6.841/18.405J: Advanced Complexity	Wednesday, February 12, 2003
Lecture Lecture 3	
Instructor: Madhu Sudan	*Scribe: Bobby Kleinberg*

1 The language MinDNF

At the end of the last lecture, we introduced the language MinDNF, defined as follows

$$\begin{aligned}\text{MinDNF} \quad = \quad & \{(\phi, k) : \ \phi \text{ is a DNF formula and } \exists \text{ DNF formula } \phi' \text{ s.t. } |\phi'| \leq k \text{ and} \\ & \forall x_1 \ldots x_n \ \phi'(x_1, \ldots, x_n) \Leftrightarrow \phi(x_1, \ldots, x_n)\}.\end{aligned}$$

To be precise about some of the terms used above, we recall the following definitions relating to DNF formulae. If $x_1, \ldots, x_n$ are Boolean variables whose negations are denoted by $\overline{x_1}, \ldots, \overline{x_n}$, we refer to the symbols $\{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$ as *literals*. A DNF formula is a Boolean formula expressed as a disjunction of clauses, each of which is a conjunction of one or more literals. As a matter of notation, we use "$\cdot$" to denote logical AND, and "$+$" to denote logical OR. Thus a typical DNF is

$$\phi(x_1, \ldots, x_n) = x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_3} \cdot x_4 \cdot x_5$$

The *size* of a DNF formula ϕ, denoted $|\phi|$, is defined as the number of literals appearing in ϕ (counted with multiplicities).

As we noted last time, the language MinDNF appears to be strictly harder than NP. But a non-deterministic Turing machine with access to an oracle for SAT can accept MinDNF in polynomial time, as follows: first guess the formula ϕ', then use the SAT oracle to verify that $(\phi'(x_1, \ldots, x_n) \Leftrightarrow \phi(x_1, \ldots, x_n))$ is a tautology. Thus we have

$$\text{NP} \leq \text{MinDNF} \leq \text{NP}^{\text{NP}},$$

though it's not obvious which of the above inclusions is strict. In fact, it was shown only recently that MinDNF is NP^{NP}-complete.

The language MinDNF appears to have some extra power beyond that of NP, and the power comes from the fact that it was able to combine an existential step ($\exists \phi'$) with a universal step ($\forall x_1, \ldots, x_n$), whereas NP only has the power to perform existential steps. To formalize this, we introduce the notion of *alternation*.

2 Alternating Turing machines

An alternating Turing machine (ATM) is defined analogously to an ordinary Turing machine (infinite tape, finite alphabet, finite state control) but with two special states, denoted "$\exists$" and "$\forall$". Both of these special states have two outgoing transitions. In state $\exists$, the machine accepts if and only if at least one of the outgoing transitions accepts. In state $\forall$, the machine accepts if and only if all of the outgoing transitions accept.

In thinking about the computation of an alternating Turing machine, it is helpful to represent the computation as a tree, as in figure 1. Each node of the tree is labeled with the machine's configuration (tape contents, position of R/W head, state of control) and has arrows pointing to the configurations reachable by outgoing transitions from that node. The outcome of the computation is determined recursively as follows. A node which is in the machine's accept state q_f accepts. A node in the $\exists$ state accepts if and only if at least one of its children accepts. A node in the $\forall$ state

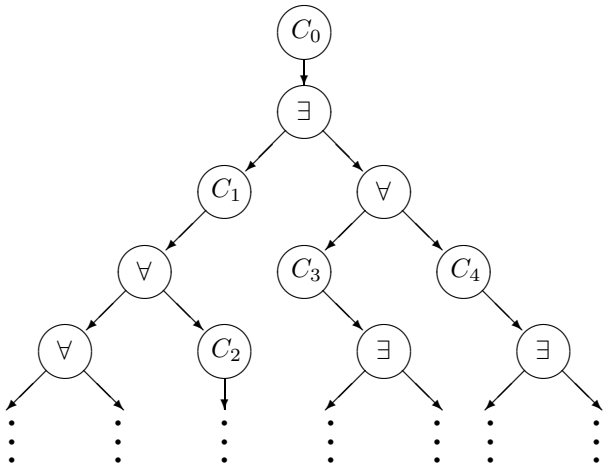

Figure 1: Sample computation tree for an alternating Turing machine

accepts if and only if both of its children accept. Every other node has only one child, and accepts if and only if its child accepts. The machine accepts if and only if the root of its computation tree accepts.

Note that non-deterministic Turing machines may be defined in the same manner, but using only the $\exists$ state without the $\forall$ state. Thus, NP is the class of all languages accepted by a polynomial-time ATM which uses the $\exists$ but not the $\forall$ state. Similarly, co-NP is the class of all languages accepted by a polynomial-time ATM which uses the $\forall$ but not the $\exists$ state.

In studying the computational power of alternating Turing machines, the following resources are of interest:

- **TIME**, defined as the depth of the computation tree.
- **SPACE**, defined as the maximum space used in any configuration in the tree.
- **ALTERNATION**, defined as the maximum (over all computation paths) of the number of transitions from $\exists$ to $\forall$ and vice-versa.

Definition 1. *The class* $\mathrm{ATISP}[a(n), t(n), s(n)]$ *is defined to be the class of languages accepted by an ATM which uses alternation* $\leq a(n)$*, time* $\leq t(n)$*, and space* $\leq s(n)$*. Note that an expression such as* $t(n)$ *appearing in these inequalities is to be interpreted literally as* $t(n)$*, not as* $O(t(n))$*.*

As special cases of this definition, we have:

- $\mathrm{ATIME}(t(n)) := \mathrm{ATISP}[\infty, t(n), \infty]$
- $\mathrm{ASPACE}(s(n)) := \mathrm{ATISP}[\infty, \infty, s(n)]$
- $\Sigma_i^P := \bigcup_{p \in poly} \mathrm{ATISP}[i, p(n), \infty]$ *where the first special state is* $\exists$*.*
- $\Pi_i^P := \bigcup_{p \in poly} \mathrm{ATISP}[i, p(n), \infty]$ *where the first special state is* $\forall$*.*

3 Relationships with classical resources

The classes Σ_i^P and Π_i^P are very important and will be the subject of future lectures. For now, we are interested in exploring the relationship between the classes $\text{ATIME}(t(n)), \text{ASPACE}(s(n))$ and classical complexity classes defined using time- or space-bounded deterministic Turing machines.

3.1 Relating ATIME and SPACE

Theorem 1. *If $s(n) \geq \log n$, $SPACE(s(n)) \subseteq ATIME(O(s(n)^2)) \subseteq SPACE(O(s(n)^2))$.*

Proof. The containment $\text{ATIME}(s(n)) \subseteq \text{SPACE}(O(s(n)))$ is the less interesting half of the theorem, and we will not spend much time on it. The idea of the proof is that a deterministic machine may simulate an alternating machine by performing a depth-first search of its computation tree. If the tree has depth $s(n)$ and we are doing the simulation in a space-efficient manner, we will maintain the complete configuration of the node currently being visited (requiring $O(s(n))$ space), and we will maintain a stack representing the computation path leading to this node. This stack has depth $s(n)$, and each entry in the stack must remember a sufficient set of information to enable the simulation to "back up" to the previous configuration when it pops up the stack. For example, it suffices for each stack entry to store

1. its control state,
2. whether it moved the R/W head left or right,
3. what symbol did it overwrite on the work tape, and
4. how many of its outgoing transitions are already known to be accepting.

Each of these bits of information requires only $O(1)$ space per stack entry.

The more interesting half of the theorem is the containment $\text{SPACE}(s(n)) \subseteq \text{ATIME}(O(s(n)^2))$. To prove this, we must begin by precisely formulating what we mean by "configuration of a Turing machine" and quantifying the number of bits needed to store a configuration. A configuration (depicted in figure 2) is specified by giving the contents of the machine's work tape as a finite sequence of symbols $\sigma_1\sigma_2 \ldots \sigma_{s(n)}$ from the alphabet Σ; if the machine is in state q and its read/write head is situated at position j on the tape, then we replace σ_j in the sequence above with $<\sigma_j, q>$. In the case of space-bounded computation, where the input is given on a separate read-only tape, the machine's configuration also contains a counter representing the read-only head's position on this tape. Thus, if the machine uses $s(n)$ space on its work tape, the configuration is represented by a string of $s(n)$ symbols from a finite alphabet, plus a counter taking values between 0 and n. The number of bits required to represent a configuration is therefore $O(s(n) + \log n)$, which is $O(s(n))$ because of our assumption that $s(n) \geq \log n$. The transcript of a $\text{SPACE}(s(n))$ computation can be represented by a sequence of configurations, each having size $O(s(n))$. The length of the sequence is $2^{O(s(n))}$, since this is an upper bound on the running time of any $\text{SPACE}(s(n))$ computation when $s(n) \geq \log n$.

When describing pseudo-code for alternating Turing machine algorithms, we will write "GUESS" to refer to the machine entering a $\exists$ state, and "FORALL { CHECK1; CHECK2 }" to refer to the machine entering a $\forall$ state in which it accepts if and only if both CHECK1 and CHECK2 accept.

Given a machine M which decides a language L in $\text{SPACE}(s(n))$, we must exhibit an ATM which accepts L and runs in time $O(s(n)^2)$. The algorithm will be based on the following procedure $\text{CHECK}(C_0, C_f, T)$ which checks whether M has a computation path which starts in configuration C_0 at time 0 and ends in C_f at time T, by guessing the configuration at the midpoint of the computation and recursively checking both halves.

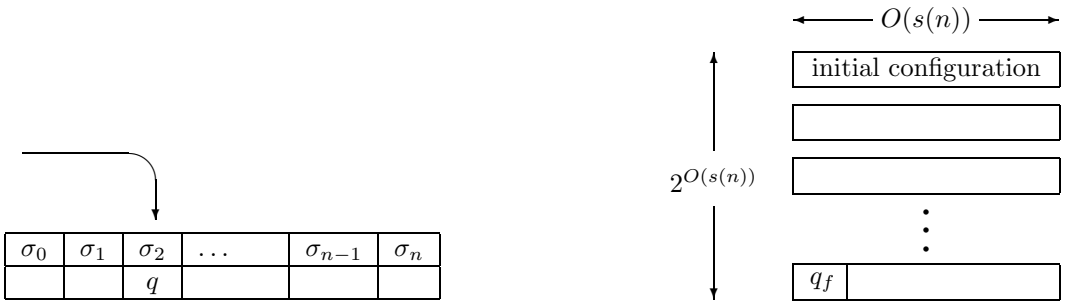

Figure 2: Turing machine configuration and computation transcript

CHECK(C_0, C_f, T):

1. GUESS a configuration C_{mid}.
2. FORALL $\{$CHECK$(C_0, C_{\text{mid}}, \lfloor T/2 \rfloor)$; CHECK$(C_{\text{mid}}, C_f, \lceil T/2 \rceil)\}$

The running time of CHECK(C_0, C_f, T) is $O(s(n) \log T)$, as is easily established by induction. Indeed, step 1 requires time $O(s(n))$ — the size of C_{mid} — and, by the induction hypothesis, step 2 requires time $O(s(n) \log(T/2)) = O(s(n) \log T) - O(s(n))$.

An ATM which runs in time $O(s(n)^2)$ and simulates M can now be described as follows: guess an accepting configuration C_f, and run CHECK$(C_0, C_f, 2^{O(s(n))})$. The initial guess of C_f requires time $O(s(n))$, and the CHECK step requires time $O(s(n) \log(2^{O(s(n))})) = O(s(n) \cdot s(n)) = O(s(n)^2)$, as desired. □

4 A complete problem for ATIME(poly)

Theorem 1 establishes that ATIME(poly) = PSPACE, which in turn sheds light on the phenomenon of PSPACE-completeness. Define TQBF to be the language

$$\text{TQBF} = \{\phi | \exists x_1 \forall x_2 \exists \ldots \forall x_n \ \phi(x_1, \ldots, x_n) \text{ is true}\}.$$

Note the analogy with SAT, which is the language

$$\text{SAT} = \{\phi | \exists x_1 \ldots x_n \ \phi(x_1, \ldots, x_n) \text{ is true}\}.$$

Just as SAT is NP-complete, it is natural to guess that TQBF is ATIME(poly)-complete, since it captures the expressive power of alternating quantifiers in much the same way that SAT captures the expressive power of existential quantifiers. This indeed turns out to be the case, as we will see in Theorem 2 below. Thus, the PSPACE-completeness of TQBF follows directly from the fact that ATIME(poly) = PSPACE, and indeed it is tempting to think of this as the "real reason" why TQBF is PSPACE-complete.

Theorem 2. *TQBF is ATIME(poly)-complete.*

Proof Sketch. It is clear how to build an ATM that decides TQBF in polynomial time: the machine passes through an initial sequence of $\exists$ and $\forall$ states in which the truth values of $x_1, \ldots, x_n$ are set, then runs a polynomial-time computation to determine the truth value of $\phi(x_1, \ldots, x_n)$.

To prove that every language in ATIME(poly) is reducible to TQBF, proceed as in the proof of the Cook-Levin theorem. The computation of an ATM which runs in time $t(n)$ can be represented by a sequence of $t(n)$ configurations, each of size $O(t(n))$. One introduces $O(t(n)^2)$ Boolean variables to completely describe the machine's configuration at each time step, and writes down a formula

which expresses the fact that the computation started in configuration C_0, finished in an accepting configuration, and performed legal state transitions at every step. The alternating quantifiers are used to express the possible outcomes of the steps in which the computation entered a $\forall$ or $\exists$ state. One verifies, by construction, that the size of the formula is polynomial in the input length, and that the quantified formula is true if and only if the ATM accepts that input. $\square$

Extending the intuition we've gained from SAT and TQBF, it is natural to define the following two languages $i\exists$TQBF, $i\forall$TQBF and to guess that they are complete for Σ_i^P, Π_i^P, respectively. In a future lecture, we'll see that this is the case.

Definition 2. *The classes $i\exists TQBF$ and $i\forall TQBF$ are defined by*

$$\begin{aligned} i\exists TQBF &= \{\phi | \exists x_1^1 \dots x_{n_1}^1 \forall x_1^2 \dots x_{n_2}^2 \dots\dots \exists x_1^i \dots x_{n_i}^i \ \phi(x_1^1, \dots, x_{n_i}^i) \text{ is true}\} \\ i\forall TQBF &= \{\phi | \forall x_1^1 \dots x_{n_1}^1 \exists x_1^2 \dots x_{n_2}^2 \dots\dots \forall x_1^i \dots x_{n_i}^i \ \phi(x_1^1, \dots, x_{n_i}^i) \text{ is true}\} \end{aligned}$$

5 Philosophical aside: debate systems

Resources studied in complexity theory are interesting to the extent that they characterize the expressive power of interesting phenomena, and one is led to ask what type of phenomenon is characterized by the resource of alternation. One way of describing the answer is in terms of "debate systems."

Imagine a scenario in which a verifier V (with the power to perform deterministic polynomial-time computations) is charged with determining the truth or falsehood of a statement of the form "$x \in L$," where x is a word and L is a language. The verifier interacts with two entities, Y and N, both of whom have unlimited computational power. Y does everything in her power to convince V that the answer is yes, while N does everything in her power to convince V that the answer is no. V assumes that Y and N have infinite computational power, and that they are both trying as hard as possible to argue for their respective answers. For what languages can V discover, for each x, whether $x \in L$? The answer depends on the type of interaction allowed between V, Y, and N.

No interaction: In this case, V must decide on its own whether $x \in L$; the answer can be determined if and only if L is in P.

One-sided interaction with Y: Suppose Y is allowed to send advice to V, but N is not allowed to refute the advice. The class of languages which can be decided by such a protocol is NP. For instance, if L is SAT and ϕ is a formula, Y will try to send V a satisfying assignment. If ϕ is satisfiable, V can verify this in polynomial time using Y's advice. If ϕ is not satisfiable, V can verify that Y's assignment does not satisfy ϕ, and will conclude that $\phi \notin SAT$ based on the assumption that Y is omnipotent and is trying as hard as possible to argue that $\phi \in$ SAT.

Full-fledged debate between Y and N: If Y and N are both allowed to send messages to V, and to respond to each other's messages, the class of languages which can be decided by such a protocol is ATIME(poly) = PSPACE. As an example, if (ϕ, k) is an instance of MinDNF, then Y begins by naming a formula ϕ' of length $\leq k$ and asserting that $\phi' \Leftrightarrow \phi$. N attempts to refute this claim by sending an assignment $x_1, \dots, x_n$ to distinguish ϕ' from ϕ. V computes $\phi'(x_1, \dots, x_n)$ and $\phi(x_1, \dots, x_n)$, and tests whether they have the same truth value. If so, then N failed to refute Y and we conclude that $(\phi, k) \in$ MinDNF. If not, then N's refutation was successful and we conclude that $(\phi, k) \notin$ MinDNF.

6 Space-bounded alternating computation

Theorem 1 established a relation between time-bounded alternating computation and space-bounded deterministic computation. We will now relate *space*-bounded alternating computation to *time*-

bounded deterministic computation!

Theorem 3. *Assume $s(n) \geq \log n$. Then ASPACE($O(s(n))$) = TIME($2^{O(s(n))}$).*

Proof. We will abbreviate $2^{O(s(n))}$ as 2^s for simplicity. Once again, we can prove the containment ASPACE($O(s(n))$) $\subseteq$ TIME(2^s) by reasoning about computation trees for ATM's. Actually, this time we represent the computation using a directed graph rather than a tree. An ATM which runs in space $O(s(n))$ has 2^s possible configurations (recalling that $s(n) \geq \log n$), and one can define the directed graph G whose vertices are the possible configurations and whose edges represent the legal transitions from one configuration to another. There is a straightforward deterministic algorithm to enumerate all the vertices and edges of this graph in time 2^s. One can then mark all of the vertices which represent accepting configurations, via the following algorithm:

1. Set $t = 0$.
2. For each vertex v of G:
   (a) If M's control state at v is the accepting state, mark v.
   (b) If the state at v is "$\exists$", and at least one edge (v, w) points to a marked vertex w, mark v.
   (c) If the state at v is "$\forall$", and both edges (v, w) point to a marked vertex w, mark v.
   (d) Else there is a unique edge (v, w). If w is marked, then mark v.
3. Increment t. If $t > |V(G)|$, halt; else return to step 2.

The inner loop runs in time 2^s, the time required to look up w and check whether it is marked. There are 2^s iterations of the outer loop, and each contains 2^s iterations of the inner loop. Thus the entire algorithm runs in time $2^{O(s)}$, as desired.

To prove the containment TIME(2^s) $\subseteq$ ASPACE($O(s(n))$), we model the computation of a deterministic Turing machine M as a tableau whose rows represent the sequence of configurations. (See figure 3.) If the Turing machine runs in time 2^s, then it also runs in space 2^s, so the tableau has 2^s rows and 2^s columns. Naively, the algorithm should guess the contents of the entire tableau and verify the tableau's correctness; however, guessing the tableau's contents all at once would require space 2^s. The key observation which makes the simulation space-efficient is that the correctness of a cell in the tableau can be checked by looking at just three cells in the row above. Formally, the ATM uses a subroutine CHECK(t, i, σ) defined as follows and illustrated in figure 3.

CHECK(t, i, σ)
1. GUESS $\{\sigma_1, \sigma_2, \sigma_3\}$
2. Verify that $(\sigma_1, \sigma_2, \sigma_3) \rightarrow \sigma$ is a legal transition
3. FORALL $\{\text{CHECK}(t-1, i-1, \sigma_1); \text{CHECK}(t-1, i, \sigma_2); \text{CHECK}(t-1, i+1, \sigma_3)\}$.

Note that CHECK requires $O(s(n))$ bits to store t and i, and this is really the only space required by the algorithm. Let's adopt, for simplicity, the convention that an accepting computation halts in state q_f, with the read/write head at the leftmost cell of the tape. In that case, CHECK$(2^s, 0, q_f)$ runs in space $O(s(n))$ and determines whether M accepts the input, as desired. □

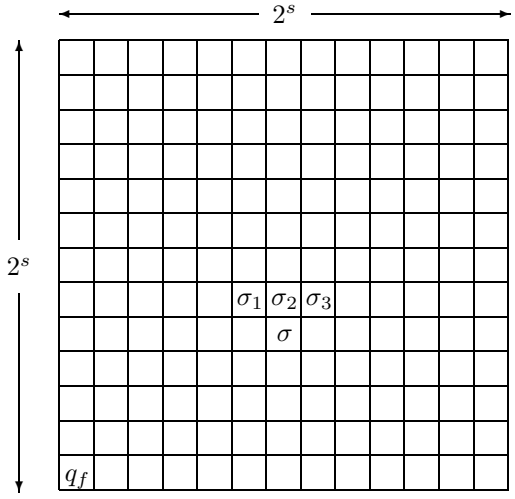

Figure 3: Tableau with CHECK(t, i, σ) configuration