**18.405/6.841 Advanced Complexity** **Spring Term, 1996**

# Lecture 2 : February 8

*Lecturer: Michael Sipser* *Scribe: Ioana Popescu*

In the first part of this lecture we prove that the language CVAL is P-complete with respect to log-space reductions. We then move on to the study of *Alternating Turing Machines*, a natural generalization of the standard Turing Machines.

## 1 CVAL is P-complete

Recall that $CVAL = \{\langle C, x\rangle |$ circuit $C$ evaluates to '1' on input $x$, i.e. $C(x) = 1\}$. We have to show that :

1. $CVAL \in P$
2. $\forall A \in P$, $A \leq_L CVAL$ .

**Proof:**

The first part is clear, since given a circuit and an input we can easily simulate the computation of the circuit on that input in polynomial time.

For the second part, let $A \in P$ and let M be a TM that decides $A$ in time $n^k$. We will exhibit a log-space reduction $f$ from $A$ to CVAL, $w \xrightarrow{f} \langle C, x\rangle$ such that $w \in A$ iff $C(x) = 1$.

Consider the computation tableau of M on $w$. This is a $n^k \times n^k$ table where each row is a configuration: the first row is a start configuration, row $i$ is the configuration at the $i$th step. This tableau completely describes the computation of M on $w$, since M is guaranteed to halt within $n^k$ steps and therefore uses at most $n^k$ tape cells. For simplicity we will also assume that M has a single accepting configuration, in state $q_{accept}$ and with the head on the leftmost cell of the tape.

We will encode the contents of the tableau by a circuit that outputs '1' iff the last configuration in the tableau is an accepting configuration. For every cell on a row (configuration) we have a contact corresponding to each alphabet and state symbol . A constant number of $\vee$ and $\wedge$ gates set the wires on one row in terms of those on the preceding row.

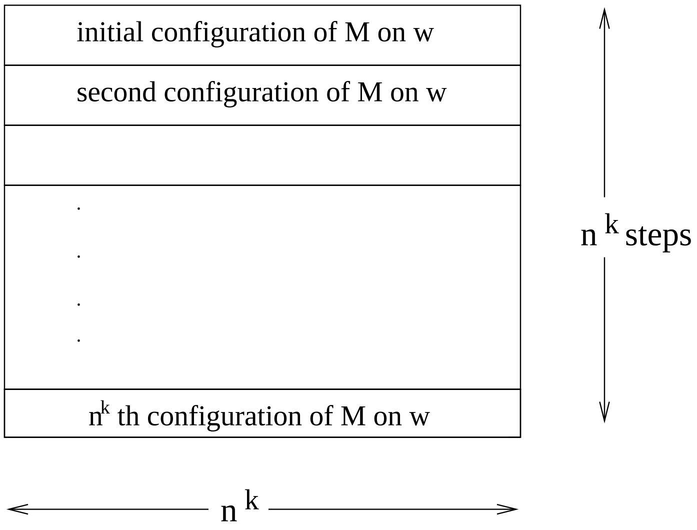


Figure 1 : The computation tableau of M on w

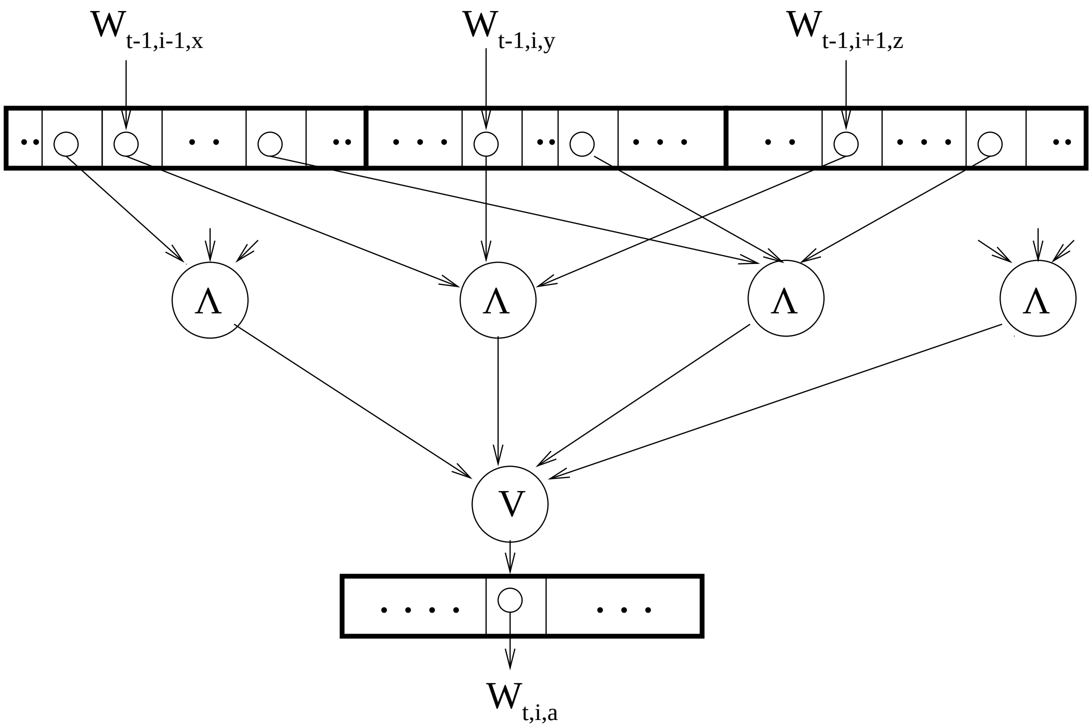


Figure 2: Connecting $W_{t,i,a}$ to the row above it

More formally, for every cell $(t, i)$ in the tableau and every alphabet or state symbol $a$ we have a wire $W_{t,i,a}$. We connect these wires so that contact $W_{t,i,a}$ is turned on iff the value of the $i$'th symbol in the $t$'th configuration is $a$. But this is uniquely determined by the symbols in cells $i-1, i$ and $i+1$ in the previous configuration. To model this, every wire $W_{t,i,a}$ in row $t$ is connected to some wires in row $t-1$ via a constant size 'tree' (subcircuit) of $\vee$ and $\wedge$ gates, as in Figure 2. On the first level of the tree we have a $\vee$ gate and on the next we have $\wedge$'s between all possible triples $W_{t-1,i-1,*}$, $W_{t-1,i,*}$, $W_{t-1,i+1,*}$ that yield $a$ on position $i$ of the next configuration, i.e. $W_{t,i,a}$ (* stands for a generic symbol) .

The input $x$ to the circuit is given by the values $W_{0,*,*}$ of the wires on the first row. All we have to do is set the contacts on the upper level so that they simulate the start configuration. That is we set :

- $W_{0,0,q_0} \leftarrow 1$
- $W_{0,i,w_i} \leftarrow 1,\ i = 1, .., n$
- $W_{0,i,blank} \leftarrow 1,\ i = n+1, .., n^k$
- $W_{0,*,*} \leftarrow 0$ in all other cases.

¿From the construction of the circuit it follows that $w \in A$ iff $W_{n^k,0,q_{accept}} = 1$. For the construction of $C$ we need $O(log(n))$ memory to store indices $i$ and $t$, and for each triple $(t, i, a)$ we need a constant amount of memory to build up the 'trees' for the wires $W_{t,i,a}$. So the reduction is indeed log-space, and the proof is complete. ■

# 2 Alternating Turing Machines

An *Alternating Turing Machine* (ATM) is a natural generalization of the standard nondeternimistic Turing Machine, where all states are labeled $\vee$, $\wedge$ , *accept* or *reject*. That is, beside existential ($\vee$) branches, the Alternating Machine is also allowed universal ($\wedge$) branches.

A computation of an ATM can be visualized as a tree of configurations where all the inner nodes are labeled $\vee$ or $\wedge$, and all the leaves are labeled *accept* or *reject*. Given a computation of an ATM we use boolean logic to evaluate it. A universal $\wedge$ node evaluates to *accept* iff all of its successors evaluate to *accept*. An existential $\vee$ node evaluates to *accept* iff at least one of its successors evaluates to *accept*. A computation is accepting iff its initial configuration evaluates to *accept*, and it is rejecting otherwise. Acceptance is hence determined by evaluating the entire computation tree, from the leaves to the root.

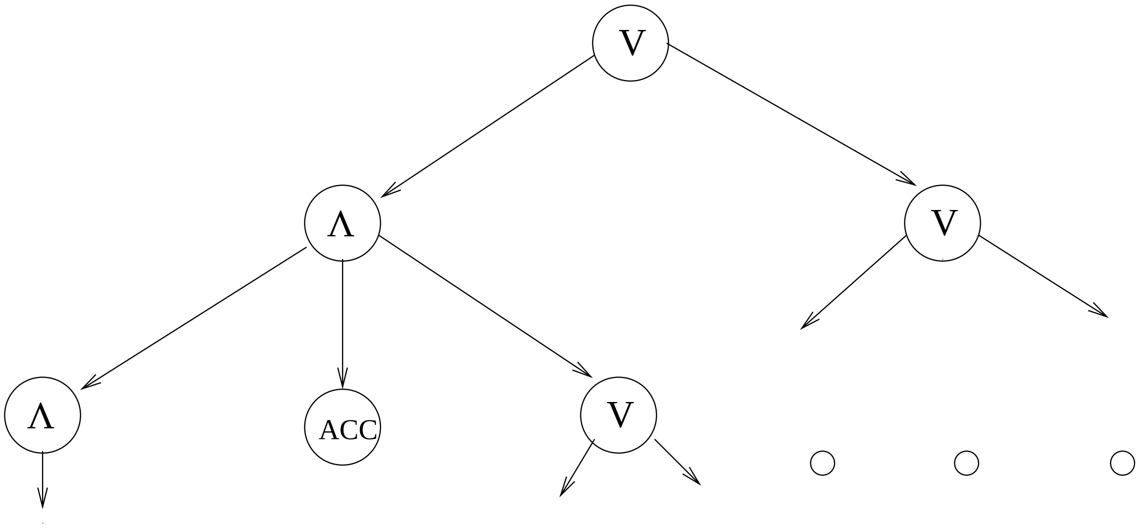


Figure 3: A computation tree of an ATM

Complexity is defined for ATM's in the same way as for NTM's. The running time of an ATM is the length of the longest branch. The space used by an ATM is the size of the largest configuration in its computation tree. The corresponding complexity classes are defined as follows:

$$ATIME(t(n)) = \{A | \exists \text{ ATM that decides A in time } O(t(n))\}$$

$$ASPACE(t(n)) = \{A | \exists \text{ ATM that decides A in space } O(t(n))\}$$

We can thus define alternating complexity classes AL, AP, APSPACE etc. corresponding to the standard complexity classes L, P, PSPACE etc..

**Examples :**

1. $SAT \in AP$. This is clear, since a nondeterministic TM is just a special case of ATM, where all inner states are existential. In fact we have:

   **Fact 1** *$NP \subseteq AP$ .*

2. $\overline{SAT} \in AP$. To see this, replace in the computation tree for SAT all $\vee$ nodes by $\wedge$ and vice-versa ,then take the output level and flip *accept* and *reject* states. The procedure works in general, so it holds:

   **Fact 2** *$CoNP \subseteq AP$ .*

3. $TQBF \in AP$. This is basically because alternation can simulate whatever quantifiers do. Since TQBF is PSPACE-complete, we have that:

   **Fact 3** *$PSPACE \subseteq AP$ .*

   Moreover, we shall see later that in fact:

   **Fact 4** *$PSPACE = AP$ .*

4. $MIN = \{\phi$ : no shorter formula is logically equivalent to $\phi\} \in AP$. We build an ATM to mirror the quantified formula: $\forall$ shorter formula $\psi$ over the same variables, $\exists$ an assignment $x$ to the variables such that $\psi(x) \neq \phi(x)$. This ATM will have two levels of branching: the first is universal, corresponding to all shorter formulae $\psi$ and the second is existential, trying all possible assignments on the variables.

## 2.1 The Relationship Between Alternating and Standard Complexity Classes

The previous examples suggest strong connections between alternating and standard complexity classes. These are incorporated in the following general result:

**Theorem 1** *The following relations between complexity classes hold :*

1. *For $t(n) \geq n$, $ATIME(t(n)) \subseteq SPACE(t(n)) \subseteq ATIME(t^2(n))$;*
2. *For $t(n) \geq \log n$, $ASPACE(t(n)) = \bigcup_c TIME(c^{t(n)})$.*

**Corollary 1** *The following identities are direct consequences of the above theorem:*

- $AL = P$; *( by 2.)*
- $AP = PSPACE$; *( by 1.)*
- $APSPACE = EXPTIME$; *( by 2.)* .

We now proceed to prove the first part of the theorem.

**Proof:**

We begin by showing that $ATIME(t(n)) \subseteq SPACE(t(n))$. The idea is , given an alternating computation tree of height $t(n)$, to evaluate it by performing a depth first search using $O(t(n))$ space for the stack.

Let $A \in ATIME(t(n))$ and let $M$ be an ATM that decides $A$ in time $t(n)$. Given an input $w$ of length $n$ we perform a DFS on the computation tree reusing stack space for each branch. When going down on a branch all we have to do is simulate M, and this uses at most $O(t(n))$ memory.

In order to backtrack, we have to store some data that will enable us to go back from a configuration to its predecessor in the tree. To do that, we store the index of the out-edge we are using in every step of the simulation. This takes a constant number of bits per step, since the out-degree of each configuration is bounded by some constants that only depends on $M$. To backtrack from a configuration in the tree, we simply repeat the whole computation from the initial configuration up to (but excluding) the last step. As all the branches in the tree are of length at most $t(n)$ and we only use constant additional storage space per step, the total memory we use is still $O(t(n))$.

Once we have evaluated all the successors of a given configuration, we can evaluate that configuration too, and backtrack to its predecessor. We keep on doing that, until we evaluate the initial configuration. This proves the first inclusion.

We now prove $SPACE(t(n)) \subseteq ATIME(t^2(n))$ .

Let $A \in SPACE(t(n))$ and let $M$ be a TM that decides $A$ using $t(n)$ space. The configuration tableau of $M$ has $t(n)$ cells on each row and $c^{t(n)}$ on each column, where $c$ is a constant that only depends on $M$.

We construct an ATM accepting the same language as $M$. Basically this ATM should be able to check that we can get from the *start* configuration to the *accept* configuration in time $t^2(n)$. The idea is similar to the one used to show PSPACE-completeness for TQBF. We devise a recursive algorithm that will check if we can get from some configuration $C_i$ to another configuration $C_j$ in $t$ steps, that is if $C_i \xrightarrow{t} C_j$ :

- existentially guess a 'middle' configuration $C_{mid}$ ;
- recursively check if $C_i \xrightarrow{t/2} C_{mid}$ and $C_{mid} \xrightarrow{t/2} C_j$; accept iff both hold.

Now we just have to run the algorithm and check if $C_{start} \xrightarrow{t^2(n)} C_{accept}$. Notice that each recursion step costs $t(n)$ to 'guess' a middle configuration, since each row of the tableau has length $t(n)$. The depth of the recursion is again $t(n)$, so this gives total (alternating) time $t^2(n)$ to simulate $M$.

■