Annals of Discrete Mathematics 24 (1985) 1-20



# Input Driven Languages are Recognized in $\log n$ Space

by
Burchard von Braunmühl
Rutger Verbeek
University of Bonn

Abstract

We show that the class of input driven languages is contained in $\log n$ space, an improvement over the previously known bound of $\log^2 n/\log\log n$ space [Me 8o]. We also show the power of deterministic and nondeterministic input driven automata is the same and that these automata can recognize e.g. the parenthesis languages and the leftmost Szilard languages (cf. [Me 75], [Ly 76], [Ig 77]).

## 1. Introduction

Since the famous algorithm of Lewis, Hartmanis and Stearns [LSH 65] many attempts have been made to determine the exact lower and upper bounds for the recognition of context free languages. It is quite improbable that the $\log^2 n$ upper bound can be improved for general CFLs because every improvement would improve the bound for the deterministic simulation of nondeterministic Turing machines [Su 75]. Even for deterministic CFLs no better space bound than $\log^2 n$ is known. Only the simultaneous time-space bound for DCFLs is much better than in the general case of CFLs: using a black pebbling strategy for the computation graphs of deterministic pushdown automata Cook [Co 79] has shown that DCFLs can be recognized using $\log n$ pebbles (i.e. in $\log^2 n$ space) and polynomial time simultaneously. The strategy was improved by the authors [BV 8o],[BCMV 83] and shown to be almost optimal (as pebbling strategy) [Ve 81], [Ve 83]. On the other hand several important subclasses of DCFLs are known to be recognized in $\log n$ space (e.g. Dyck sets [RS 72], parenthesis languages [Me 75],[Ly 76] and leftmost Szilard languages [Ig 77]). All these examples are shown to be included in, or are easily (i.e. by a homomorphism) transformable to the class of input driven languages (defined in [Me 8o]), which is already known to require a bit less than $\log^2 n$ space: according to Mehlhorn [Me 8o]

there is a black pebbling strategy with $(\log n/\log\log n)^2$ pebbles where every pebble requires only $O(\log\log n)$ space.

In this paper we present a $\log n$ space algorithm for the recognition of input driven languages. Our algorithm uses a black and white pebbling strategy [CS 76] for the computation graphs, where we make use of the fact, that the structure of the computation graph of an input driven PDA can be computed by a counter automaton (and hence in $\log n$ space). The pebbles are distributed on the graph in such a way that their positions need not to be stored but can be recomputed. In this way our algorithm uses $O(\log n)$ pebbles, where each pebble requires only bounded space. Our approach gives a simple proof for the space complexity of a large subclass of DCFLs. In terms of pebbling complexity the strategy is optimal: it pebbles every n-node computation graph using $O(\log n)$ pebbles in $n$ steps and there are computation graphs which require $\Omega(\log n)$ black and white pebbles [Ve 83].

Since our aim is to prove a space bound for some DCFLs we shall present the algorithm directly as simulation algorithm for input driven PDAs rather than as black and white pebble game.

Unfortunately, there exist some classes of DCFLs that are contained in DSPACE($\log n$) but are not obviously transformable to input driven languages (certainly they all are $\log n$ - space reducible to any other nontrivial language): counter languages (where the space bound is obvious), the languages recognized by finite minimal stacking PDAs [Ig 78] (which can be proved to be accepted in $\log n$ space via a simple black pebbling strategy with a bounded number of pebbles) and the two sided Dyck sets [LZ 77], which is the only known example of a non input driven DCFL in DSPACE($\log n$) for which the space bound is not obvious.
In section 2, we show that parentheses languages are input driven, leftmost Szilard languages are homomorphic transformable to input driven languages and the (nondeterministic) input driven languages are all deterministic (i.e. acceptable by deterministic input driven PDAs) which again proves that parentheses languages are DCFLs.
In section 3, we present the simulation algorithm for input driven languages which uses $\log n$ space and $n^2$ time on a Turing machine.

## 2. Properties of input driven languages

We call a (deterministic or nondeterministic) real-time pushdown automaton (PDA) "input driven", if it reacts to some input symbol always with a push (the stack grows) or always with a pop (the stack falls). The moves of the stack are controlled only by the actual input symbol. The sequence of push and pop, (i.e. the graph of the stack height function $h(t)$ $(1 \leq t \leq n)$) is determined by the input and is readable from it.

Definition

A real-time pushdown automaton (PDA) is denoted by

$$P = (\Sigma, \Gamma, Q, \#, q_o, Q_f, \delta)$$

where $\Sigma, \Gamma, Q$ are finite disjoint sets of input symbols, pushdown symbols and states, $\# \in \Gamma$ (bottom symbol), $Q_f \subseteq Q$ (accepting states), $q_o \in Q$ (initial state), and $\delta$ is a finite subset of $Q \times \Gamma \times \Sigma \times Q \times \Gamma^*$.

P is input driven, (idPDA) if $\Sigma$ is the disjoint union of $\Sigma_o, \Sigma_1, \Sigma_2$ and $\delta \subseteq \bigcup_{i=o}^{2} Q \times \Gamma \times \Sigma_i \times Q \times \Gamma^i$. ($x \in \Sigma_o$ means pop, $x \in \Sigma_2$ means replace and push)

A language $L \subseteq \Sigma^*$ is input driven (idCFL), if it is recognized by some input driven PDA.

A configuration is a pair $(q,W)$ containing the state $q$ and the content of the pushdown store $W$. For every $w \in \Sigma^*$, $q,q' \in Q$, $X \in \Gamma$, $W,W' \in \Gamma^*$, we define $\vdash_w$ inductively by

$(q,WX) \vdash_e (q,WX)$,

$(q,WX) \vdash_{wx} (q',W')$ iff there are $W_1, W_2 \in \Gamma^*$, $Y \in \Gamma$, $q_1 \in Q$, such that $(q,WX) \vdash_w (q_1, W_1Y)$, $(q_1,Y,x,q',W_2) \in \delta$ and $W' = W_1W_2$.

If $x_1 \ldots x_n$ is the input word, we define the stack height at time $t$ (that is just before step $t$) by

$h(t) = |W|$ if $(q,\#) \vdash_{x_1 \ldots x_{t-1}} (q',W)$ for $1 \leq t \leq n+1$.

($x_i x_{i-1}$ is the empty word).

Many subclasses of DCFLs that are known to be recognizable in $\log n$ space are input driven or reducible to input driven languages by a straightforward reduction. We illustrate this with two examples.

Definition

A context free grammar is called a parenthesis grammar if every rule is of the form

$A \to (W)$

with one sort of brackets "(" and ")" that are terminals and where $W$ does not contain any bracket.
A CFL is called a parenthesis language if it is generated by some parenthesis grammar. □

Theorem 1

Every parenthesis language is input-driven.

Proof

The idPDA P accepts some parenthesis language $L \subseteq \Sigma^*$ using a bounded buffer containing a (suffix of the) right side of a rule as state and stack symbol. P performs a nondeterministic top-down analysis of the input. When a "(" is scanned P replaces the first symbol of the buffer by the right side of a corresponding grammar rule and pushs the rest of the buffer. When ")" is scanned the buffer must be empty and the top is transferred into the buffer. When a non bracket symbol is read it is compared with the first symbol of the buffer. P accepts with empty stack.

Suppose a string V (part of a right side of a rule) is encoded by the symbol $\underline{V}$. P starts in state $\underline{S}$ and uses the following table:

| q,X,a | q',W | remarks |
|---|---|---|
| $\underline{AV}$, X, ( | $\underline{U}$, $\underline{XV}$ | for all rules $A \to (U)$, push-step |
| $\underline{e}$, X, ) | X, e | pop-step |
| $\underline{aV}$, X, a | $\underline{V}$, X | comparison of the input |
| $\underline{e}$, X, ( | reject | |
| $\underline{aV}$, X, ( | reject | terminal a |
| $\underline{V}$, X, ) | reject | $V \neq e$ (e = empty word) |
| $\underline{bV}$, X, a | reject | $b \neq a$ |

□

The second example is the class of Szilard languages for leftmost derivations.

Definition

Suppose $G = (N,T,S,\{P_1,\ldots,P_k\})$ is a context free grammar. Associate with every leftmost derivation of $G$ the sequence of numbers of rules that are applied during the derivation. The language of all such sequences (associated with legal derivations) is called leftmost Szilard language of $G$ ($LS(G)$).

Theorem 2

(1) If $G$ is a CFG and the number of nonterminals on the right hand sides of the rules is at most 2, then $LS(G)$ is input-driven (recognized even by a deterministic one-state id PDA).

(2) For every CFG $G$ there is a homomorphism $h$, such that $h(LS(G))$ is input driven.

Proof

(1) The table for the (deterministic) idPDA recognizing $LS(G)$ is the following:

| q,X,a | q',W | | | |
|---|---|---|---|---|
| q,A,i | q,e | if $P_i = (A \to x)$ | and | $x \in T^*$ |
| q,A,i | q,B | if $P_i = (A \to xBy)$ | and | $x,y \in T^*$ |
| q,A,i | q,CB | if $P_i = (A \to xByCz)$ | and | $x,y,z \in T^*$ |

reject in all other cases

Obviously, the top always contains the leftmost nonterminal of the derived string. If the stack becomes empty, a terminal string is derived. Thus the PDA accepts $LS(G)$ with empty stack.

(2) Since in our model of idPDAs the stack height can grow by at most 1, in each step we must pad the input in order to enable the PDA to store all new nonterminals. This is done by the following simple homomorphism:

$h(i) = i\#^{l(i)}$, where $l(i)$ is the number of nonterminals on the right side of $P_i$.

Now a number i in the input causes a pop-step, that stores the right side of $P_i$ in the state, and the #'s cause a series of push-steps that transfers the contents of the state symbol by symbol onto the stack (beginning with the rightmost symbol). It is also possible to obtain a deterministic one-state machine, if the right side of $P_i$ is written onto the top.

□

Our next theorem shows that the power of deterministic and nondeterministic input driven automata is the same. The algorithm in section 3 is nevertheless described for the nondeterministic case, since the deterministic simulation below increases the number of states and pushdown symbols exponentially. Also the proof of Theorem 1 shows the value of the nondeterministic version: it would be very tedious to show directly that parenthesis languages are deterministic input driven.

Theorem 3

If P is a nondeterministic input driven PDA, then there exists a deterministic input driven PDA P' accepting the same languages.

Proof

The simulation by P' is real time and just parallel to the computation of P. P' simultaneously keeps all possible nondeterministic computation paths step by step storing all necessary information in the cells of its stack. P' is able to do so, because the sequence of pushes and pops i.e. the stack height graph depends only on the input word and is not affected by the nondeterminism.

Thus the stack movements of P and P' are exactly the same. But the contents of corresponding cells of P and P' are different: when P writes nondeterministicly one symbol into a cell, P' writes all symbols in this cell that could possibly be printed by P. Yet reading a cell again P' doesn't know which of the symbols belongs to which computation path (possibly having died in the meantime). Therefore P' prints not only the new symbols but also keeps the old symbols and additionally prints the symbols which have to be printed in the cell above, i.e. P' prints triples of symbols (old, new, above), where 'old' and 'new' serve as pointer connecting correlated symbols in neighbouring cells. Looking at the old-symbol of a tuple and at the above-symbol of the tuples on the cell below,

P' is able to realize which tuples above are coming from which tuples below, i.e. which tuples are printed in the same computation path. Furthermore P' prints with each symbol the state in which P would read this symbol.

More precisely P' works as follows:

(pA qB --) in a stack cell means:

1) this is the top of the stack
2) the last time P reached this cell from below P printed A into this cell and reached state p (nondeterministicly)
3) q is the actual state and B the actual top symbol in the computation of P.

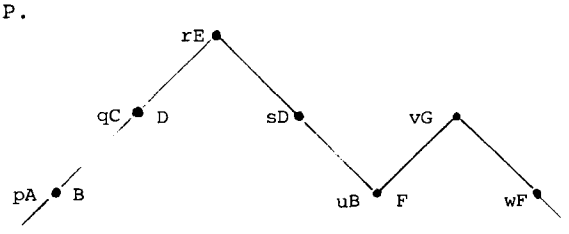
(pA - B qC) in a stack cell means:

1) like 2) above
2) the last time P printed on this cell (nondeterministicly) P printed B on and C above this cell and entered state q.

Let P' find T = (pA qB --) under the tuples on its top and x on its input tape. P' erases this tuple if no instruction of P exists beginning with (qBx). If there are instructions ($qBx\ r_iC_i$), P' replaces T by the tuples ($pA\ r_iC_i$ --) (i=1,...,n). If there are instructions ($qBx\ r_iC_iE_i$) P' replaces T by ($pA\text{-}C_i\ r_iE_i$) and prints ($r_iE_i\ r_iE_i$ --) above that cell. If there are instructions ($qBx\ r_i$) then P' first erases T and then replaces each tuple (sD-CpA) for some D,C,s under this cell by the tuples ($sD\ r_iC$ --).
Note that P' prints no tuple twice on a cell. Thus there are only finitely many tuples on each cell.

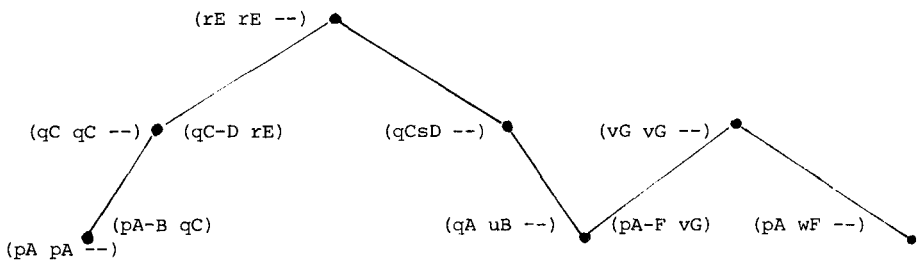
In the following example for reason of simplicity we consider just one computation path of P.

Computation of P:




(P reads A in state p, changes to state q and prints BC.
P reads E in state r, changes to state s and erases the top.)

Simulation of P':




As immediate consequences we obtain:

Corollary

All parenthesis languages are deterministic. □

Corollary

The class of idCFLs is contained in the class of real-time DCFLs. □

Corollary

The class of idCFLs is closed under complementation and reversal. □

The idPDAs seem to form the only natural class of automata below the Turing machines that accept nonregular languages for which the powers of the deterministic and nondeterministic versions are equal.

## 3. The simulation algorithm

### The principle of the simulation

Let $w = a_1 \ldots a_n$ be the input of our input driven PDA M and let $v = a_i \ldots a_{k-1}$ be a subword of w, such that in a computation of M applied to w, the stack height at time $t_i$ (i.e. reading the input symbol $a_i$) is the same as at time $t_k$ and never lower in between. Because the graph h(t) depends on the input only, each computation on w produces the same graph. So the following definition makes sense: The table $T_v$ of v is a relation out of $(Q \times \Gamma)^2$ with $(q,A,q',A') \in T_v$ iff the following holds: if at time $t_i$ M is in state q and the stack top symbol is A then at time $t_k$ M reaches state q' reading stack top A'.

To find the result of the computation of M on w, we compute $T_w$.

Let us consider the directed, acyclic computation graph G of M computing w. The nodes are the points (t,h(t)) of the "graph" of h. (t,h(t)) → (t',h(t'))) is a (directed) edge of G iff t' = t+1 or (h(t+1) > h(t) and t' is the next time after t with h(t) = h(t')).
That means that an edge lies on the curve h(t) (i.e. on the shape of the computation graph) or is cutting a mountain of the curve horizontally. In the second case a table corresponds to this edge. The idea is to determine the tables on the peaks of our push-pop mountain range, then to go down the left and right flanks and to compute the table in every step.

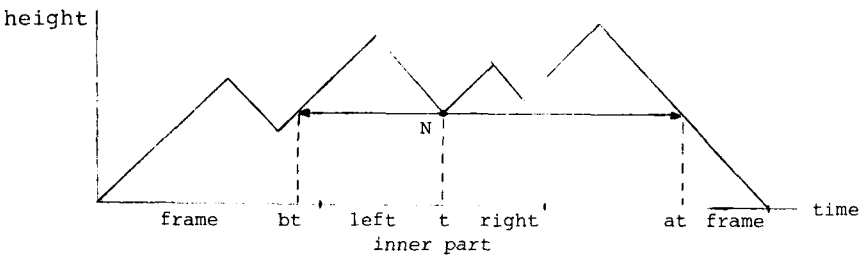
We compute the table corresponding to the actual horizontal edge from the PDA-relation $\delta$ and the last computed table (on the level above). In the bottom of a valley we combine the meeting tables into one table.

This is a special black and white pebbling strategy. This variant of the pebble game on graphs [CS 76] is played as follows: you may place a white pebble or remove a black pebble everywhere at any time and you may place a black pebble or remove a white pebble only if all predecessors are pebbled (black or white) (or no predecessor exists). On our computation graph G the game starts without any pebble. The goal is: the node (n,h(n)) = (n,o) has a black pebble and no other node is pebbled. To come to this goal you can choose a black pebbling strategy beginning with node (1,h(1)) ending with node (n,h(n)) or you may prefer a white pebbling strategy working in the opposite direction. The table idea is a mixed strategy with black and white pebbles. We are putting white pebbles on the peaks and on the left flanks of the mountains and black pebbles on the right flanks. Each table corresponds to a pair of a white and a black pebble lying on nodes connected by an horizontal edge. If a black and a white pebble meet at the bottom of a valley, then both of them are deleted.

Our aim is to save space (and time). Thus we must not remember too many tables at a time. As usual this is achieved by a divide-and-conquer strategy of logarithmic depth.

We cannot handle parts of the mountain range with different starting and ending height. Therefore we divide a mountain range (with ends of equal height) into three parts that are supposed to have an almost equal length: we look for the lowest node N with time t in the middle third and use this as cut point for the division into three parts: the left inner part containing all nodes between the last node before N and lower than N, and N itself, the right inner part with all nodes between N and the first node after N and lower than N, and the frame containing the remaining nodes.

Example:




Obviously the length of all these three subranges is at most 2/3 of the divided mountain range.

The frame is not a contiguous subrange, it contains a gap. In the following we formally consider only subranges with gap. They are denoted by $(l,r,g_1,g_2)$ = (left border, right border, left border of the gap, right border of the gap). If a subrange does not contain a gap, we state a gap of length o at the end of the middle third, i.e. $g_1 := g_2 := \lfloor l + \frac{2}{3}(r-l) \rfloor$. The length of a subrange with gap is $g_1 - l + r - g_2$.

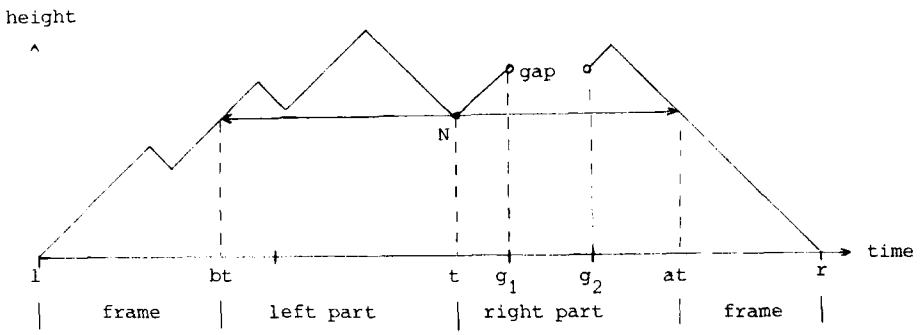
Our dividing strategy will guarantee:

(1) logarithmic depth
(2) all subranges contain just one gap.

A subrange with gap is divided in a similar way as in the simpler case without gap: consider the greater of the two parts $l \ldots g_1$ and $g_2 \ldots r$. Divide it into two parts of equal length and search for the lowest node N in the inner part
(i.e. in $\lfloor l + \frac{g_1 - l}{2} \rfloor \ldots g_1$ or in $g_2 \ldots \lfloor g_2 + \frac{r-g_2}{2} \rfloor$ respectively).

Use N as cut point in the same way as above.
Observe that in the case of a subrange without gap both instructions produce the same partition.

Example




Consider w.l.g. a subrange S with $g_1-l > r-g_2$.
S falls into

- one subrange with a gap $S_1 = (t,at,g_1,g_2)$,
- the frame (with a gap) $S_2 = (l,r,bt,at)$,
- one subrange without a gap $S_3 = (bt,t,g,g)$, where $g = \lfloor bt + \frac{2}{3}(t-bt) \rfloor$.

By the definition of t:

$$|S_1|,|S_2| \leq \frac{3}{4}|S|,$$

$$|g_1-t|,|t-g_2|,|bt-l|,|r-at| \leq \frac{1}{2}|S|.$$

Only $|S_3|$ may be larger than $\frac{3}{4}|S|$. In the next dividing step $S_1,S_2$ fall into subranges with length $\leq \frac{9}{16}|S|$ (the subrange without a gap is not larger than $\frac{1}{2}$ S), and $S_3$ falls into subranges of length $\leq \frac{2}{3}|S|$. Thus after at most $2 \cdot \log_{3/2}|S|$ divisions the length of the resulting subranges is at most 2.

In the algorithm the gap will represent a subrange with a known table. Thus we must consider the inner parts before the frame. To avoid complications, we consider the inner part with gap before the other inner part.

As an illustration of our strategy we give a sketch of a recursive function TAB that computes the table for a subrange (co means comment):

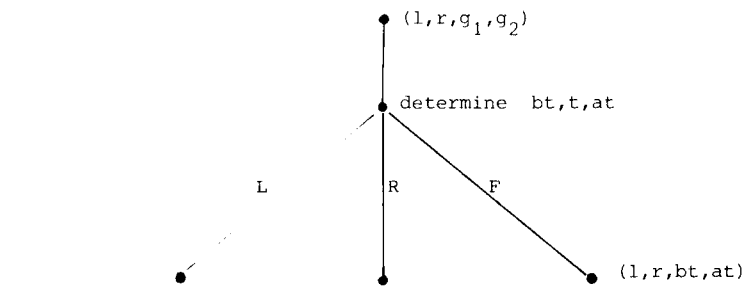
```
Function TAB (l,r,g1,g2);
begin if not l ≤ g1 ≤ g2 ≤ r then g1 := g2 := ⌊l + 2/3(r-l)⌋;
                                  G := identity table fi
      co G is global and contains the table for the gap;
      if (g1-l) + (r-g2) ≤ 2
      then determine TAB directly from PDA's table, the input
           and the table of the gap G
      else co the length of the subrange is greater than 2;
           let w.l.G. g1-l ≥ r-g2;
           determine t,bt,at as described above;
           co in the considered case is t ≤ g1 ≤ g2 ≤ at;
           TR:= TAB(t,at,g1,g2); co right part (with gap);
           TL:= TAB(bt,t,g1,g2); co left part (without gap);
           G := combination of TL and TR;
           co this is the table for the gap of the frame;
           T := TAB(l,r,bt,at);
           TAB := T
      fi
end
```

Since the depth of recursion is $O(\log n)$ and the local space of the procedure is also $O(\log n)$, the space complexity of this procedure is $O(\log^2 n)$. On a machine with random-access input the time complexity is $O(n \cdot \log n)$.

The log n - space algorithm

In the algorithm sketched above, the recursive function uses space $O(\log n)$ for storing the actual limits of the subrange. Our log-space algorithm will use only a constant amount of local space (for storing tables); the limits are computed when they are used. The only difficulty is the reconstruction of the old limits when the frame is considered.

Our dividing strategy may be viewed as traversing a ternary tree, where the edges are labelled L,R,F (for left part, right part, frame). If the frame is considered, we must recompute the limits (l,r) of the father of the actual node in the tree.




For this recomputation we use a stack over {L,R,F} containing the path from the root of the tree to the actual node. The limits can be recomputed in $O(\log n)$ space using the information stored in this stack.

In order to avoid too many recomputations we compute the table for a short subrange directly from the input symbols $x_1 \ldots x_{g_1-1}$, $x_{g_2} \ldots x_{r-1}$ and the table for the gap, if its length is smaller than $m = \log_c n$ where $c = 4|Q \times \Gamma|$. It is easy to see that a simulation of all $O((c/4)^m) = O(\sqrt{n})$ possible computation paths can be done in $O(\log n)$ space and $O(n)$ time.
This computation is done in the following subroutine, whose formal realization is left to the reader:

```
function EVAL(l,r,g1,g2): table;
co EVAL determines the table for  x1...xr-1  using the table for
   xg1...xg2-1  if  g1-l + r-g2 ≤ logcn;
```

Before we give the description of the algorithm (in a Pascal-like language) we define some more subroutines:

```
function BT(t : integer): integer; co determines  bt;
begin BT := max{i < t | h(i) < h(t)} + 1 end;

function AT (t : integer) : integer; co determines  at;
begin AT := min{i > t | h(i) < h(t)} - 1 end;

function LP(a,b : integer) : integer; co determines the lowest node
                                         in  (a,b);
begin local var i,min : integer;
      min := ∞;
      for i=a  to  b  do if  h(i) < min  then  LP := i; min := h(i)
                      fi od
end;

procedure RECOMPUTE;
co computes the actual values for the (global) variables  l,r
   using the (global) stack  S;
begin local var  ḡ1,ḡ2,p,t;
      p := 1; co  set stack pointer to bottom(S);
      (l,r,ḡ1,ḡ2) := (1,n+1,o,o); co initial values;
      repeat if not l ≤ ḡ1 ≤ ḡ2 ≤ r  then  ḡ1 := ḡ2 := ⌊l+2/3(r-l)⌋ fi;
             if  ḡ1-l ≥ r-ḡ2
             then  t := LP(⌊l + (ḡ1-l)/2⌋)
             else  t := LP(⌊g2 + (r-ḡ2)/2⌋)
             fi;
```

```
            case S(p) of
            'L': (l,r) := (BT(t),t);
            'R': (l,r) := (t,AT(t));
            'F': (g̅1,g̅2) := (BT(t), AT(t));
            esac;
            p := p+1;
      until p > length(S);
end;
```

The crucial procedure is the parameterless recursive function TAB. We describe it together with the main program:

```
program IDCFL;
type table : set of (Q × Γ)^2
global var m,l,r,g1,g2 : integer;
           TG : table; co table of the gap;
           S : stack over  {L,R,F};
function BT ...
function AT ...
function LP ...
procedure RECOMPUTE ...
function EVAL ...
```

```
function TAB() : table; co returnes the table for (l,r);
local var TL,TR : table; co for the left and right inner part;
begin if not l ≤ g1 ≤ g2 ≤ r
      then g1 := g2 := ⌊l + 2/3(r-l)⌋;
           TG := {(qX qX) | q ε Q, X ε Γ}
      fi;
      if (g1-l) + (r-g2) ≤ m
      then TAB := EVAL(l,r,g1,g2)
      else if (g1-l) ≥ (r-g1)
           then co the left part is larger and will contain the
                   dividing point, the gap is in the right part;
                l := LP(l + ⌊(g1-l)/2⌋,g1); r := AT(l);
                push R on S; TR := TAB(); pop S;
                r := l; l := BT(l);
                push L on S; TL := TAB(); pop S;
                r := AT(r);
           else co similar, but dividing point in the right part and
                   first L then R, the instructions are omitted;
           fi;
           TG := {(pX qY) | (∃rZ)(pX rZ) ε TL & (rZ qY) ε TR};
           (g1,g2) := (l,r); RECOMPUTE;
           push F on S; TAB := TAB(); pop S;
      fi;
end;
begin m := ⌊log n/ log(4|Q|·|Γ|)⌋; co if length of a subrange is
                                       smaller than m, then EVAL is
                                       used;
      (l,r,g1,g2) := (1,n+1,0,0); TG := ∅; S := ∅;
      if (q0,qacc) ε TAB() then accept else reject
end.
```

Theorem 4

The algorithm works on a Turing machine with tape $O(\log n)$ and time $O(n^2)$ using a two-way input tape, $O(n^2/\log n)$ using random access input.

Proof

The global space including the stack $S$ and the local space of the nonrecursive procedures is $O(\log n)$. TAB uses $O(1)$ local space and the depth of recursion is $O(\log n)$. Thus the $O(\log n)$ space bound is obvious.

For the time bound we must consider the cases of tape input and random-access input separately.

In the case of (two-way) input tape every call of AT,BT or LP takes $O(n)$ steps. EVAL can easily be implemented so that it takes $O(n)$ steps. RECOMPUTE takes $O(n \log n)$ steps, since $O(\log n)$ calls of AT,BT and LP have to be executed. The whole ternary tree representing the computation of TAB contains $O(n/\log n)$ nodes (every subtree contains at least one leaf that represents a subrange at least of length $\frac{\log n}{3}$). Thus $O(n/\log n)$ calls of TAB (and hence of RECOMPUTE) have to be performed. From this the $O(n^2)$ time-bound follows immediately.

In the case of random-access input BT,AT and LP take time proportional to the length of the actual subrange. Since this is exponentially decreasing in the depth of recursion, RECOMPUTE takes at most $O(n)$ steps. Thus the total time is $O(n^2/\log n)$.

□

References

[BCMV 83] B.v.Braunmühl, S.A. Cook, K. Mehlhorn, R. Verbeek:
The recognition of deterministic CFLs in small time and space,
Inf. Contr. 56, pp. 34-51.

[BV 8o] B.v. Braunmühl, R. Verbeek:
A recognition algorithm for DCFLs optimal in time and space,
Proc. 21th FOCS, pp. 411-42o.

[Co 79] S.A. Cook:
Deterministic CFLs are accepted simultaneously in polynomial time and log squared tape,
Proc. 11th ACM-STOC, pp. 338-345.

[CS 76] S.A. Cook, R. Sethi:
Storage requirements for deterministic polynomial time recognizable languages.
JCSS 13, 25-37.

[Ig 77] Y. Igarashi:
The tape complexity of some classes of Szilard languages,
SIAM 3. Comp. 6, pp. 46o-466.

[Ig 78] Y. Igarashi:
Tape bounds for some subclasses of deterministic context-free languages,
Inf. Contr. 37, 321-333.

[LSH 65] P.M. Lewis, R.E. Stearns, J. Hartmanis:
Memory bounds for the recognition of context-free and context-sensitive languages,
Proc. 6th IEEE-SWAT, pp. 191-212.

[Ly 77] N. Lynch:
Log space recognition and translation of parenthesis languages,
JACM 24, pp. 583-59o.

[LZ 77] R. Lipton, Y. Zalcstein:
Word problem solvable in log-space,
JACM 24, 522-526.

[Me 76] K. Mehlhorn:
Bracket languages are recognizable in logarithmic space,
Information processing letters 5, pp. 168-17o.

[Me 8o] K. Mehlhorn:
Pebbling mountain ranges and its application to DCFL-recognition,
Proc. 7th ICALP, pp. 422-432.

[PH 7o] M.S. Paterson, C.E. Hewitt:
Comparative schematology,
project MAC, Conf. on concurrent systems and parallel computation, pp. 1o9-127.

[RS 72] R.W. Ritchie, F.N. Springsteel:
Language recognition by marking automata,
Inf. Contr. 2o, 313-33o.

[Su 75] I.H. Sudborough:
On tape bounded complexity classes and multi-head finite automata,
JCSS 1o, 177-192.

[Ve 81] R. Verbeek:
Time-space tradeoffs for general recursion,
22nd IEEE-FOCS, pp. 228-234.

[Ve 83] R. Verbeek:
Complexity of pushdown computations,
Habilitationsschrift,
Institut für Informatik der Universität Bonn.