Linear Bounded Automata

The last machine model of computation which we shall examine is the *linear bounded automaton* or *lba*. These were originally developed as models for actual computers rather than models for the computational process. They have become important in the theory of computation even though they have not emerged in applications to the extent which pushdown automata enjoy.

Here is the motivation for the design of this class of machines. Computers are finite devices. They do not have unending amounts of storage like Turing machines. Thus any actual computation done on a computer is not as extensive as that which could be completed on a Turing machine. So, to mimic (or maybe model) computers, we must restrict the storage capacity of Turing machines. This should not be as severely as we did for finite automata though. Here is the definition.

> **Definition.** *A **linear bounded automaton** (lba) is a multi-track Turing machine which has only one tape, and this tape is exactly the same length as the input.*

That seems quite reasonable. We allow the computing device to use just the storage it was given at the beginning of its computation. As a safety feature, we shall employ endmarkers (* on the left and # on the right) on our lba tapes and never allow the machine to go past them. This will ensure that the storage bounds are maintained and help keep our machines from leaving their tapes.

At this point, the question of accepting sets arises. Let's have linear bounded automata accept just like Turing machines. Thus for lba halting means accepting.

For these new machines computation is restricted to an area bounded by a constant (the number of tracks) times the length of the input. This is very much like a programming environment where the sizes of values for variables is bounded.

Now that we know what these devices are, let's look at one. A set which cannot be accepted by pushdown machines (this is shown in the material on formal languages) is the set of strings whose length is a perfect square. In symbols this is:

$$\{a^n \mid n \text{ is a perfect square } \}.$$

Here is the strategy. We shall use a four track machine with the input written on the first track. The second and third tracks are used for scratch work while the fourth track is holds strings of square length which will be matched against the input string.

To do this we need to generate some strings of square length. The second and third tracks are used for this. On the second track we will build strings of length k = 1, 2, 3, and so forth. After each string is built, we construct (on the fourth track) a string whose length is the square of the length of the string on the second track by copying the second track to the fourth exactly that many times. The third track is used to count down from k. Here is a little chart which explains the use of the tracks.

track	*content*
1	a^n (input)
2	a^k
3	a^{k-m}
4	a^{mk}

Then we check to see if this is the same length as the input. The third track is used for bookkeeping. The algorithm is provided as figure 1.

```
repeat
  clear the 3rd and 4th tracks
  add another a to the 2nd track
  copy the 2nd track to the 3rd track
  while there are a's written on the 3rd track
     delete an a from the 3rd track
     add the 2nd track's a's to those on 4th track
until overflow takes place or 4th track = input
if there was no overflow then accept
```

Figure 1 - Recognition of Perfect Square Length Inputs

Now we've seen something of what can be done by linear bounded automata. We need to investigate some of the decision problems concerning them. The first problem is the halting problem.

Theorem 1. *The halting problem is solvable for linear bounded automata.*

Proof. Our argument here will be based upon the number of possible configurations for an lba. Let's assume that we have an lba with one track (this is allowed because can use additional tape symbols to simulate tracks as we did with Turing machines), k instructions, an alphabet of s

tape symbols, and an input tape which is n characters in length. An lba configuration is the same as a Turing machine configuration and consists of:

a) an instruction,
b) the tape head's position, and
c) the content of the tape.

That is all. We now ask: how many different configurations can there be? It is not too difficult to figure out. With s symbols and a tape which is n squares long, we can have only s^n different tapes. The tape head can be on any of the n squares and we can be executing any of the k instructions. Thus there are only

$$k*n*s^n$$

possible different configurations for the lba.

Let us return to a technique we used to prove the pumping lemma for finite automata. We observe that if the lba enters the same configuration twice then it will do this again and again and again. It is stuck in a loop.

The theorem follows from this. We only need to simulate and observe the lba for $k*n*s^n$ steps. If it has not halted by then it must be in a loop and will never halt.

Corollary. *The membership problems for sets accepted by linear bounded automata are solvable.*

Corollary. *The sets accepted by linear bounded automata are all recursive.*

Let's pursue this notion about step counting a bit more. We know that an lba will run for no more than $k\&n\&s^n$ steps because that is the upper bound on the number of configurations possible for a machine with an input of length n. But, let us ask: *exactly* how many configurations are *actually* reached by the machine? If we knew (and we shall soon) we would have a sharper bound on when looping takes place. Thus to detect looping, we could count steps with an lba by using an extra track as a step counter.

Now let's make the problem a little more difficult. Suppose we had a nondeterministic linear bounded automaton (nlba). We know what this is; merely a machine which has more than one possible move at each step. And, if it can achieve a halting configuration, it accepts. So we now ask: how many configurations can an nlba reach for some input? We still have only $k*n*s^n$

possible configurations, so if we could detect them we could count them using n tape squares. The big problem is how to detect them. Here is a rather nifty result which demonstrates nondeterminism in all of its glory. We start with a series of lemmata.

Lemma. *For any nondeterministic linear bounded automaton there is another which can locate and examine m configurations reachable (by the first lba) from some input if there are at least m reachable configurations.*

Proof. We have an nlba and an integer m. In addition we know that there are at least m configurations reachable from a certain input. Our task is to find them.

If the nlba has k instructions, one track, s symbols, and the input is length n, then we know that there are at most k&n&s^n possible configurations (C_i) reachable from the starting configuration (which we shall call C_0). We can enumerate them and check whether the nlba can get to them. Consider:

```
x = 0
for i = 1 to k*n*s^n
    generate C_i
    guess a path from C_0 to C_i
    verify that it is a proper path
    if C_i is reachable then x = x + 1
verify that x ≥ m (otherwise reject)
```

This is a perfect example of the *guess and verify* technique used in nondeterministic operation. All we did was exploit our definition of nondeterminism. We looked at all possible configurations and counted those which were reachable from the starting configuration.

Note also that every step above can be carried out using n tape squares and several tracks. Our major problem here is to count to k*n*s^n. We need to first note that for all except a few values of n, this is smaller than $(s+1)^n$ and we can count to this in base s+1using exactly n tape squares.

Since we verify that we have indeed found at least m configurations our algorithm does indeed examine the appropriate number of reachable configurations if they exist.

Lemma. *For any nondeterministic linear bounded automaton there is another which can compute the number of configurations reachable from an input.*

Proof. As before we begin with an arbitrary machine which has k instructions, one track, s symbols, and an input of length n. We shall iteratively count the number of configurations (n_i) reachable from the initial configuration. Consider:

```
n0 = 1
i = 0
repeat
  i = i + 1
  ni = 0
  m := 0
  for j = 1 to k*n*s^n
    generate Cj
    guess whether Cj can be reached in i steps or less
    if path from C0 to Cj is verifiable then
       ni = ni + 1
       if reached in less than i steps then m = m + 1
  verify that m = ni-1 (otherwise reject)
until ni = ni-1
```

The guessing step is just the algorithm of our last lemma. We do it by finding all of the configurations reachable in less than i steps and seeing if any of them is C_j or if one more step will produce C_j. Since we know n_{i-1}, we can verify that we have looked at all of them.

The remainder is just counting. We do not of course have to save all of the n_i, just the current one and the last one. All of this can be done on n squares of tape (and several tracks). Noting that we are done when no more reachable configurations can be found finishes the proof.

Theorem 2. *The class of sets accepted by nondeterministic linear bounded automata is closed under complement.*

Proof. Most of our work has been done. To build a machine which accepts the complement of the set accepted by some nlba involves putting the previous two together. First find out exactly how many configurations are reachable. Then examine all of them and if any halting configurations are encountered, reject. Otherwise accept.

Our final topic is decision problems. Unfortunately we have seen the only important solvable decision problem concerning linear bounded automata. (At least there was one!) The remaining decision problems we have examined for other classes of machines are unsolvable. Most of the proofs of this depend upon the next lemma.

> **Lemma.** *For every Turing machine there is a linear bounded automaton which accepts the set of strings which are valid halting computations for the Turing machine.*

The proof of this important lemma will remain an exercise. It should not be too hard to see just how an lba could check a string to see if it is a computation though. After all, we did a rather careful analysis of how pushdown machines recognize invalid computations.

> **Theorem 3.** *The emptiness problem is unsolvable for linear bounded automata.*
>
> **Proof.** Note that if a Turing machine accepts no inputs then it does not have any valid halting computations. Thus the linear bounded automaton which accepts the Turing machine's valid halting computations accepts nothing. This means that if we could solve the emptiness problem for linear bounded automata then we could solve it for Turing machines.

In the treatment of formal languages we shall prove that the class of sets accepted by linear bounded automata properly contains the class of sets accepted by pushdown machines. This places this class in the hierarchy

$$\text{fa} \subset \text{pda} \subset \text{lba} \subset \text{TM}$$

of classes of sets computable by the various machine models we have been examining.

(By the way, we could intuitively indicate why lba's are more powerful than pushdown machines. Two observations are necessary. First, a tape which can be read and written upon is as powerful a tool as a stack. Then, note that a pushdown machine can only place a bounded number of symbols on its stack during each step of its computation. Thus its stack cannot grow longer than a constant times the length of its input.)

The other relationship we need is not available. Nobody knows if nondeterministic linear bounded automata are more powerful than ordinary ones.