform-cost random access machine in time $O(n)$ for inputs of length $n$. The result was of interest because such a machine is a natural model for a variety of backtracking algorithms, particularly as used in pattern matching problems. The linear time result was surprising because of the fact that such machines may run as many as $2^n$ steps before halting; similar problems with "combinatorial explosions" are well known to occur in applications of backtracking. Cook's result inspired the development of a number of efficient pattern matching algorithms, for example those in [2] and [3].

However, it is impractical to use Cook's algorithm directly to do pattern matching, since it involves a large constant time factor and much storage. The purpose of this note is to present an alternate, simpler simulation algorithm which involves consideration only of the configurations actually reached by the automaton. It can be expected to run faster and use less storage (depending on the data structures used), thus bringing Cook's result a step closer to practical utility.
TERMINOLOGY

Let $M = (Q, \mathcal{J}, \mathcal{P}, \delta, q_0, Z)$ be a deterministic two-way pushdown automaton, fixed throughout this paper. $Q$ is the set of states, $\mathcal{J}$ the set of input symbols, $\mathcal{P}$ the set of stack symbols including the bottom-of-stack symbol $Z$, $\delta$ the transition function and $q_0$ the initial state. A configuration of $M$ is a triple $C = (q, i, A)$ where $q$ is a state, $A \in \mathcal{J}$, $\text{Top}(C)$ is a stack symbol and $0 \leq i \leq n+1$ for input $x \in \mathcal{J}^*$ of length $n$. An instantaneous description or ID is a pair $(C, Z\alpha)$ where $C$ is a configuration, $\alpha \in \mathcal{P}^*$ and the right-most symbol of $Z\alpha$ is $\text{Top}(C)$.

In one move by $\delta$ $M$ may change its state, move the input scanning head, and push, pop or leave the stack unchanged. Alternately, $M$ may either accept or halt without accepting. A more detailed description of $M$'s operation may be found in [1].

We define $\text{pop}(C)$, $\text{push}(C)$, $\text{accept}(C)$, $\text{halt}(C)$ to be true just in case configuration $C$ causes the automaton to pop, push, accept or halt, respectively (in one step). We let $\text{NEXT}(C) = D$ just in case $C$ yields an instantaneous description $(D, \alpha)$ in one step (false if $\text{pop}(C)$, $\text{accept}(C)$ or $\text{halt}(C)$ is true).

Further, $\text{FOLLOW}(C, A) = (t, j, A)$ just in case the ID $(C, Z \text{Top}(C))$ yields $((t, j, Z), Z)$ in one step (false unless $\text{pop}(C)$ is true). $C \rightarrow D$ is true just in case the ID $(C, \text{Top}(C))$ yields $(D, \text{Top}(C))$ in some number of steps, and every intermediate ID has at least two symbols in its stack. The reflexive transitive closure of $\rightarrow$ is written $\rightarrow^*$. The terminator of $C$ is $D$ just in case $C \rightarrow^* D$ and $\text{pop}(D)$. Note that $D$ is unique if it exists.
THE ALGORITHM

The algorithm is presented in the style of Algol. Its heart is the recursive procedure SIM(C) whose purpose is to return the terminator of C; this is also stored in table entry T[C], to avoid recomputation after the first call. As long as the automaton does not push, it is simulated step by step until the terminator is found, or a halt or accept configuration is reached. Note: We assume for now that looping does not occur. A push configuration C is handled by a recursive call to calculate the terminator of NEXT(C), that is the configuration containing the new top symbol which was just pushed. From this it is easy to find the D such that C ⊢ D.

During the call SIM(C), the stack LIST will contain all the configurations D_1, D_2, ..., D_m so that C = D_1 ⊢ D_2 ⊢ ... ⊢ D_m, stopping when D_m is a pop, accept or halt configuration, or it is found that the terminator of D_m has already been computed (i.e. when T[D_m] ≠ 0). The algorithm ends by storing the terminators of D_1, ..., D_m into table T for future reference. Correctness of the algorithm should be evident, as it merely simulates the automaton one step at a time, taking a "short cut" in case a configuration is encountered whose terminator has already been computed.
begin configuration LAST;
configuration array T [configuration];
configuration procedure SIM(C); configuration C;

begin configuration D, E; stack of configuration LIST;
if T[C] ≠ 0 then SIM := T[C]
else begin
    D := C; LIST := EMPTY;
    while not (pop(D) or accept(D) or halt(D)) do
        begin
            push D onto LIST;
            if push(D) then begin E := SIM(NEXT(D));
                D := FOLLOW(E, Top(D))
            end
        else D := NEXT(D);
    if T[D] ≠ 0 then D := T[D]
end;
if accept(D) then accept else if halt(D) then halt;
T[D] := D;
while LIST ≠ EMPTY do
    begin pop E from LIST; T[E] := D
    end;
SIM := D
end;
end PROCEDURE SIM;
LAST := SIM(Initial Configuration);
end
TIME ANALYSIS

To analyze the algorithm's time usage, define the C-call to consist of all steps between the time SIM is first called with C as actual parameter, and the time the first call is completed. Define the C-list to be the value of LIST during the C-call, just after the first while loop is exited. Now we make a few observations.

1. SIM(C) is not called from within the C-call (if so, the automaton is in a loop).
2. Any SIM(C) call after the C-call takes constant time since T[C] ≠ 0 after the C-call.
3. A configuration D is placed on the C-list only if T[D] = 0.
4. No configuration D appears on two C-lists. To see this, suppose D is on the C₁-list and the C₂-list. We may assume the C₁-call begins before the C₂-call.

Case 1
The C₁-call finishes before the C₂-call begins. Then T[D] becomes nonzero during the C₁-call, so D is not placed on the C₂-list.

Case 2
The C₁-call contains the C₂-call.

a) if D is not on the C₁-list before the C₂-call starts, then the C₂-call sets T[D] ≠ 0, and so D will never appear on the C₁-list;

b) otherwise C₁ ⊢* D, (D, Top(D)) yields (C₂, α) in some number of steps for some α, and C₂ ⊢* D. In this case the automaton is in a loop, with an infinitely growing stack just in case |α| > 1.
5. \( \text{SIM}(C) \) is called at most a constant number of times. To see this, note that whenever \( \text{SIM}(C) \) is called there must be \( C_1, B \) such that 
\( B \) is in the \( C_1 \)-list and \( \text{NEXT}(B) = C \); call this a \( C_1, B \)-call. If 
there is both a \( C_1, B \)-call and a \( C_2, B \)-call then \( B \) appears on two 
lists, a contradiction. Thus the number of \( \text{SIM}(C) \) calls is at most 
the size of \( |B| \cdot \text{NEXT}(B) = C \). But it is easy to see that this set 
is of size independent of the length of the input string.

6. The time for the \( C \)-call is \( O(\text{size}(C \text{-list})) \).

Consequently the total time is bounded by
\[
\sum_C (\text{time for } C \text{-call} + \text{ time for other } \text{SIM}(C) \text{ calls}) 
\leq \sum_C O(\text{size}(C \text{-list})) + \sum_C \text{constant} = O(n) + O(n) \cdot \text{constant} = O(n).
\]

A more detailed analysis would surely reveal a smaller time constant 
than that of \[1\], since the algorithm above examines the smallest possible 
number of configurations — namely those which actually occur in the com-
putation. For example, a 3-state automaton accepting \( |x^R y| \), \( x, y \in \{a, b\}^* \), 
y a subword of \( x \) when given the input \( ba^m ca^n b \) requires examination of 
9(\( m+n \)) configurations by the method of \[1\] and \( m+n \) by ours (both modulo an 
additive constant).

In the above we assume that the automaton was nonlooping; now sup-
pose it does loop. Then there must be reachable IDs such that \( (D, \alpha) \) yields 
\( (D, \alpha \beta) \) in some nonzero number of steps. If we apply the algorithm above, 
\( D \) will appear on two \( C \)-lists if \( \beta \neq \epsilon \), or twice on the same \( C \)-list if \( \beta = \epsilon \). 
The converse follows from observation 4 above. Consequently we can re-
cognize loops by storing a truth value \text{ONLIST}[D] \text{ for each configuration } \text{D}. Initially \text{ONLIST} is initialized to \text{false} for every \text{D}. The algorithm is
modified by inserting just before "push D onto LIST;" the following:

\[
\begin{align*}
\text{if } \text{ONLIST}[D] \text{ then halt} \\
\text{else } \text{ONLIST}[D] := \text{true};
\end{align*}
\]

REFERENCES

