IAS/PCMI Summer Session 2000
Clay Mathematics Undergraduate Program
Basic Course on Computational Complexity

Lecture 3: Nondeterministic Computation

David Mix Barrington and Alexis Maciel
July 19, 2000

1. Definitions and Examples

In the previous lecture, we saw a decision algorithm for the 3-colorability problem based on the exhaustive search method. That algorithm ran in exponential time because, in the worst case, it had to examine every possible 3-coloring of the input graph. Can we design an algorithm for 3-colorability that runs in polynomial time? The answer to this question is not known. Nevertheless, we can say something more about the complexity of this problem.

The 3-colorability problem has the following interesting property: if a graph is 3-colorable and if somehow we can find a 3-coloring of the graph, then we can verify, in polynomial time, that the 3-coloring is valid and that the graph is indeed 3-colorable. What we are going to do in this lecture is add to our model of a machine the ability to correctly *guess* a 3-coloring of the graph, if one exists. A machine for 3-colorability will then proceed as follows: it will attempt to guess a 3-coloring, verify that whatever it guessed is indeed a 3-coloring and accept if it is. If a graph is 3-colorable, then there is a way for the machine to accept it. If a graph is not 3-colorable, then whatever it does, the machine will reject it. So the graphs that can possibly be accepted are precisely those that are 3-colorable.

How will we incorporate this guessing ability into the definition of a machine? Recall that our current model of a machine is deterministic: for each observable state, the machine has exactly one behavior. We define a *nondeterministic* machine by specifying any number of possible behaviors for each observable state. So at each step in the computation, a nondeterministic machine may have several options.

In the deterministic case, we say that a machine decides a language if it accepts every string in the language and rejects all the others. In the nondeterministic case, we say that a machine defines a language: the set of all strings it may accept. In other

words, a string is in the language of a nondeterministic machine if and only if there is a sequence of choices that leads the machine to accept the string. All other strings, for every possible sequence of choices, will either be rejected, result in a nonterminating computation, or in one that gets stuck somewhere (because no behavior is defined for the corresponding observable state).

Here's an example. Suppose we are given two strings x and y as input, separated by some marker, and we want to decide whether the x appears as a substring in y. (x is a substring of y if y is of the form uxv, for some strings u and v.) A machine could solve this problem by trying all possible starting locations for x in y. But notice that this may require many passes through large portions of y. In fact, this algorithm runs in time $\Omega(n^2)$.

A simpler machine that goes through its input only once can be designed using nondeterminism. The machine simply scans y until it guesses it has found the starting location of x. It then verifies that the guess was correct. If x does appear in y, then some guess will be correct and lead the machine to accept. Otherwise, all guesses will be wrong and cause the machine to reject.

We say that a nondeterministic machine runs in time f if it takes at most f steps before it halts, for every possible sequence of choices. Similarly for space. For example, the above nondeterministic machine for the substring problem runs in time $O(n)$.

NTIME(f) is the class of languages defined by nondeterministic machines that run in time $O(f)$. Similarly for NSPACE(f). Some interesting classes are NP = NTIME($n^{O(1)}$), NSPACE(1), NL = NSPACE($\log n$), and NPSPACE = NSPACE($n^{O(1)}$). A subclass of NSPACE(1) corresponds to the nondeterministic counterpart of the DFA. An NFA is a DFA that has any number of possible behaviors for each observable state.

Now let's go back to 3-colorability. As mentioned earlier, the idea behind a nondeterministic machine for this problem is to attempt to guess a 3-coloring and verify that whatever was guessed is indeed a 3-coloring. Given a graph with n nodes, the guessing stage can be implemented by having the machine write in its memory a string of n symbols each chosen from a set of 3. This string will be interpreted as an assignment of colors to the nodes of the graph. The nondeterminism is in the choice of which color to write for each node. Once the color assignment is written it is an easy matter to verify whether it constitutes a 3-coloring. Thus 3-colorability is in NP.

As another example, consider the graph reachability problem of the previous lecture. There we showed that the problem is in P. By using nondeterminism, we can

obtain a machine that is very efficient in terms of space. The idea is to have the machine simply guess a sequence of nodes through the graph and then verify whether this sequence constitutes a path from s to t. The key to using a small amount of space is to not write the entire sequence of nodes right away but write it one node at a time, as needed. If the graph has n nodes, then each node can be identified by a $\log n$-bit number. Thus graph reachability is in NL.

We have formally defined nondeterministic machines, and the languages they define, but how are we supposed to think about the computation of a nondeterministic machine? One possibility is to pretend that the machine has the magical ability to make the correct choices, if these exist. Or that the machine has access to a mysterious guide that whispers in its ear the correct choices. If the input can be accepted, the machine will find a way. If the input cannot be accepted, the machine will end up rejecting no matter what it does.

Another possibility is to imagine that every time the machine is faced with a choice, it splits itself in two so it can explore both options at the same time, in parallel. Or, equivalently, we can imagine that the nondeterministic machine is actually not a single machine but a collection of deterministic machines operating in parallel. Every time a choice needs to be made, a different machine will be assigned the task of continuing the computation according to each possible option. In either case, the input is accepted if any of the parallel computations leads to acceptance.

In any case, it is important to realize that nondeterministic machines do not constitute a reasonable notion of algorithm. After all, real world computers are deterministic. For this reason, we have avoided saying that nondeterministic machines decide languages and used instead the terminology that nondeterministic machines *define* languages. If a nondeterministic machine is going to give us an algorithm for the language it defines, we will have to simulate it using a deterministic machine. One simple way of doing this is to simply try all possible sequences of choices, being careful not to get stuck prematurely in nonterminating computations. In the next section, we will see that this can be done but with a huge loss of efficiency. For example, NP machines will give us exponential-time algorithms. Since a large number of problems of significant practical interest do have NP machines but no known polynomial-time algorithm, the question of whether nondeterminism can be simulated more efficiently, without this exponential loss, is of great importance.

2. Deterministic Simulations

Now that we have added to our model of a machine this mysterious ability to make correct guesses or choices, it is appropriate to ask how much power we have really added to our machines? In this section we provide a partial answer that depends on the complexity bounds involved.

First, in the case of finite automata, we show that NFA's can be simulated by DFA's. Thus, in this case, nondeterminism does not add any more power. Note, however, that the DFA may have a much larger (but still constant) number of states than the NFA. In instances where this blow-up in the number of states is not a problem, then nondeterminism in finite automata can be viewed as a convenient tool since NFA's are sometimes easier to design and understand than equivalent DFA's.

We say that an NFA and a DFA are *equivalent* if the DFA decides the language of the NFA.

Theorem 1 *Every NFA has an equivalent DFA.*

Proof The idea is that the DFA will read the input one symbol at a time keeping track of the states in which the NFA could be. Initially, the NFA is in its start state. At any later moment, the NFA could be in any of a certain subset of its states. The states of the DFA will correspond to subsets of the states of the NFA.

The behavior of the DFA is to update the set of current possible states of the NFA. More precisely, suppose that the DFA is in state $\{q_1, \ldots, q_k\}$ and sees input symbol a, where $q_1, \ldots, q_k$ are states of the NFA. And suppose that the behavior of the NFA for state q_i and symbol a is to go to any of the states in the set Q_i. Then the behavior of the DFA for that observable state is to go to state $Q_1 \cup \cdots \cup Q_k$. The accepting states of the DFA are all the subsets that contain at least one accepting state of the NFA. It is clear then that the DFA will accept if and only if the NFA could have accepted. □

We now present a more general deterministic simulation. We show that nondeterministic machines running in time t can be simulated by deterministic machines running in time $2^{O(t)}$. Therefore, for example, NP machines can be simulated in deterministic exponential time. So, in a sense, NP machines correspond to a special kind of exponential-time algorithm. As mentioned earlier, whether or not these algorithms can be simulated more efficiently, perhaps in polynomial time, is an open question of great importance.

Theorem 2 *Every nondeterministic machine running in time t has an equivalent deterministic machine that runs in time $2^{O(t)}$.*

Proof Let N be a nondeterministic machine and k the maximum number of behaviors N has for any observable state. Every sequence of choices of N can be represented as a string of numbers from $\{1, \ldots, k\}$. We design a deterministic machine D that simulates N on every possible sequence of choices, starting with those of length 1, then those of length 2, 3, etc. If N accepts its input, it does so for a sequence of choices of length no greater than t. If N does not accept its input, then D will know that fact after having gone through all the sequence of choices of length at most t.

To carry out the simulation, D will divide its memory in two parts, one for the current sequence of choices, the other for the contents of the N's memory. For each sequence of choices, D will erase the N's memory, simulate N from its initial configuration and then update the sequence of choices to the next one. All this can be done in time $O(t)$. Since the number of sequences of choices of length at most t is at most k^{t+1}, the entire simulation can be carried out in time $2^{O(t)}$. □

3. Different Perspectives on Nondeterminism

Earlier we provided two different ways in which one can view the computation of a nondeterministic machine. One way referred to a magical power or all-powerful guide. The other was in terms of parallel computation. In either case, we were still trying to see a nondeterministic machine as a device that accepts or rejects inputs, a *decider*. In this section, we present two alternative perspectives on nondeterminism in which nondeterministic machines are no longer viewed as deciders. We will focus on the case of NP.

First, recall that we can verify in polynomial time that a graph is 3-colorable, as long as we are provided with a 3-coloring of the graph. A polynomial-time *verifier* for 3-colorability is a polynomial-time machine that given a graph and a string interprets the string as a color assignment to the nodes of the graph and checks to see that this color assignment is a 3-coloring. More generally, an algorithm V is a verifier for language L if $x \in L$ precisely when V accepts $\langle x, c \rangle$ for some c. c is called a certificate or proof of membership for x. The running time of a verifier is measured only in terms of the length of x. NP consists exactly of those languages that have polynomial-time verifiers.

Second, suppose you are playing the following solitaire game with graphs. You generate an arbitrary graph (never mind how) and try to 3-color it. You win if you

succeed, you lose if you fail. The graphs for which you can win are of course precisely those that are 3-colorable. Now generalize the game as follows. Consider an arbitrary nondeterministic machine. Each machine yields a different game. For a particular machine, given an arbitrary input, you make the machine's choices and you win if the machine ends up accepting. The set of inputs for which you can win defines a language. NP is the class of languages that can be defined by such games when played on nondeterministic polynomial-time machines.

4. Exercises

1. Given a list of numbers and a number k, the *subset sum* problem is to determine whether there a subset of the numbers that adds to exactly k. For example, the answer is yes for $\langle(3, 4, 12, 7, 4), 20\rangle$ and no for $\langle(3, 4, 12, 7, 4), 6\rangle$.

   (a) Show that the problem is in NP if the input numbers are given in binary.

   (b) Show that the problem is in P if the input numbers are given in unary.

2. As defined in the lecture, a nondeterministic machine runs in time f if for every possible sequence of choices, the machine halts after at most f steps. An alternative definition is to say that a nondeterministic machine runs in time f if for every possible sequence of choices *that leads to accept*, the machine halts after at most f steps. In other words, for those sequence of choices that do not lead to accept, the machine may take longer and is not even required to halt. Show that the class NTIME(f) is the same using either definition if f is assumed to be *time constructible*. That is, if given 1^n, the binary representation of $f(n)$ can be computed in time $O(f(n))$.

3. (a) Define (in full detail) an NFA with 3 states for the following language: the set of binary strings that contain 110 as a substring.

   (b) Convert this NFA to an equivalent DFA by using the algorithm presented in this lecture.

4. In an undirected graph, a set of nodes is called a *clique* if every two of its nodes is connected by an edge. Given an undirected graph G and a number k, the *clique* problem is to determine whether G contains a clique of size k and the *maximum clique* problem is to determine whether the largest clique of G has size k. While it is easy to see that clique is in NP, it is not known whether or not maximum clique is. Show that if P = NP, then maximum clique is in P and

there is a polynomial-time algorithm that given a graph finds one of its largest cliques. (Hint: Keep in mind that P is closed under complementation.)

5. Show that a language is in NP if and only if it has a polynomial-time verifier.