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

In one move by δ 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 pop(C), push(C), accept(C), 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 NEXT(C) = D just in case C yields an instantaneous description (D, α) in one step (false if pop(C), accept(C) or halt(C) is true). Further, FOLLOW(C, A) = (t, j, A) just in case the ID $(C, Z\ \mathrm{Top}(C))$ yields $((t, j, Z), Z)$ in one step (false unless pop(C) is true). $C \vdash D$ is true just in case the ID $(C, \mathrm{Top}(C))$ yields $(D, \mathrm{Top}(C))$ in some number of steps, and every intermediate ID has at least two symbols in its stack. The reflexive transitive closure of $\vdash$ is written $\vdash^*$. The <u>terminator</u> of C is D just in case $C \vdash^* D$ and 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 $C \vdash D$.

During the call SIM(C), the stack LIST will contain all the configurations $D_1, D_2, \ldots, D_m$ so that $C = D_1 \vdash D_2 \vdash \ldots \vdash D_m$, 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 $T[D_m] \neq 0$). The algorithm ends by storing the terminators of $D_1, \ldots, D_m$ 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] $\neq$ 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] $\neq$ 0 then D := T[D]
            end ;
        if accept(D) then accept else if halt(D) then halt;
        T[D] := D;
        while LIST $\neq$ 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 with 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 $T[C] \neq 0$ 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 C_1-list and the C_2-list. We may assume the C_1-call begins before the C_2-call.

   Case 1

   The C_1-call finishes before the C_2-call begins. Then $T[D]$ becomes nonzero during the C_1-call, so D is not placed on the C_2-list.

   Case 2

   The C_1-call contains the C_2-call.

   a) if D is not on the C_1-list before the C_2-call starts, then the C_2-call sets $T[D] \neq 0$, and so D will never appear on the C_1-list;

   b) otherwise $C_1 \vdash^* D$, (D, Top(D)) yields (C_2, α) in some number of steps for some α, and $C_2 \vdash^* D$. In this case the automaton is in a loop, with an infinitely growing stack just in case $|\alpha| > 1$.

5. SIM(C) is called at most a constant number of times. To see this, note that whenever SIM(C) is called there must be C_1, B such that B is in the C_1-list and 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 SIM(C) calls is at most the size of $\{B \mid \text{NEXT}(B) = C\}$. 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 0(size(C-list)).

Consequently the total time is bounded by

$$\sum_C (\text{time for C-call} + \text{time for other SIM(C) calls})$$

$$\leq \sum_C 0(\text{size(C-list)}) + \sum_C \text{constant} = 0(n) + 0(n) \cdot \text{constant} = 0(n).$$

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 computation. For example, a 3-state automaton accepting $\{xcy^R \mid x,y \in \{a,b\}^*$, 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 suppose it does loop. Then there must be reachable IDs such that (D,α) 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 recognize loops by storing a truth value ONLIST[D] for each configuration D. Initially ONLIST is initialized to <u>false</u> for every D. The algorithm is modified by inserting just before "push D onto LIST;" the following:

```
if ONLIST[D] then halt
             else ONLIST[D] := true;
```

REFERENCES

[1] Cook, S.A. Linear time simulation of deterministic pushdown automata, Proceedings IFIP Congress 1971, pp. 75-80 North-Holland, Amsterdam (1971).

[2] Morris, J.H., Jr., and Pratt, V.R. A linear pattern matching algorithm, Technical Report No. 40, Computing Center, University of California, Berkeley (1970).

[3] Weiner, P. Linear pattern matching algorithms, Conference Record, IEEE 14 annual Symposium on Switching and Automata Theory, pp. 1-11 (1973).