This is more material from Chapter 5 of the text, and the beginning of Chapter 6.

1 Other characterizations. Alternating Turing Machines

We saw that we can define sets of languages $\Sigma_i^p$ and $\Pi_i^p$ for $i = 1, 2, \ldots$ by alternating sequences of polynomially bounded quantifiers. These are generalizations of $NP = \Sigma_1^p$ and $coNP = \Pi_1^p$, and the extension of our belief (bet?) that $P \neq NP \neq coNP$ is that this is an infinite hierarchy, with $\Sigma_{i+1}^p \supset \Sigma_i^p$, $\Pi_{i+1}^p \supset \Pi_i^p$ and $\Sigma_i^p \neq \Pi_i^p$, with neither contained in the other.

1.1 Alternating Turing Machines

We can build a Turing machine model that subsumes both nondeterministic and co-nondeterministic computations.

We will generalize the ‘nondeterministic Turing machine’ model. One way of looking at this model (in the time bounded case) is by observing the directed graph of its configurations, as the machine computes. We saw that this can be made into a dag (directed acyclic graph), or even a tree, by recording
the previous configurations that led to a given configuration. We now claim that again, without loss of generality, we can assume that a nondeterministic Turing machine has 1 or 2 possible transitions from a given configuration. The trick is one from data structures: every tree can be represented by a binary tree, and the transformation is easy. See for example, https://xlinux.nist.gov/dads/HTML/binaryTreeRepofTree.html (based on Knuth’s exposition in The Art of Computer Programming.

We can now model a nondeterministic time-bounded computation as follows: consider the possible configurations of a machine $M$ on input $x$ of length $n$ in a computation which takes $T$ steps. Starting with the initial configuration ($M$ has blank worktapes, the input tape contains $x$, $M$ is in its initial state $q_0$ and is scanning the first symbol of $x$) we build the rooted tree with this root. Every vertex is a configuration $C$, and its two children are the two configuration reachable from $C$ in a single step (there is no loss of generality in allowing every move to be nondeterministic—for deterministic moves we just record the choice and do the same action – remember that we are recording all choices on a worktape. The resulting graph is a tree of depth $T$.

Now put labels on this tree according to the following recursive rule:

- at the leaves (depth $T$ vertices): if the configuration is accepting label the vertex with 1, otherwise label it with 0
- for all other nodes, if at least one of its two children have label 1, it gets the label 1, otherwise (if both have label 0) it gets the label 0

The output of $M$ is the label of the root.

It should be clear that this labeling process results in exactly the same string being accepted as in the the standard definition of nondeterministic acceptance. (Prove!)

We could have a similar machine model for co–NP, by slightly changing the labelling rules:

- at the leaves (depth $T$ vertices): if the configuration is accepting label the vertex with 1, otherwise label it with 0

1This process is more powerful than nondeterministic computation. We can get equivalence if we treat 0 as ‘unknown’.
• for all other nodes, if both of its two children have label 1, it gets the label 1, otherwise (if at least one has label 0) it gets the label 0

Again, if we treat label 0 as ‘unknown’ we get exactly \( \text{coNP} \).

We can go further: have vertices of both kinds – some want one successful child, some want both to be successful. Formally: partition the set of states \( Q \) of \( M \) into existential states and universal states. Call a configuration existential or universal according to the kind of the state in the configuration. model is the alternating Turing machine. Acceptance is defined by having the root receive the label 1, under the rules

• at the leaves (depth \( T \) vertices): if the configuration is accepting label the vertex with 1, otherwise label it with 0

• for all other nodes,
  
  – if the configuration is universal, if both of its two children have label 1, it gets the label 1 otherwise (if at least one has label 0) it gets the label 0
  
  – if the configuration is existential, if at least one of its two children have label 1, it gets the label 1, otherwise (if at least one has label 0) it gets the label 0

It should be clear that polynomial time bounded alternating Turing machines recognize exactly the languages in \( \text{PSPACE} \).

(Proof sketch: The tree can be evaluated using space proportional to the depth of the tree time. times the space needed to compute transitions. For the other direction, it should be clear how to recognize \( \text{TQBF} \).)

It is also clear that we can assume that in the whole tree existential and universal configurations alternate along paths from the root to a leaf (use unnecessary branchings if needed–this at most doubles the depth...) More generally we can assume that at any given depth, either all configurations are universal, or all configurations are existential (Prove! – hint: use needless transitions)

It should be also clear that, in analogy with our constructions for \( \text{NP} \) and \( \text{coNP} \) we could define restricted alternating Turing machines for each level of the Polynomial Hierarchy. For example, we could define \( \Sigma_p^2 \)-machines, as alternating Turing machines that run in polynomial time, and along every
path from the root to a leaf we have a sequence of existential configurations followed by universal configurations.

1.2 Complete problems and oracles

There are complete problems for all levels of the hierarchy. For example the language complete for $\Sigma^p_2$ is

$$\{x \exists u_1 \exists u_2 \cdots \exists u_n \forall v_1 \forall v_2 \cdots \forall v_n \Phi(u_1, u_2, \cdots, u_n, v_1, v_2, \cdots, v_n, x)\}$$

with $\Phi$ a 3-CNF formula.

The proof that this language is complete is totally analogous to the proof of the Cook-Levin Theorem: one encodes the computation of the machine into circuits, the builds the 3-CNF formula that asserts that the circuit is correct.

The complete language for $\Sigma^p_i$ is similar, starting with existential quantifiers, and having $i-1$ alternations of existential and universal quantifiers, followed by a 3-CNF formula. As with $NP$ and $coNP$, the complete language for the $\Pi^p_i$ classes is the negation of the logical formula: for example the complete language for $\Pi^p_2$ will be of the form

$$\{x \forall u_1 \forall u_2 \cdots \forall u_n \exists v_1 \exists v_2 \cdots \exists v_n \Phi(u_1, u_2, \cdots, u_n, v_1, v_2, \cdots, v_n, x)\}$$

One can than use such complete languages as oracles: it is easy to see that one could also define $\Sigma^p_{i+1} = NP^{\Pi^p_i}$. This is immediate from the definition of $\Sigma^p_{i+1}$.

In fact, since oracles give both Yes and No answers, one also has $\Sigma^p_{i+1} = NP^{\Sigma^p_{i+1}}$. In particular, $\Sigma^p_2 = NP^{NP}$.

Finally we have a curious little

**Lemma 1.** If $PH$ has a complete language, then it collapses (the hierarchy is not infinite).

**Proof.** If $H$ is complete for $PH$, $L \in \Sigma^p_i$ for some $i$. But if $L$ is complete for $PH$, then, in particular, every language in $\Sigma^p_{i+1}$ is reducible to $L$, and so is in $\Sigma^p_i$. But then $\Sigma^p_{i+1} = \Sigma^p_i$, and the hierarchy collapses at level $i$. \qed

4
1.3 Comments

2 Boolean Circuits

We are all familiar with the binary Boolean gates AND, OR and the unary NOT. We may build Boolean circuits from these gates.

Claim 2. Let \( f \) be any Boolean function of \( n \) variables (\( f : \{0, 1\}^n \rightarrow \{0, 1\} \)). Then there is a Boolean circuit that computes \( f \). Moreover, the circuit can be written as an OR of terms, where each term is an AND of literals, and a literal is a variable or a negated variable.

Proof. Let \( Z = \{ z \in f^{-1}(1) \} \). That is \( Z \) is the set of \( z \in \{0, 1\}^n : f(z) = 1 \). Corresponding to each such \( z \) consider the formula \( f_z = \bigwedge_{i=1}^n u_i \) where \( u_i = x_i \) if the \( i \)-th bit of \( z \) is 1, and \( u_i = \overline{x_i} \) if the \( i \)-th bit of \( z \) is 0. Now it is clear, by construction that \( f_z(z) = 1 \) and for any \( w \neq z \) we have \( f_z(w) = 0 \). Therefore the Boolean function \( f \) can be computed by the formula

\[
    f(x) = \bigvee_{z \in Z} f_z(x)
\]

(Negation and De Morgan’s Laws give us the result that \( f \) could also be computed by a CNF formula.)

It should be clear that a series of ANDs or a series of ORs can be computed by a Boolean circuit with binary AND and OR circuits. This gives us the circuit we believe to exist.

In general these CNF or DNF circuits are far from optimal (in the number of gates needed.) We only promised to show that for any Boolean function of \( n \) variable there is an \( n \)-input Boolean circuit that computes it, and we have done so.

2.1 The Computer Architecture Excuse

I argued in class that the claim above, plus basic Architecture course show that any computer is really a computer, or, better, that any computer, working in a reasonable (polynomial) amount of time is equivalent to a Boolean circuit.

In a nutshell, a computer is a bunch of registers, with Boolean functions between them, and code that tells which of the Boolean functions to apply...
(which registers provide the input bits and which register to store the output bits.) These ‘details’ are dealt with in Computer Architecture courses, but us, theoreticians, we agree that any such computation is just a big Boolean circuit.

So we have found another model of computation: for every $n$ a Boolean circuit that deals with $n$ bit inputs.

**Church-Turing Thesis, revisited** Any computation can be done by Boolean circuits.

A version of the polynomial time Church-Turing thesis (namely, that all feasible computations are the class $P$) is that all feasible computations can be done by polynomial size circuits.

Note that the converse is not true. Since circuits are not uniform, the noncomputable function \textit{unaryHalt}(x) = if $x = 1^n$ then $HALT(n)$ else 0

(where $HALT(n) = 1$ iff the $n$-th Turing machine halts on input $n$) has very small circuits....