Universität Stuttgart
Fakultät Informatik

Institut für Informatik
Breitwiesenstraße 20-22
D-70565 Stuttgart

A Census Technique for Simple Computing Devices

Holger Petersen

Report Nr. 1997/07

June 27, 1997

Abstract

We develop a technique that uses pebbles in a very economical way for simulating a multi-counter automaton on a sequence of input strings and maintain a count of those strings accepted by the counter automaton. Based on this result we show the failure of a previously proposed method for constructing witness sets separating classes of languages accepted by pebble automata with an increasing number of pebbles. We also give a recognition algorithm for another family of languages suspected to separate the lower levels of the hierarchy.

1 Introduction

The investigation of restricted Turing machines that are allowed to mark a fixed number of positions of their input goes back at least to the work of Kreider and Ritchie [11]. Subsequently various modifications of this concept (here called pebble automaton) have been considered, e.g. [5, 8, 9, 10, 13, 14, 15]. It is well-known that deterministic (nondeterministic) pebble automata characterize the complexity classes DSPACE($\log n$) (NSPACE($\log n$)), see [18, Section 3.2] for a construction. Alternation gives the class P (deterministic polynomial time) [10].

A main concern of complexity theory is the question whether an increase of the available resources increases the computational power as well. Rephrased in terms of pebble automata we ask whether there are languages separating the classes of languages accepted by automata equipped with k and $k+1$ pebbles. A traditional method to establish such a hierarchy employed in most of the cited references consists of two steps:

- Use a diagonal argument in order to separate some levels of the hierarchy.
- Refine the separation obtained in the first step with the help of transformational methods.

While the first step is comparatively easy for deterministic automata, the second step tends to be rather technical due to the limited power of the computational models under consideration. It is another drawback of transformational methods that only automata having identical characteristics can be separated with respect to increasing resources.

A completely different way to obtain separation results, here called census approach, has been suggested by Hsia and Yeh [8]. They propose to take a language L of high complexity from level k of the hierarchy and form sequences of words separated by a new symbol, such that the number of words from L equals the number of words from its complement $\overline{L}$. Then the language consisting of all such sequences should separate levels k and $k+1$.

Such a construction would be of considerable interest since it would lead to natural examples of witness languages paralleling the situation for one-way automata [19] (see also [12, Theorem 6.11]), where the hierarchy results rely on the combinatorial difficulty of languages.

We will establish in this article that the construction from [8] does not work even for the first non-trivial level of the hierarchy, $k = 2$. In addition we describe generalizations of their idea that still do not give the desired separation. We also discuss the gap in the argument given for Theorem 1 of [8], the basis for a hierarchy of languages recognizable by pebble automata, an error of intrinsic importance that has survived the reviews [3, 6, 7].

In an earlier work [15] the author stated without proof the weaker result that the argument of [8] failed for $k \geq 3$, provided that bounded counting automata and pebble automata of corresponding levels $k-1$ could be separated, a yet unproven hypothesis for $k \geq 3$. The present work improves this result to a level where the inequivalence of the latter models is known, thus showing that the construction of [8] indeed fails.

In a similar vein we look at possible candidates for separating the power of two versus three pebbles that have been proposed by Chang e.a. [2]. Note however that they do not claim to have a proof. We show that these candidate languages can be accepted with the help of two pebbles.

Our results can be taken as evidence for the difficulty of finding concrete languages accepted by pebble automata, but not with as few as two pebbles or pointers on the input string. In this sense the result of [4] might be optimal, that a language related to string matching cannot be accepted by an automaton with two heads one of which can see the end-markers but cannot distinguish input symbols.

2 Preliminary Remarks

In this section we will summarize some definitions and known facts relevant for our investigation.

A *pebble automaton* with k pebbles is a deterministic single-head two-way finite automaton with end-markers that has k pebbles (markers) available which it may place on its tape squares. The pebbles can later be recognized, picked up, and redistributed. A *bounded counting* automaton with k counters operates similarly but has k counters instead of pebbles. These counters can be increased, decreased, and tested for being zero. Throughout the computation of the automaton the counters have to be bounded by the input length. Note that the automaton cannot actively check whether the bound has been reached. Formal definitions of these devices can be found in [16]. A third computational model we will use is the deterministic finite multi-head automaton with k sensing heads, i.e., heads that can "feel" the presence of other heads on the same square. It should be clear that these devices can be simulated by k pebble automata. All automata start their computation on the first input symbol (if the input string is not empty) and accept by going to final states.

We note that the distinction made in [8] between halting and non-halting pebble automata is now obsolete due to Sipser's technique for halting deterministic space bounded computations [17].

The following result is due to Blum and Hewitt [1] and, independently, A.R.Meyer:

Proposition 1 *Every language accepted by a one-pebble automaton is regular.*

It is easy to design automata equipped with a single counter that accept languages like $E = \{a^n b^n \mid n \geq 0\}$ or $P = \{a^{2^n} \mid n \geq 0\}$.

Proposition 2 *There are non-regular sets accepted by bounded counting automata with one counter.*

We remark that a halting two-way one counter automaton (being deterministic) always respects a linear bound on the counter and therefore can be transformed into a bounded counting automaton by suitably compressing the counter contents.

From [16] we know:

Proposition 3 *Every bounded counting automaton with k counters can be simulated by a pebble automaton with $k+1$ pebbles.*

Slightly improving the simulation in [16] we have [15]:

Proposition 4 *Every pebble automaton with k pebbles can be simulated by a bounded counting automaton with k counters.*

We summarize the situation, where "level k" denotes the classes of languages accepted by automata with k pebbles or counters respectively.

- At level 0 the classes coincide with the regular sets.
- At level 1 bounded counting automata are stronger than pebble automata.
- At level $k \geq 2$ bounded counting automata are at least as powerful as pebble automata.

3 The Candidate Languages

We recall that in [8] languages L' are constructed which are accepted by automata with $k+1$ pebbles, but are claimed to be acceptable by no such machine with at most k pebbles. Let $L \subseteq \Sigma^*$ be a language that can be accepted by an automaton with k pebbles, but by no automaton with $k-1$ pebbles. We have indicated above that such languages exist for $k=2$. Then a language L' is formed from L by adding a new symbol $\#$ to the alphabet of L and letting L' consist of all $w_1\#w_2\#\cdots\#w_m$ such that the number of words $w_i \in \Sigma^*$ in L is equal to the number of words not in L. The idea of the proof is that an automaton accepting L' has to use k pebbles to decide each w_i and needs an additional pebble to store a count in order to check the equality. There are however other ways to do the counting.

Theorem 1 *If L is a language accepted by a bounded counting automaton with $k-1$ counters for some $k \geq 2$, then the L' constructed from L can be accepted by a pebble automaton with k pebbles.*

Note that, by the results mentioned in the preceding section, such an L, which in addition cannot be accepted by a pebble automaton with $k-1$ pebbles, would satisfy the conditions of the construction outlined above, since it can be accepted by an automaton with k pebbles.

Before giving the proof of Theorem 1 we will look at its consequences.

Corollary 1 *The languages E' and P' (where E and P are the non-regular languages defined above) can be accepted by pebble automata with two pebbles.*

Proof of the Corollary: Languages E and P can be accepted by bounded counting automata with one counter. Thus they satisfy the conditions required by Theorem 1 for $k=2$. $\square$

It is this corollary that contradicts the main result of [8], their Theorem 1 (note however that Corollary 1 of [8], the infinite hierarchy, is valid, see e.g. [5, 9, 14, 15]).

Proof of Theorem 1: We will first give an informal outline of the construction and then present a more detailed algorithm.

We describe an automaton M with k sensing heads accepting L' (remember that such an automaton can be simulated with the help of k pebbles).

Every word w_i over the alphabet of L appearing in the input together with the separator symbol immediately to the right ($\#$ for $1 \leq i < m$, an end-marker for $i=m$) will be called *block* i. The length of w_i is denoted by b_i. A head resting on block i determines a number d_i, the distance to the separator symbol within the same block. If $w_i \in \{a\}^*$ for $i < m$ we get the following situation (the head position is underlined):

$$\overbrace{aaa\cdots aa\,\underbrace{\underline{a}aa\cdots aaa}_{d_i}}^{b_i}\#$$

For a given input $w_1\#w_2\#\cdots\#w_m$ we renumber the w_i (without actually rearranging them) according to their length (larger numbers are assigned to longer words). If there are words of the same length we count them left to right. From now on words will be referred to by these new numbers.

First M uses two heads to determine the rightmost block of maximum length (procedure `maxblock`). This is the block of w_m. Then it leaves head 1 on a distinguished block (initially block m) while the other heads are moved onto $w_m, w_{m-1}, \ldots, w_1$ in turn, testing them for membership in L.

We explain how M, if it has head 1 pointing to some word w_j with $j \geq i$, can move another head from w_i to w_{i-1}. In order to achieve this M stores the length of w_i as the distance d_j with the help of head 1. Then it moves head 2 left onto the word of the next block and checks whether its length equals d_j by moving the heads in parallel. In case of equality it has found w_{i-1}. Otherwise it moves head 1 back to its position before the comparison, moves the second head onto the next word to the left and repeats this process. If no word of the same length as w_i can be found, the second head is moved to the right end of the input, head 1 is moved one symbol to the right (thus decrementing d_j), the count that is encoded by the block that head 1 resides in is updated (see below), and the comparison is continued. Eventually either w_i is found to be the first word or w_{i-1} is located.

If word w_{i-1} has been found membership in L is decided by a function `member` based on the automaton accepting L. This function has to satisfy the following requirements:

- It moves all heads except head 1 to the block occupied by head 2. Since the heads are sensing they can find this block.
- It initializes the positions of heads 2 through k and initializes the count stored as the distance of head 1.
- It uses the separator symbol as an (additional) end-marker. This is no difficulty, because the direction of the last move of a head determines which type of end-marker may be encountered.
- It never leaves the input area.
- After deciding w_{i-1} it restores the distance stored previously by head 1. This is easily achieved (procedure call `letdibl(1,2)`) since it equals b_{i-1}.

Note that `member` can use $k-1$ "real" heads on w_{i-1} and head 1 as a counter, which is represented by the position of head 1 in its current block. Thus our simulation is more general than required for the assertion of the theorem if $k \geq 3$. Due to the construction head 1 never has to leave its block, since this block is at least as long as w_{i-1}.

Depending on the outcome of the test M increments the count kept by the position of head 1 or leaves it unchanged. The count is encoded by the block that head 1 scans.

It remains to be outlined how the block that head 1 scans encodes the count of words belonging to L. Just before the call to `member` this count will (with one exception) be one more than the number of words to the right of the block scanned by head 1 that are at least as long as indicated by the distance of head 1 to its separator.

The exception is the initial situation indicated by a boolean variable `first`, which records the first occurrence of a word belonging to L.

We now give a detailed algorithm for the recognition of L'. Since no access to the symbols of words w_i is required at this level of the description M's interface to the input consists of the following boolean functions:

`endl(i: head)` Reports whether head `i` rests on the left end-marker.

`endr(i: head)` Reports whether head `i` rests on the right end-marker.

`onsep(i: head)` Reports whether head `i` rests on an end-marker or on a separator #.

`sense(i,j: head)` Reports whether heads `i` and `j` rest on the same tape square.

Procedures `left(i: head)` and `right(i: head)` move a head one square to the left or right, `home(i: head)` moves a head onto the left end-marker.

The procedures `alignl(i: head)` and `alignr(i: head)` move a head onto the leftmost and rightmost symbol of a block, `blockl(i: head)` and `blockr(i: head)` move it onto the left and right neighboring block (right aligned).

Head `i` is moved to the position of head `j` with the help of the following procedure:

```
procedure visit(i,j: head);
begin
        while not sense(i,j) and not endr(i) do right(i);
        while not sense(i,j) do left(i)
end;
```

The next procedure compares the lengths of the blocks occupied by heads `i` and `j` for the relation greater or equal. It destroys the positions of these heads within their blocks.

```
function geblbl(i,j: head): boolean;
begin
        alignl(i);
        alignl(j);
        while not (onsep(i) or onsep(j)) do      (* compare lengths *)
          begin
               right(i);
               right(j)
          end;
        geblbl := onsep(j)
end;
```

With the help of the next procedure the length of a block is duplicated as the distance of a head to the right block-separator of another block. Old distances are lost.

```
procedure letdibl(i,j: head);
begin
        alignr(i);
        alignl(j);
        while not onsep(j) do
        begin
          left(i);
          right(j)
        end
end;
```

This is one of the crucial procedures. It compares the distance stored by head `i` and the length of the block occupied by head `j` for equality. The distance is preserved, while the position of head `j` within its block is destroyed:

```
function eqdibl(i,j: head): boolean;
begin
        alignl(j);
        while not(onsep(i) or onsep(j)) do       (* compare lengths *)
        begin
          right(i);
          right(j)
        end;
        eqdibl := onsep(i) and onsep(j);         (* both on end-markers *)
        repeat                                   (* + blocklen. - dist. *)
          left(i);
          left(j)
        until onsep(j);
        right(i);
        right(j)
end;
```

The next procedure locates the rightmost block of maximum length.

```
procedure maxblock;
begin
        home(2);
        home(1);
        while not endr(1) do                     (* for every block *)
        begin
          blockr(1);
          if geblbl(1,2) then visit(2,1)
        end
end;
```

The following procedures increment resp. decrement the number stored implicitly by the block that head 1 occupies.

```
procedure incr;
begin
        if not first then first := true          (* special case *)
        else
        begin
          repeat                                 (* locate long block *)
            blockl(1)
          until geblbl(1,2);
          letdibl(1,2)                           (* restore distance *)
        end
end;

procedure decr;
begin
        repeat                                   (* never decr. from 0 *)
          blockr(1)
        until geblbl(1,2);                        (* locate long block *)
        letdibl(1,2)                              (* restore distance *)
end;
```

This procedure updates the count kept by the block that head 1 rests upon.

```
procedure restcnt;
begin
        visit(2,1);
        alignr(2);
        while not endr(2) do                      (* for every block *)
        begin
          blockr(2);
          if eqdibl(1,2) then decr;               (* critical length *)
          alignr(2)
        end
end;
```

The following predicate incorporates all the operations implemented above and tests membership in L', making use of the predicate `member` for L (the word to be tested is pointed to by head 2).

```
function check: boolean;
begin
        first := false;
        maxblock;
        visit(1,2);
        alignl(1);
        left(1);
        repeat                                    (* for all distances *)
          right(1);
          restcnt;
          home(2);
          repeat
            blockr(2);
            if eqdibl(1,2) then                   (* blocks of cur. len. *)
              if member then incr;
            alignr(2)
          until endr(2)
        until onsep(1);
        home(2);
        if first then blockl(1);
        alignr(1);
        alignr(2);
        while not (endr(1) or endr(2)) do         (* check count = m/2 *)
        begin
          blockr(2);
          if not endr(2) then
```

```
            begin
              blockr(2);
              blockr(1)
            end
          end;
          check := endr(1) and endr(2)             (* equality *)
end;
```

4 Generalizations

One way to interpret the result of the Section 3 is that a predicate concerning the number of words from L that occur in the list is evaluated. Based on a language $L \subseteq \Sigma^*$ and a function f from non-negative integers to the rationals we can define

$$L_f = \{w_1\#w_2\#\cdots\#w_m \mid f(m) = |\{1 \leq j \leq m \mid w_j \in L\}|, \forall 1 \leq i \leq m : w_i \in \Sigma^*\}$$

and investigate, for which f and k it is true, that a k pebble automaton can accept L_f if L can be accepted by a bounded counting automaton with $k-1$ counters.

We know that for $k \geq 2$ and $f(m) = m/2$ this statement holds (Theorem 1), since $L' = L_f$ (for odd m the condition cannot be satisfied). This can easily been extended to other ratios than 1/2.

By generalizing the technique developed in Section 3 we can, e.g., show, that for $k \geq 2$ the non-semilinear correspondence $f(m) = \max(\{2^p \mid 2^p \leq m, p \geq 0\} \cup \{0\})$ and for $k \geq 3$ the function $s(m) = \lfloor\sqrt{m}\rfloor$ are possible. The idea is to simulate an automaton that receives the count as one of its head positions and compares it with the value of the function. We will briefly sketch this construction for $s(m)$. It is clear that the computation of an automaton on input $\#^m$ can be simulated by ignoring all input symbols except # and virtually inserting a # before the right end-marker. Let the count encoded as the distance of head 1 to the right end-marker be c, the other heads are on the end-marker encoding 0. The relation

$$\sum_{i=1}^{c}(2i-1) = c^2$$

admits the computation of c^2 as the position of head 2 by letting head 3 oscillate between the right end-marker and the first head's current position at distance i, adding $2i-1$ to the position of head 2. After one pass of head 3 the distance i is decremented by one. If it reaches 0 the process terminates. In this way the automaton can verify $c^2 \leq m$ by rejecting if the number encoded by head 2 exceeds m. In order to verify that $(c+1)^2 > m$ the count c is recomputed, incremented, and $(c+1)^2$ partially computed as described above, this time accepting if and only if the computed value exceeds m.

5 Other Candidate Languages

In this section we will discuss a different family of languages that has been proposed as a source for candidates separating two and three pebble automata. It is based on the language of marked palindromes (x^R is the reversal of string x).

$$L = \{x\#x^R \mid x \in \{0,1\}^*\}.$$

Let $L^k = LL \cdots L$, where L is taken k times. While L and L^2 can clearly be accepted by two pebble automata, Chang e. a. [2] point out that the recognition of L^k seems to require three pebbles for $k \geq 3$ (they do not claim to have a proof for this statement).

We will show how to accept L^k with the help of two pebbles for arbitrary k, in fact we can do it with a deterministic two-way one counter automaton. It is clear that such an automaton can be simulated by a two pebble automaton that uses one of its pebbles to encode the count. The count of a halting one counter automaton is linearly bounded by the input length and thus can be transformed into a bounded counting automaton.

Lemma 1 *Fix an input segment $u\#v$, $u, v \in \{0,1\}^*$ and $|u| = |v|$. Let the counter of an automaton A initially contain $|u|$ and its head be placed on $\#$. Then the counter automaton can determine whether $u = v^R$.*

Proof. While the counter is not zero A moves its heads left and decrements the counter. Then it reads the scanned symbol a. It remembers a, moves the head right restoring the count by incrementing for every symbol until it reaches the central $\#$. Then it performs a similar excursion to the right reading a symbol b, and returns to $\#$ again restoring the count. If $a \neq b$ the loop terminates and the automaton has established $u \neq v^R$, otherwise the count is decremented until the counter is zero in this step and $u = v^R$. □

We will now consider the entire input and write it as

$$x_1\#\hat{x}_1 x_2\#\hat{x}_2 \cdots x_k\#\hat{x}_k.$$

The ultimate goal for membership in L^k is to establish that there is factorization of this form with $x_i = \hat{x}_i^R$ for $1 \leq i \leq k$. A necessary condition is that $|x_i| = |\hat{x}_i|$ for $1 \leq i \leq k$. Note that these lengths are uniquely determined if such a factorization exists. The next lemma shows that the lengths can be computed as the contents of the counter.

Lemma 2 *A one counter automaton A is able to compute $|x_i|$ for inputs admitting the factorization above with $|x_j| = |\hat{x}_j|$ for $1 \leq j \leq k$ on its counter for $1 \leq i \leq k$.*

Proof. We will describe a procedure len(i) for computing $|x_i|$ on the counter assuming that a factorization of the described form exists. If the input does not respect these conditions the procedure may fail. This property will later be useful for checking the validity of the input string. We omit the head number 1 in the following pseudo-code, increment and decrement modify the counter, zero is the test. Note that i is bounded by k, therefore the loop can be implemented in the finite control of A.

```
procedure len(i);
begin
        while not zero do decrement;
        while not endl do left;
        for j := 1 to i do
        begin
          right;
          while not (onsep or zero) do
          begin
            decrement;
```

```
              right
            end;
            if not zero then abort;
            while not onsep do
            begin
              increment;
              right;
            end
          end
end;
```

□

Now we can combine our procedures.

Theorem 2 *For every $k \geq 1$ there is a one counter automaton accepting L^k.*

Proof. First the counter automaton A accepting L^k checks that there are exactly k symbols #. Now A executes the procedure call len(k) according to Lemma 2. Note that, even if len(k) does not fail, the input may not possess a valid factorization due to the length of $\hat{x}_k$. Therefore A compares its counter and $|\hat{x}_k|$. If these numbers are equal it recomputes each len(i) in turn and checks that $x_i = \hat{x}_i^R$ for $1 \leq i \leq k$ which is possible by Lemma 1. □

Corollary 2 *For every $k \geq 1$ there is a two pebble automaton accepting L^k.*

6 Concluding Remarks

We have been able to give recognition procedures for languages that have been proposed for separating finite automata with an increasing number of pebbles resp. two-way heads.

From the method we used in Section 3 we can deduce quite precisely the gap in the argument given in the proof of Theorem 1 of [8]. It is the assumption that the words in a given list can be processed according to the order of their appearance (p. 76).

It remains open whether there are separating languages for the models of computation considered here that are not based on diagonal arguments.

References

[1] M. Blum and C. Hewitt. Automata on a 2-dimensional tape. In *Proceedings of the 8th Annual Symposium on Switching and Automata Theory, Austin, 1967*, pages 155–160, 1967.

[2] J. H. Chang, O. H. Ibarra, M. A. Palis, and B. Ravikumar. On pebble automata. *Theoretical Computer Science*, 44:111–121, 1986.

[3] V. Claus. Review 29,454. *Computing Reviews*, 17(1):34, 1976.

[4] P. Ďuriš and Z. Galil. Fooling a two way automaton or one pushdown store is better than one counter for two way machines. *Theoretical Computer Science*, 21:39–53, 1982.

[5] J. Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1:336–344, 1972.

[6] J. Hartmanis. Review 7203. *Mathematical Reviews*, 52:1016, 1976.

[7] J. Hořejš. Review 94029. *Zentralblatt für Mathematik und ihre Grenzgebiete*, 341:546, 1977.

[8] P. Hsia and R. T. Yeh. Marker automata. *Information Sciences*, 8:71–88, 1975.

[9] O. H. Ibarra. On two-way multihead automata. *Journal of Computer and System Sciences*, 7:28–36, 1973.

[10] K. N. King. Alternating multihead finite automata. *Theoretical Computer Science*, 61:149–174, 1988.

[11] D. L. Kreider and R. W. Ritchie. A basis theorem for a class of two-way automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 12:243–255, 1966.

[12] M. Li and P. M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer, Berlin-Heidelberg-New York, 1993.

[13] B. Monien. Transformational methods and their application to complexity problems. *Acta Informatica*, 6:95–108, 1976. Corrigenda: ibid. 8:383–384, 1977.

[14] B. Monien. Two-way multihead automata over a one-letter alphabet. *R.A.I.R.O. — Informatique Théorique et Applications*, 14:67–82, 1980.

[15] H. Petersen. Automata with sensing heads. In *Proceedings of the Third Israel Symposium on the Theory of Computing and Systems, Tel Aviv, 1995*, pages 150–157. IEEE Computer Society Press, 1995.

[16] R. W. Ritchie and F. N. Springsteel. Language recognition by marking automata. *Information and Control*, 20:313–330, 1972.

[17] M. Sipser. Halting space-bounded computations. *Theoretical Computer Science*, 10:335–338, 1980.

[18] A. Szepietowski. *Turing Machines with Sublogarithmic Space.* Number 843 in Lecture Notes in Computer Science. Springer, Berlin-Heidelberg-New York, 1994.

[19] A. C. Yao and R. L. Rivest. $k+1$ heads are better than k. *Journal of the Association for Computing Machinery*, 25:337–340, 1978.