INFORMATION AND CONTROL 56, 34–51 (1983)

The Recognition of Deterministic CFLs in Small Time and Space

BURCHARD VON BRAUNMÜHL

Institut für Informatik,
Universität Bonn, Wegelerstrasse 6, 5300 Bonn, West Germany

STEPHEN COOK

Department of Computer Science,
University of Toronto, Toronto, M5S 1A4, Canada

KURT MEHLHORN

FB10–Informatik,
Universität des Saarlandes, 6600 Saarbrücken, West Germany

AND

RUTGER VERBEEK

Institut für Informatik,
Universität Bonn, Wegelerstarsse 6, 5300 Bonn, West Germany

Let $S(n)$ be a nice space bound such that $\log^2 n \leqslant S(n) \leqslant n$. Then every DCFL is recognized by a multitape Turing machine simultaneously in time $O(n^2/S(n))$ and space $O(S(n))$, and this time bound is optimal. If the machine is allowed a random access input, then the time bound can be improved so that the time–space product is $O(n^{1+\varepsilon})$.

1. INTRODUCTION

It is well known that each context free language (CFL) can be recognized by an algorithm using polynomial time and n^2 space (Hopcroft and Ullman, 1979), and by quite a different algorithm using superpolynomial time and $\log^2 n$ space (Lewis *et al.*, 1965). However, no algorithm is known for the recognition of an arbitrary CFL in polynomial time and sublinear space simultaneously.

For each *deterministic* CFL (DCFL), however, there is a recognition algorithm which runs in polynomial time and $\log^2 n$ space simultaneously, as

0019-9958/83 $3.00

first shown by Cook (1979). This puts each DCFL in the class SC of languages recognizable simultaneously in polynomial time and polynomial in $\log n$ space (Cook, 1981). In particular, Sudborough's complete language (1978) for the class of DCFLs is in SC, and this is perhaps the most natural language known to be in SC but not known to be recognizable in space $\log n$. (See Sudborough, 1978) for a discussion of whether all DCFLs can be recognized in $\log n$ space.) In Sudborough (1980), the author describes a family of languages complete for SC, and these provide other examples. But these complete language are "contrived" in the sense that the function $\log^k n$ must be used explicitly to describe them.

Although we do not know whether all CFLs are in SC, Ruzzo (1979) has shown they are all in the class NC dual to SC (Pippenger, 1979; Dymond and Cook, 1980; Hong, 1980). Here NC is the class of languages accepted by a multitape Turing machine in polynomial time and polynomial in $\log n$ reversals.

Returning to DCFLs, Cook (1979) proved a time upper bound of $n^5 \log^2 n$ and a space upper bound of $\log^2 n$ for his original recognition algorithm. Mehlhorn (1980) improved the time bound to $n^{2.87}$ with the same space bound by developing another algorithm for machines with a random access input. (More generally, he described a time–space tradeoff for such machines.) Independently von Braunmühl and Verbeek (1980) found a modification of Cook's algorithm which works on Turing machines in time $n^2/\log^2 n$ and space $\log^2 n$ (more generally with a time–space product of n^2) and on machines with a random access input in time $n^{1+\varepsilon}$ and space $\log^2 n$ (or in linear time and space n^{ε}, etc.) This result is optimal in the case of Turing machines, and for machines with random access input Verbeek (1981) has shown that the algorithm is optimal in the class of "pebbling strategies."

In the present paper a new algorithm for DCFL recognition is presented with the same time and space complexity for both machine models as the one in von Braunmühl and Verbeek (1980). It connects the ideas in Mehlhorn (1980) and von Braunmühl and Verbeek (1982), and allows an easier proof of both correctness and complexity.

The paper is organized as follows. Section 2 presents basic definitions, and Section 3 presents a simplified version of the recognition algorithm which is not quite optimal. Section 4 presents the final improved algorithm, yielding the time and space bounds stated in Theorem 1 for machines with random access input and in Theorem 2 for Turing machines.

2. Basic Definitions

We consider deterministic PDAs with the following restrictions:

(1) Every step is a push step (exactly one symbol is pushed onto the stack) or a pop step (one symbol is popped).

(2) Any computation with input w contains at most $|w|$ push steps. Obviously every DCFL is accepted by such a DPDA P.

A *configuration* of P is given by a triple $C = (q, W, v) =$ (state, stack content, position of the input head). Let $Comp(P, w) = C_0, C_1, ..., C_m$ be the computation of P with input w. C_i is a *push-configuration* if the step from C_i to C_{i+1} is a push move. C_i is called *pop-configuration* if C_i is followed by a pop move. The *push-time* (or short *time*) of C_i is the number of push-configurations C_j before C_i (i.e., with $j < i$). (Thus a pop-configuration has the same push-time as the succeeding push-configuration.) We intend to describe a divide-and-conquer strategy for Comp(P, w).

Given an integer $e \geqslant 2$ (to be determined later from $|w|$ and the intended space complexity) we define a *section* as a consecutive part of Comp(P, w). All configurations with the same push-time form a *section of rank* 0. Thus a 0-section consists of a (maybe) empty sequence of pop moves followed by one push move. A *section of rank d* (or d-section) consists of e consecutive sections of rank $d-1$ and hence contains exactly e^d push moves. Thus the ith d-section contains exactly all configurations C with pushtime t, $(i-1) \cdot e^d \leqslant t < i \cdot e^d$.

The configuration with lowest stack height in a section (if there is more than one, the latest) is called the *representative* of that section. The representative of a d-section is a *d-configuration* (the 0-configurations are just the push-configurations). A configuration C_i is *visible from* C_j iff

(1) $i < j$,

(2) the stack height of all C_k, $i < k \leqslant j$, is greater than that of C_i.

The *cut* of C is the last configuration visible from C. (Thus the cut is the last push-configuration before C with stack-height $h-1$, where h is the stack height of C. If WX is the stack content of C, then the stack of the cut contains exactly W.)

A section S is called *current with respect to* C, if C is in S. S is *completed* if C is after S. A d-section S is *rightmost with respect to* C (related to time t_0), if the 0-section with time t_0 is in or before S and S is the last completed d-section whose representative is visible from C. In other words d-section S is rightmost with respect to C if S is completed at C and visible from C and there is no d-section S' which is to the right of S, and which is also completed at C and visible from C. (Thus the cut of C is the representative of the rightmost 0-section with respect to C.)

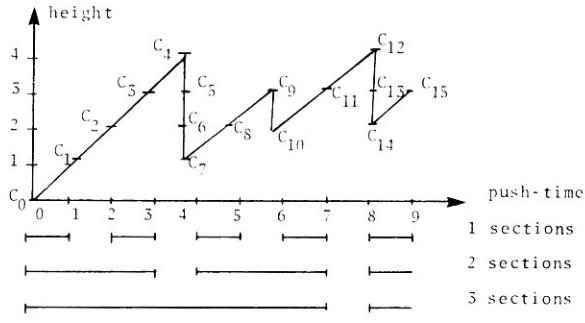

FIGURE 1

EXAMPLE. We illustrate these concepts with an example (see Fig. 1). Let $e = 2$.

0-sections: C_0, C_1, C_2, C_3, C_4–C_7, C_8, C_9–C_{10}, C_{11}, C_{12}–C_{14}, C_{15},
1-sections: C_0–C_1, C_2–C_3, C_4–C_8, C_9–C_{11}, C_{12}–C_{15},
2-sections: C_0–C_3, C_4–C_{11}, C_{12}–C_{15},
3-sections: C_0–C_{11}, C_{12}–C_{15};

C_7 is representative of 0-section C_4–C_7, 1-section C_4–C_8, 2-section C_4–C_{11}. It is not representative of 3-section C_0–C_{11}; C_0 is the representative of 3-section C_0–C_{11}. Configurations C_0, C_7, and C_{10} are visible from C_{11}. Let $C = C_{11}$. Then 1-sections C_0–C_1, C_2–C_3, C_4–C_8 are completed at C and 1-section C_9–C_{11} is current at C. Also the rightmost d-section with respect to C is C_9–C_{10} for $d = 0$, C_4–C_8 for $d = 1$, and C_0–C_3 for $d = 2$. There is no rightmost 3-section with respect to C related to time $t_0 = 0$.

Since a configuration in general requires space n, a space-efficient simulation cannot store configurations. For the simulation of a push-step we need only the configuration's *surface* (q, X, v) = (state, top of the stack, input position). After a pop step the new surface contains the stack symbol below the top, which is the stack symbol in the surface of the cut. If the cut (surface) is not stored, we must repeat a part of the computation up to the cut. Such *recomputations* can be nested. The key to our simulation is a strategy to remember some of the surfaces in order to avoid too many long recomputations.

In addition to the surfaces some information necessary for the recomputations is stored: a *marker* $M = (q, X, v, h, t, \mathrm{ct})$ is an extract of a configuration containing state, top symbol, input position, stack height, push-time, and push-time of the cut. If M is a marker then we use $q(M)$, $X(M)$, $v(M)$, $h(M)$, $t(M)$, $\mathrm{ct}(M)$ to denote the state, top symbol, input position, stack height, push-time and push-time of the cut of marker M.

In an obvious way we apply the terms computation, push-marker, pop-markers, section, rank, representative, visible, cut, current, etc. to markers. Just as for surfaces, the next marker M' can be obtained from M and the cut of M:

if M is a push-marker, then $q(M')$, $v(M')$, $X(M')$ according to the move of P, $h(M') = h(M) + 1$, $t(M') = t(M) + 1$, $\mathrm{ct}(M') = t(M)$,

if M is a pop-marker, then $q(M')$, $v(M')$ according to the move of P, $X(M')$ from the cut, $h(M') = h(M) - 1$, $t(M') = t(M)$, $\mathrm{ct}(M') = \mathrm{ct}$ of the cut.

(The stack of the configuration corresponding to M' is the same as the stack of the cut-configuration.)

3. The Basic Algorithm

For the simulation of a pop-step the stored markers are used in two different ways: If the cut is stored we can obtain the next marker from the preceding one and the cut. If not, the last stored marker serves as a starting point for the recomputation.

Our algorithm will have the following (storage) property that guarantees the correctness of the simulation.

Invariant. Consider a (re-) computation with starting time t_0 and end time t_1. Suppose M is the last computed marker. Let S be any d-section. Then the representative M' of S is in storage iff

(1) M' is visible from M and S is completed at M, and

(2) the enclosing $(d+1)$-section is current or rightmost wrt. M and not before t_0.

Example continued. Let $t_0 = 0$, $t_1 = 15$ and suppose that the marker corresponding to C_7 was computed last. Then we have:

(1) the representatives of 0-sections C_1, C_2, C_3,..., are not stored because they are not visible from M;

(2) *the representative of 0-section C_0 is stored because the enclosing* 1-section C_0–C_1 is rightmost with respect to C_6;

(3) the representative of 1-section C_0–C_1 (which is C_0) is stored because the enclosing 2-section C_0–C_3 is rightmost.

(4) the representative of 2-section C_0–C_3 (which is C_0) is stored because the enclosing 3-section C_0–C_{11} is current:

Thus we store	Enclosing $(d+1)$-section is rightmost	Current
0-representatives	C_0	
1-representatives	C_0	
2-representatives	C_0	
3-representatives		C_0

The cut of M is the last visible marker and thus is contained in the current 1-section or (if no marker of that section is visible from M) in the rightmost 1-section. Hence an algorithm satisfying the invariant correctly simulates Comp(P, w).

The invariant suggests the following storage structure: For every rank $d \geqslant 0$ (up to $\lfloor \log_e |w| \rfloor$) we provide two lists:

L_d^r for the d-markers of the rightmost $(d+1)$-section,

L_d^c for the d-markers of the current $(d+1)$-section.

The markers on the lists are ordered according to their time. L_d is the concatenation of L_d^r and L_d^c. If L is a list then bottom (L) is the marker with smallest time and top (L) is the marker with largest time on L.

LEMMA 1. *For every rank $d \geqslant 0$, every computation (starting time t_0), and every time in that computation:*

(a) *the markers on L_d, after t_0, are ordered according to strictly increasing height h,*

(b) *top L_d is representative of the rightmost d-section (related to t_0),*

(c) *bottom L_d^r = top L_{d+1},*

(d) *every list contains at most e markers.*

Proof. (a) All markers on L_d are visible. Thus a later marker has greater height. Therefore increasing time implies increasing height.

(b) Obvious.

(c) If $L_d^r \neq \emptyset$, bottom L_d = bottom L_d^r is the lowest d-marker of the rightmost $(d+1)$-section and hence its representative is contained in the current or rightmost $(d+2)$-section. Since it is the latest visible $(d+1)$-marker in a completed $(d+1)$-section, it is top L_{d+1}. If $L_d^r = \emptyset$, no marker of a completed $(d+1)$-section is visible and hence $L_{d+1} = \emptyset$.

(d) By the definition of $(d+1)$-section, it contains exactly e d-sections. Hence it contains at most e visible d-markers. ∎

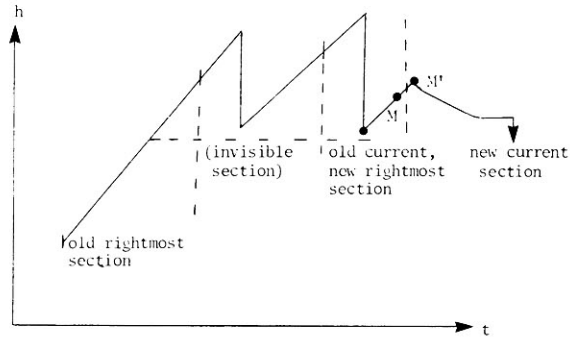

FIGURE 2

Now it is possible to derive the algorithm from the invariant. The current marker is not the representative of a complete 0-section. Thus we store it in a separate register R.

(a) Suppose R contains a push-marker M (see Fig. 2). Then it is the last marker (and representative) of a 0-section which is now completed. Thus R has to be stored on L_0. The next marker M' can be computed from M and is stored in R. It is possible that the next marker belongs not only to a new 0-section but also to a new d-section for some $d > 0$. In this case the old current section is now completed and hence rightmost. Suppose d is maximal, such that a d-section is complted, $d > 0$ (i.e., $t(M') \equiv 0 \pmod{e^d}$, $t(M') \not\equiv 0 \pmod{e^{d+1}}$). Then, by the invariant, for every $i < d$ the i-markers of the old rightmost $(i+1)$-section have to be deleted. The representative of the new rightmost $(i+1)$-section has to be added to L^c_{i+1}. Thus the following instructions simulate a push-step:

$L^c_0 := L^c_0$ *concat* R;
compute new R from R;
if $t(R) \equiv 0 \pmod{e}$
then let d be maximal with $t(R) \equiv 0 \pmod{e^d}$
  for i *from* 0 *to* $d-1$
  do $L^r_i := L^c_i$; $L^c_i :=$ empty;
    $L^c_{i+1} := L^c_{i+1}$ *concat* bottom L^r_i
  od

Example continued. C_7 is a push-marker. The push-move from C_7 to C_8 completes 0-section C_4–C_7 (with representative C_7). It does not complete a 1-section. Hence the marker corresponding to C_8 is stored in R and lists L^r_d, L^c_d are changed to

d	L_d^r	L_d^c
0	C_0	C_7
1	C_0	
2	C_0	
3		C_0

C_8 is a push-marker. The push-move from C_8 to C_9 completes 0-section C_8 and 1-section C_4–C_8. It does not complete a 2-section. Hence the marker corresponding to C_9 is stored in R and lists L_d^r, L_d^c are changed to

d	L_d^r	L_d^c
0	C_7, C_8	
1	C_0	C_7
2	C_0	
3		C_0

(b) Suppose R contains a pop-marker M. The next marker M' can be computed from M and $\bar{M}$, the cut of M, which is the last push-marker visible from M and hence stored at the top of L_0.

Since $t(M') = t(M)$, no section becomes completed, but $\bar{M}$ becomes invisible and has to be deleted from all the lists it appears on. (First we delete it from L_0. If L_0 is now empty and $L_1 \neq \emptyset$, then, by Lemma 1(c), we have also to delete top L_1. If L_1 is empty we delete top L_2etc.) Furthermore, if $\bar{M}$ is the representative of a completed d-section ($d > 0$), then this (old rightmost) section is no longer rightmost. Thus, by the invariant, the $(d-1)$-markers of the new rightmost section have to be recomputed.

Example continued. C_9 is a pop configuration. We can compute the marker of C_{10} from the markers of C_9 and C_8. Store C_{10}'s marker in R and change lists L_d^r, L_d^c to

d	L_d^r	L_d^c
0	C_7	
1	C_0	C_7
2	C_0	
3		C_0

C_{10} is a push-configuration and so is C_{11}. The move from C_{10} to C_{11} completes a 0-section and the move from C_{11} to C_{12} completes 0-section C_{11},

1-section C_9–C_{11}, 2-section C_4–C_{11} and 3-section C_0–C_{11}. Lists L_d^r, L_d^c are changed to

d	L_d^r	L_d^c
0	C_{10}, C_{11}	
1	C_7, C_{10}	
2	C_0, C_7	
3	C_0	

Also the marker corresponding to C_{12} is stored in R; C_{12} is a pop-configuration. We remove C_{11} from L_0^r and compute C_{13}'s marker from R and C_{11}. Next we compute the marker corresponding to C_{14} from R (which contains C_{13}) and C_{10}. Also we remove C_{10} from L_0^r and L_1^r. At this point no marker in 1-section C_9–C_{11} is visible any longer and hence 1-section C_9–C_{11} is not rightmost any longer. Rather, 1-section C_4–C_8 becomes rightmost.

Let d be maximal, such that $\bar{M}$ is a d-marker (see Fig. 3). Then $\bar{M}$ is on L_d, but by Lemma 1(c), $\bar{M}$ is not bottom L_d. Thus d is the highest rank such that $\bar{M}$ is on L_d, and the lowest rank such that, after deleting $\bar{M}$, L_d is not empty. The recomputation that restores the markers of the rightmost i-sections ($i \leqslant d$) starts from M_0, the new top-marker of L_d, which is the representative of the d-section containing the $\mathrm{ct}(M') = \mathrm{ct}(\bar{M})$. This *recomputation* has *rank d.*

The following instructions simulate a pop-step:

compute the new marker from R and top L_0 and store it in R;
let d be minimal such that L_d contains more than one marker;
for $i = 0$ *to* d *do* delete top L_i;
if $d > 0$ *then* perform a recomputation of rank d starting from top L_d up to the cut of M' using the algorithm recursively
fi

In our example we have $M_0 = C_7$ and $\mathrm{ct}(M') = C_7$. The recomputation from C_7 to C_7 is thus trivial; we only have to store the marker corresponding to C_7 in a copy of register R.

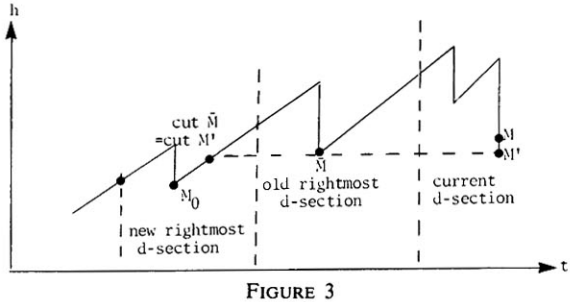

FIGURE 3

(c) The main computation stops, if no next step exists. A recomputation stops, if the cut of M' is computed and stored in R. Then the cut is stored on L_0 and all current sections of the recomputation are completed. If m is the rank of the recomputation, the current sections of rank 0, 1,..., $m-1$ have to be completed before we resume the calling procedure at M' (their sections are rightmost M'):

$L_0 := L_0$ *concat* R;
for $i := 0$ *to* $m-1$ *do if* $i < m-1$ *then*
$L^c_{i+1} := L^c_{i+1}$ *concat* bottom L^c_i *fi*
$L^r_i := L^c_i$; $L^c_i :=$ empty
od

Example continued. In our example, the recomputation stops with C_7 in register R. We can now return to the main computation. This will complete 0-section C_4–C_7. Hence lists L^r_d, L^c_d are changed to

d	L^r_d	L_d
0	C_7	
1	C_7	
2	C_0, C_7	
3	C_0	

The complete procedure consists of these three steps. It uses the parameters M (=starting marker), m (=rank of recomputation), and up (=time of latest marker to be computed).

The main procedure is:

begin $M_0 := (q_0, X_0, 1, 1, 0, 0)$;
call sim(M_0, ∞, ∞)
end

The simulation is done in a recursive procedure sim.

procedure sim(M, m, up)
begin $R := M$;
while $t(R) < up$
do co the invariant refers to this point;
if R is a push-marker
then $L^c_0 := L^c_0$ *concat* R;
compute new R from R;
if $t(R) \equiv 0 \pmod{e}$
then let d be maximal with $t(R) \equiv 0 \pmod{e^d}$;
for i *from* 0 *to* $d-1$
do $L^c_{i+1} := L^c_{i+1}$ *concat* bottom L^c_i;
$L^r_i := L^c_i$; $L^c_i :=$ empty
od
fi;

push-step: end of a d-section

```
                    if R is a pop-marker
                    then do compute new R from R and top L_0;
                            let d be minimal with |L_d| ≥ 2;
                            for i from 0 to d do delete top L_i od;
pop-step            recomputation { if d > 0 then call sim(top L_d, d, ct(R)) fi
                            od
                    fi;
end of the          if no step follows R
simulation          then if q(R) is accepting then accept else reject fi
                    fi
                od;
completion of a     L_0^c := L_0^c concat R;
recomputation       for i from 0 to m − 1
                    do if i < m − 1 then L_{i+1}^c := L_{i+1}^c concat bottom L_i^c fi;
                       L_i^r := L_i^c; L_i^c := empty
                    od
                end
```

The next lemma gives the space complexity of the algorithm.

LEMMA 2. *Suppose* $n = |w|$, $r = \lfloor \log_e n \rfloor$, $e \geqslant 2$.

(a) *Only the lists* $L_0, \ldots, L_r$ *are used.*

(b) *The depth of recursion of* sim *is at most* r.

(c) *The space complexity of the algorithm is* $O(e \cdot r \cdot \log n)$.

Proof. (a) Comp(P, w) contains at most $|w|$ push configurations. Since $e^r \leqslant |w| < e^{r+1}$, no section of rank $r+1$ is completed and hence $L_{r+1}, \ldots,$ are not used.

(b) If a marker of rank $d \leqslant r$ becomes invisible, a section of rank d is recomputed, but not the representative of this section (which is the starting marker of the recomputation). Thus only markers of rank $d' < d$ are recomputed and during this recomputation only markers of rank at most $d-1$ can become invisible.

(c) A marker is stored on space $O(\log n)$. Every list L_i^r, L_i^c contains up to e markers. In addition, every nested call of sim requires local space $O(\log n)$. Thus $S(n) = O(2e \cdot r \cdot \log n + r \cdot \log n) = O(e \cdot r \cdot \log n)$.

The time complexity is derived from a bound on the number of sections to be computed (a d-section is computed if its representative is set on L_d^c or if it is current at the end of the simulation).

LEMMA 3. *Suppose* $\bar{e} = \lceil n/e^r \rceil$, n, r *as in Lemma* 2.

(a) *For every rank* $d \leqslant r$ *at most* $2 \cdot \bar{e} \cdot (2e)^{r-d}$ *sections of rank* d *are computed (or recomputed).*

(b) *The number of* PDA*-steps computed during the simulation is at most* $O(n \cdot 2^r)$.

(c) *The time complexity on a logarithmic cost* RAM (*Cook*, 1972) *is* $O(n \cdot 2^r \cdot \log n)$. *This time bound also applies to a Turing machine with random access input* (*that is, a multitape Turing machine with a special index tape on which the position of the next input symbol to be accessed is written*).

Proof. (a) By induction on $(r-d)$. Every marker that is set onto some list L_i may become invisible and give rise to one recomputation of rank i.

$(d=r)$ The computation contains at most $\bar{e}$ sections of rank r. Thus (including the recomputations of rank r, when their representatives become invisible) at most $2\bar{e}$ sections of rank r are computed.

$(d-1)$ Any section of rank d contains e sections of rank $d-1$. Their representatives are laid down on L_{d-1} and may give rise to a recomputation of rank $d-1$. Since by the induction hypothesis the number of d-sections is at most $2\bar{e}(2e)^{r-d}$, at most $2e \cdot 2\bar{e}(2e)^{r-d} = 2\bar{e}(2e)^{r-(d-1)}$ $(d-1)$-sections are computed.

(b) By (a) the number of simulated push-steps (=number of 0-sections) is at most $2\bar{e}(2e)^r = \bar{e} \cdot e^r \cdot 2^{r+1} = O(n \cdot 2^r)$. The number of pop-steps cannot be greater than the number of push steps.

(c) The only statements that cost more than $O(\log n)$ are the *for* statements. Their cost depends on the rank d of the section that is completed or of the recomputation that is started and is $O(d \cdot e \cdot \log n)$. Thus the total costs for these statements or $O(\sum_{d=1}^{r} \bar{e} \cdot (2e)^{r-d} \cdot e \cdot d \cdot \log n) = O(2^r \cdot n \cdot \log n \cdot \sum_{d=1}^{r} e \cdot d/(2e)^d) = O(n \cdot 2^r \cdot \log n)$. ∎

4. Improved Algorithm

The improved algorithm differs from the basic algorithm mainly in the treatment of markers of rank 0. We will show how to treat rank 0 markers such that:

(1) A rank 0 marker takes space $O(1)$ instead of space $O(\log n)$ as for the other markers. (This will allow us to increase the length of 1-sections without destroying the space bound, thereby reducing the number of recursive calls.)

(2) A 1-section can be simulated in time linear in its length instead of time $O(e \log n)$ as in the basic algorithms. Since most of the time is spent in simulating 1-sections this will, together with the observation in (1), improve the time bound.

The details are as follows:

(1) For markers of rank 0 (except for bottom L_0^r and the current register R) only the stack symbol is stored. Moreover, a 1-section consists of s (instead of e) consecutive 0-sections. Here s is the intended space bound of the algorithm. For $d \geqslant 2$, a d-section consists of $e(d-1)$-sections. Parameter e is chosen below as $e = [s(n)/\log^2 n]$. Note that the markers of rank 0 form a contiguous top part of the pushdown store.

(2) The current register R holds the current values of q, X, v, and t. Note that ct is not stored. For $C :=$ bottom L_0^c we store the state q and information about $v(C)$ and $t(C)$. Instead of storing $v(C)$ and $t(C)$ directly we store $\Delta v = v = v(C)$ and $\Delta t = t - t(C)$, and only compute $v(C)$ and $t(C)$ when a 1-section is completed.

Storing $v(C)$ and $t(C)$ implicity is motivated as follows. Note that $v(C)$ and $t(C)$ can change frequently during the simulation of a 1-section, namely whenever L_0^c becomes empty. If $v(C)$ and $t(C)$ were stored explicitly then every such change would cost $O(\log n)$ and 1-sections could not possibly be simulated in linear time. With the implicit storage scheme v and t are only increased, and Δv and Δt are only increased and sometimes reset to zero. The following fact is well known.

FACT. *Let $N \in \mathbb{N}$. Counting from 0 to N in binary takes time $O(N)$ on a* TM.

Thus the implicit storage scheme allows us to handle quantities v, t, Δv, and Δt in average time $O(1)$ per simulated move.

(3) Whenever a 1-section is completed we need to flesh out leftmost L_0^c to a complete marker C, i.e., we need to compute $q(C)$, $X(C)$, $v(C)$, $t(C)$, $h(C)$, and ct(C). Quantities $q(C)$, $X(C)$, $v(C)$, and $t(C)$ are readily available in time $O(\log n)$. Furthermore, $h(C)$ can be computed as $h(\text{top } L_1) + |L_0^r|$ in time $O(s + \log n)$. Note that $|L_0^r| \leqslant s$. However, ct(C) is not available and cannot be computed. For this reason we redefine the cut time of a marker as

$$\text{ct}(M) = \text{push-time of rightmost 1-marker preceding } M.$$

With this new definition of cut-time we can compute ct(C) as t (top L_1). The discussion above is captured in the following definition of function leftmost (M is the starting marker)

$$\text{leftmost}(L_i^c) := \begin{cases} \text{bottom } L_i^c & \text{if } i > 0 \\ (q(C),\ X(C),\ v - \Delta v,\ h(\text{top } L_1) + |L_0^r|,\ t - \Delta t,\ t(\text{top } L_1)) & \\ \qquad \text{if } i = 0 \text{ and } L_1 \neq \text{empty} & \\ M & \text{if } i = 0 \text{ and } L_1 = \text{empty}. \end{cases}$$

The modification in the definition of cut-time forces us to look for a new criterion for the end of a recomputation. If the cut marker (i.e., the 1-marker with time up) is computed, continue the recomputation *to the end* of the 1-section (in the basic algorithm we stopped when the true cut marker was reached). Then delete the part of L_0 that is invisible from the current marker of the calling procedure. To this end we add its height to the parameter list of sim_s.

The new main procedure is

```
begin M_0 := (q_0, X_0, 1, 1, 0, 0);
      call sim_s(M_0, ∞, ∞, ∞)
end

procedure sim_s(M, up, m, h_0)
begin q := q(M); X := X(M); v := v(M); t := t(M); rt := t mod s;
      C := (q, X); Δv := 0; Δt := 0;
      co C contains q, X of the rightmost lowest marker M_1 of the current
           1-section, Δv = v − v(M_1), Δt = t − t(M_1);
      while t < up or rt < s − 1 or a pop follows
      do if a push follows
         then if L_0^c = empty
              then C := (q, X); Δv := 0; Δt := 0
              fi;
              L_0^c := L_0^c concat X;
              compute new q, X, v, Δv from q, X, v;
              t := t + 1; Δt := Δt + 1; rt := rt + 1;
              if rt = s
              then let d be maximal with t ≡ 0 (mod s · e^d);
                   for i from 0 to d do L_{i+1}^c := L_{i+1}^c concat
                        leftmost L_i^c;
                                   L_i^r := L_i^c; L_i^c := empty
                   od;
                   rt := 0
              fi
         fi;
         if pop follows
         then compute new q, v, Δv from q, X, v;
              X := top L_0; delete top L_0;
              if L_0 = empty
              then h := h(top L_1); up' := ct(top L_1);
                   let d be minimal with |L_d| ⩾ 2;
                   for i from 1 to d do delete top L_i od;
                   call sim_s(top L_d, up', d, h)
              if
         fi;
         if no step follows
         then if q is accepting then accept else reject fi
         fi
      od;
```

push-step: end of a $(d+1)$-section

pop-step: start of a recomp.

end of the simulation

```
                    L_0 := L_0 concat X;
                    for i from 0 to m − 1 do if i < m − 1
                                          then L^c_{i+1} := L^c_{i+1} concat
                                            leftmost L^c_i
end of a            fi;
recomp.             L^r_i := L^c_i; L^c_i := empty
                    od;
                    for h from h(top L_1) + |L_0| − 1 to h_0 step −1
                    do delete top
                    L_0 od
end
```

The same argument as for Lemma 3 shows that $O((n/s) \cdot 2^r)$ 1-sections are computed $(r = \lfloor \log_e(n/s) \rfloor + 1)$. The lists $L_1, \ldots, L_r$ are updated only before the beginning of a recomputation and at the end of a 1-section; this updating costs $O(s + e \cdot r \cdot \log n)$ steps. The counters t, rt, Δt, Δv are only increased and the costs for updating C are $O(1)$; therefore the simulation of the push steps of a 1-section costs $O(s)$ steps. If we add the costs of the pop-steps to the costs of the corresponding push-steps, the total costs are $O((n/s) \cdot 2^r \cdot (s + e \cdot r \cdot \log n))$. Thus we have

LEMMA 4. *Suppose* $r = \lfloor \log_e(n/s) \rfloor + 1$, $e \cdot r \cdot \log n \leqslant s$. *Then the space and time complexities of* sim_s *on a logarithmic cost* RAM *or a* TM *with random access input are*

$$\text{Space}(n) = O(s), \qquad \text{Time}(n) = O(n \cdot 2^r). \quad \blacksquare$$

We call a function s *acceptable* if $\lfloor s(n) \rfloor$ is tape constructable in time $O(n)$, $n \geqslant s(n) \geqslant 2 \log^2 n$ for almost all n, and s is nondecreasing. (For example $2 \log^2 n$, $n^{1/\log \log n}$, n^ε are acceptable.)

THEOREM 1. *If* s *is acceptable, then every* DCFL *can be recognized on a multitape* TM *with random-access input simultaneously in time* $O(n \cdot n^{1/(\log s(n) - 2 \log \log n)})$ *and space* $O(s(n))$.

Proof. Choose $e = \lceil s(n)/\log^2 n \rceil$, $r = \lfloor \log_e(n/s(n)) \rfloor + 1$. Then

$$e \cdot r \cdot \log n \leqslant e \cdot \log^2 n \leqslant 2s(n)$$

and

$$\begin{aligned} r - 1 &\leqslant \log_e(n/s(n)) = (\log n - \log s(n))/\log e \\ &\leqslant (\log n - \log s(n))/(\log s(n) - 2 \log \log n) \\ &< \log n/(\log s(n) - 2 \log \log n). \end{aligned}$$

By Lemma 4, $\text{Space}(n) = O(s(n))$, $\text{Time}(n) = O(2^r \cdot n)$, and

$$2^r < 2 \cdot 2^{\log n/(\log s(n) - 2 \log\log n)}$$
$$= 2 \cdot n^{1/(\log s(n) - 2 \log\log n)}. \quad \blacksquare$$

EXAMPLE 1.

$$\text{Space}(n) = k \cdot \log^2 n, \qquad k \geqslant 2$$
$$e = k, r = \lfloor \log n/\log k \rfloor$$
$$\text{Time}(n) = O(n^{1+1/\log k})$$

For any $\varepsilon > 0$, DCFL $\subseteq$ Time–Space$(n^{1+\varepsilon}, \log^2 n)$.

EXAMPLE 2.

$$\text{Space}(n) = n^{1/k},$$
$$e = \lceil n^{1/k}/\log^2 n \rceil,$$
$$r = \log n^{(k-1)/k}/\lceil \log(n^{1/k}/\log^2 n) \rceil + 1 \leqslant k + 1,$$
$$\text{Time}(n) = O(2^k \cdot n).$$

For any $\varepsilon > 0$, DCFL $\subseteq$ Time–Space(n, n^{ε}).

EXAMPLE 3.

$$\text{Space}(n) = 2^{\sqrt{\log n}} \cdot \log^2 n$$
$$e = 2^{\lceil \sqrt{\log n} \rceil} = n^{\lceil 1/\sqrt{\log n} \rceil}, \qquad r = \lfloor \sqrt{\log n} \rfloor + 1.$$
$$\text{Time}(n) = O(n \cdot 2^r) = O(n^{1+1/\sqrt{\log n}}),$$

$\text{Time}(n) \cdot \text{Space}(n) = O(n^{1+2/\sqrt{\log n}} \cdot \log^2 n)$ (minimal space-time product).

In the following we consider ordinary multitape Turing machines with a 2-way input tape. Lemma 5 gives a lower bound for this case that is much greater than the upper bound of Theorem 1.

LEMMA 5. *If* DCFL $\subseteq$ Time–Space$(t(n), s(n))$, *then*

$$n^2 = O(t(n) \cdot s(n)).$$

Proof. A standard argument on crossing sequences shows for the language $\{wc\bar{w} \mid w \in \{a, b\}^*\}$ and one-tape TMs, $\text{Time}(n) \geqslant c \cdot n^2/\log |Q|$, $c > 0$, where Q is the set of states. Thus for a multitape TM and $\log n \leqslant$

$S(n) \leqslant n$, $T(n) \geqslant c_1 \cdot n^2/\log(|Q| \cdot c_2^{S(n)}) \geqslant c \cdot n^2/S(n)$, $c > 0$. (See Cobham, 1966.) ∎

On a multitape TM, algorithm sim_s takes much time for the moves of the input head at the beginning and the end of a recomputation. This time is estimated in the next lemma.

LEMMA 6. *Suppose $e > 2$. Then during a simulation, the number of moves of the input head is at most $O(n^2/s)$.*

Proof. During a computation of n push-steps, at most n/s markers of rank 1 or greater are computed, not counting recomputations. Thus at most n/s recomputations are caused by this computation. Before the beginning of a recomputation the input head is moved to the input position of the starting marker and at the end it is moved back from the input position of the cut to the old one. Thus for every call of sim_s up to $2n$ moves have to be done (not counting the moves inside the recursive call). Thus the number of these moves is $2n^2/s$. The same argument yields $O(l^2/s)$ moves for every recomputation of length l.

Suppose $1 < \bar{e} = \lceil n/(s \cdot e^{r-1}) \rceil \leqslant e$. The same argument as for Lemma 3(a) shows that at most $\bar{e}(2e)^{r-d}$ markers of rank d are set on L_d ($d \geqslant 1$). Any of these may cause a recomputation of rank d and length $s \cdot e^{d-1} \leqslant 2n/(\bar{e} \cdot e^{r-d})$. Thus the number of all input moves is

$$\frac{2n^2}{s} + \sum_{d=1}^{r} \left[\bar{e}(2e)^{r-d} \left(\frac{2n}{\bar{e} \cdot e^{r-d}} \right)^2 \Big/ s \right] = \frac{n^2}{s} \left(2 + \frac{4}{\bar{e}} \cdot \sum_{d=1}^{r} \left(\frac{2}{e} \right)^{r-d} \right)$$

$$= O(n^2/s). \quad ∎$$

THEOREM 2. *Suppose s is acceptable. Then every* DCFL *can be recognized on a multitape* TM *simultaneously in time $O(n^2/s(n))$ and space $O(s(n))$ and this time bound is optimal up to a constant factor.* ∎

RECEIVED: June 15, 1982; ACCEPTED: April 6, 1983

REFERENCES

BRAUNMÜHL B. VON AND VERBEEK, R. (1980), A recognition algorithm for deterministic CFLs optimal in time and space, *in* "Proceedings, 21st Annual Symposium on Foundations of Computer Science," pp. 411–420.

COBHAM, A. (1966), The recognition problem for the set of perfect squares, *in* "IEEE Conference Record of 1966 Seventh Annual Symposium on Switching and Automata Theory," pp. 78–87.

COOK, S. (1972), Linear time simulation of deterministic two-way pushdown automata, *Inform. Process. Lett.* **71**, 85–80.

COOK, S. (1979), Deterministic CFLs are accepted simultaneously in polynomial time and log squared space, *in* "Proceedings, 11th Annual ACM Symposium on Theory of Computing," pp. 338–345.

COOK, S. (1981), Towards a complexity theory of synchronous parallel computation, L'Enseignement Mathématique, Vol. XXVII, fasc. 1–2, pp. 99–124.

DYMOND, P., AND COOK, S. (1980), Hardware complexity and parallel computation, *in* "Proceedings, 21st Annual Symposium on Foundations of Computer Science," pp. 360–372.

HONG, J. W. (1980), On similarity and duality of computation, *in* "Proceedings, 21st Annual Symposium on Foundations of Computer Science," pp. 348–359.

HOPCROFT, J. AND ULLMAN, J. (1979), "Introduction to Automata Theory, Languages, and Computation," Addison–Wesley, Reading, Mass.

LEWIS, P., STEARNS, R., AND HARTMANIS, J. (1965), Memory bounds for recognition of context-free and context-sensitive languages "IEEE Conference Record on Switching Circuit Theory and Logical Design," pp. 191–202.

MEHLHORN, K. (1980), Pebbling mountain ranges and its application to DCFL recognition, *in* "Proceedings, 7th International Colloquium on Automata, Languages and Programming," Lecture Notes in Computer Science, Vol. 85, pp. 422–435.

PIPPENGER, N. (1979), On simultaneous resource bounds (preliminary version), *in* "Proceedings, 20th Annual Symposium on Foundations of Computer Science," pp. 307–311.

RUZZO, W. L. (1979), On uniform circuit complexity (extended abstract), *in* "Proceedings, 20th Annual Symposium on Foundations of Computer Science," pp. 312–318.

SUDBOROUGH, I. H. (1978), On the tape complexity of deterministic context-free languages, *J. Assoc. Comput. Mach.* **25** (3), 405–414.

SUDBOROUGH, I. H. (1980), Efficent algorithms for path system problems and applications to alternating time-space complexity classes, *in* "Proceedings, 21st Annual Symposium on Foundations of Computer Science," pp. 62–73.

VERBEEK, R. (1981), Time–space tradeoffs for general recursion, *in* "Proceedings, 22nd Annual Symposium on Foundations of Computer Science," pp. 228–234.