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

Lecture 2: The Complexity of Some Problems

David Mix Barrington and Alexis Maciel
July 18, 2000

1. Parity

The *parity* of a bit string is simply whether the number of 1's in that string is odd or even. The parity problem is to determine the parity of an input bit string. The corresponding formal language is the set of all bit strings that have *odd* parity.

We first present a machine that decides parity. The machine scans the input from left to right. After looking at each bit, the machine remembers whether the number of 1's seen so far is odd or even. Initially, the number is 0 and therefore even. The machine then toggles between even and odd for each 1 seen in the input, and does nothing for each 0. This even/odd toggle can be incorporated into the internal states of the machine or it can be stored explicitly in its memory. In either case, the machine uses a constant amount of space and parity is in DSPACE(1).

We now construct a family of circuits for parity. Let $x = x_1 \cdots x_n$ be an input bit string of length n. Considering addition modulo 2, we want the circuit to output the sum of the input bits. Since addition modulo 2 is associative, we have that $x_1 + \cdots + x_n = (x_1 + \cdots + x_{n/2}) + (x_{n/2+1} + \cdots + x_n)$. Using this idea repeatedly, we see that parity can be computed with a binary tree of *parity gates*. The depth of this tree is $O(\log n)$. To obtain a Boolean circuit, we only need to show that these parity gates of fan-in 2 can be implemented using AND, OR and NOT gates of fain-in at most 2. This is easy since $a + b = 1 \pmod 2$ if and only if $(a \wedge \neg b) \vee (\neg a \wedge b)$. Therefore, parity is in NC^1.

2. Addition

Given two n-bit binary numbers, the *addition* problem is to determine their $(n+1)$-bit sum. Note that we are not deciding a language here but computing a function. An

n-bit binary number $a = \sum_{i=0}^{n-1} a_i 2^i$ will be represented in the usual way by the string $a_{n-1} \cdots a_0$. We refer to a_i as the ith bit of a.

Since the input consists of two numbers, we need to agree on a way of encoding two numbers as a single input string. One way is to simply concatenate the n-bit binary representations of the two numbers, giving us a $2n$-bit input string. In other words, if the two input numbers a and b have binary representations $a_{n-1} \cdots a_0$ and $b_{n-1} \cdots b_0$, then the pair $\langle a, b \rangle$ will be encoded as $a_{n-1} \cdots a_0 b_{n-1} \cdots b_0$. The results presented in this section apply equally well to any other reasonable encoding scheme.

We begin with a machine that computes addition. To simplify things, we will start by describing a machine that produces the output in reverse order, i.e., with the 0th bit first and the nth bit last. By examining a_0 and b_0, the machine can determine the 0th bit of the sum and produce it as output. It can also determine the bit that should be carried over to position 1. Now the machine adds a_1, b_1 and this carry. This determines both the 1st bit of the output and a new carry. The machine repeats this for every position in the input numbers and also produces as output the last carry.

Two details need to be taken care of. First, the machine needs to keep track of the current position in a and the current position in b. This can be done by maintaining two pointers, an idea that was mentioned in the first lecture. A third pointer, initially set to 1, can keep track of the current position of the input read head. By comparing this pointer to the other two, the machine can easily reposition its input read head over the positions being currently considered in the input numbers.

The second detail concerns the initialization of the first two pointers. To do this correctly, the machine needs to figure out where a ends and where b begins. This can be done by first scanning the input to determine its total length, verifying that this length is even and then dividing it by two. Note that dividing an even number by 2 corresponds to simply shifting its binary representation one bit to the right.

Since all the space requirements of this machine are for a carry, three pointers and a few counters, we conclude that computing addition, with the output in reverse order, is in L.

To produce the output in the correct order, we design a second machine. One thing the second machine could do is simulate the first machine so it can store its output and then reverse it. Such a simulation can be done by incorporating the program and internal states of the first machine into the program and internal states of the second machine. During the simulation, no output is produced. Instead, the second machine simply stores the output the first machine would have wanted to produce. This output can then be reversed and produced. However, this new machine now uses a linear amount of space. We can still manage to use only logarithmic space,

but we need to proceed differently.

The second machine will start by simulating the first one to figure out what the nth bit of the sum is. Once again, the first machine is not allowed to produce output directly. When the first machine tries to produce the nth bit of the sum as output, the simulation ends and the second machine produces that bit as output. The simulation is then repeated for all the other bits of the sum, one after the other, in the right order. Note that the second machine will require enough space to store the contents of the first machine's memory plus a counter or two to control the simulation. Since the first machine used a logarithmic amount of space, so will the second machine. Thus addition is in L.

It is interesting to note that what is happening here is that the second machine is using the first one to solve the following problem: given two n-bit binary numbers and a position number i, determine the ith bit of the sum of the two numbers. Since the output here consists of a single bit, this problem can be viewed as a decision problem. In fact, it can be argued that in some sense, this decision problem is equivalent to the problem of computing the original function.

We could now use an approach similar to the above sequential algorithm to construct a circuit that computes addition. This circuit would have $O(n)$ depth. Instead, we will now obtain a circuit of constant depth.

This will require a different idea called the *carry look-ahead method*. Say that position i *generates* a carry if a_i and b_i are both 1. Say that position i *propagates* carries if at least one of a_i or b_i is 1. There will be a carry coming into position $i+1$ if and only if (1) position i generates a carry or (2) position $i-1$ generates a carry and position i propagates it, or (3) position $i-2$ generates a carry and positions $i-1$ and i propagate it, or ... More symbolically, let $g_i = a_i \wedge b_i$, $p_i = a_i \vee b_i$ and let c_i denote the carry coming into position i. Then, for $i = 1, \ldots, n$,

$$c_i = \bigvee_{j=0}^{i-1} g_j \wedge p_{j+1} \wedge \cdots \wedge p_{i-1}.$$

The ith bit of the output can now be easily computed from a_i, b_i and c_i. This gives an AC^0 circuit for addition.

3. Multiplication

Given two n-bit numbers, the *multiplication* problem is to determine their $2n$-bit product. Once again, we are computing a function. We will use the same input encoding as for addition.

As for addition, we first describe a machine that produces the output in reverse order. A second machine that uses only a logarithmic amount of additional space can be constructed using the same technique as for addition.

Suppose that the two input numbers a and b have binary representations $a_{n-1} \cdots a_0$ and $b_{n-1} \cdots b_0$. Following the usual multiplication algorithm, the product of a and b can be written as $\sum_{i=0}^{n-1} ab_i 2^i$. Each of the n terms in this sum is easy to compute. Adding them can be done as follows. Add all their position-0 bits. This can be done by updating a simple counter. In the end, the position-0 bit of this counter is the position-0 bit of the sum. Produce that bit as output, subtract it from the counter and divide the counter by 2. Now add the position-1 bits to the counter. The position-0 bit of the counter is now the position-1 bit of the sum. Once again, produce that bit as output, subtract it from the counter and divide the counter by 2. Repeat this for all the positions. Then output the remaining bits of the counter. The key thing to realize is that the value of the counter will never exceed $2n$, so that its size will always be $O(\log n)$.

The above strategy gives a polynomial-time algorithm that uses $O(n^2)$ space. This large amount of space is what is needed to store the n numbers $ab_i 2^i$. We can do better in terms of space by not storing these numbers explicitly but instead computing their bits as needed. The amount of space used then decreases dramatically to only $O(\log n)$. Thus multiplication, with the output in either order, is in L.

A circuit for multiplication can be constructed by using the same divide-and-conquer strategy that was used for parity. First, compute the n numbers $ba_i 2^i$, for $i = 0, \ldots, n-1$. Then add these numbers two at a time using a binary tree of additions. Since each of these additions can be performed by an AC^0 circuit, we get an AC^1 circuit for multiplication. Note however that an NC^1 multiplication circuit can be constructed. This will be done in the advanced course.

4. Graph Reachability

Given a graph G and two nodes s and t, the *reachability* problem is to determine whether there is a path from s to t in G. We assume that the graph is directed and that it is given in some reasonable way, such as by an adjacency matrix or by an adjacency list.

We present here a polynomial-time algorithm for reachability. The algorithm works by marking nodes that can be reached from s. Initially, only s is marked. Then, the algorithm performs a series of passes through all the nodes in G. If it finds an unmarked node that can be reached from a marked node, it marks that node. This

process stops when an entire pass executes without marking a single new node. At that point, the marked nodes are precisely those that can be reached from s. It is now just a matter of verifying that t has been marked. The algorithm runs in polynomial time because the number of passes performed will always be at most the number of nodes in G.

The reader may have noticed that in describing the previous algorithm, we did not refer to a machine. Since our definition of an algorithm is in terms of a machine (one that eventually halts on every input), we were still talking, at least implicitly, about a machine. What has changed, however, is the amount and level of detail we provided. In particular, we no longer talked about pointers to the input or counters. And we certainly did not specify the precise movement of the memory heads. This higher level of description is convenient; it allows us to make abstraction of details such as the exact nature of the encoding of the input. It also stresses an important point: the fact that the reachability problem, for example, has a polynomial-time algorithm does not depend on our particular definition of an algorithm or on the particular details of the underlying model of computation. From now on, in describing algorithms, we will mention lower-level details only when necessary to establish the complexity bounds we are interested in.

It is important to realize, however, that if we wanted, we could implement any of our high-level algorithms by providing a more detailed description of a machine (with counters and head movement) or even a full-detail description including a program with all of its instructions. In fact, it may be a good exercise at this point to fully describe a machine that decides parity or one that has a counter in memory and increments it by one.

5. The Circuit Value Problem

Given a circuit C and an input w, the *circuit value* problem is to determine the output of C on w. A circuit is a labeled, directed graph so we can use any reasonable graph encoding. The node labels are simply either input bit numbers or the words AND, OR or NOT, together with possibly the mention "output".

Circuits can be evaluated in polynomial time by an algorithm similar to the reachability algorithm. The circuit evaluation algorithm figures out and records the value of all the gates in the circuit. Initially, only the value of the input gates (and any other gate with no input wires) is known. The algorithm then performs a series of passes through all the gates in C evaluating all the gates it can, i.e., all those whose input values are already known. Eventually, the output gate of the circuit will be

evaluated. The algorithm runs in polynomial time since every pass will evaluate at least one new gate so that the number of passes will be no greater than the number of gates in the circuit.

What about a circuit for the circuit value problem? Note that we cannot simply say that we will use the given circuit to figure out what its value is. We need a *single* circuit that can evaluate any given circuit. Later we will show that the circuit value problem, and in fact every problem in P, can be computed by polynomial-size circuits.

6. 3-Colorability

A graph is 3-colorable if we can assign one of three colors to each of its nodes in such a way that no two adjacent nodes get the same color. The *3-colorability* problem is to determine whether a given graph is 3-colorable.

This problem can be solved by the *exhaustive search* method. The algorithm considers, one after the other, all the possible assignments of 3 colors to the nodes of the graph. Each assignment is examined to see whether it constitutes a 3-coloring of the graph. If a 3-coloring is found, the graph is accepted. If no 3-coloring is found, the graph is rejected. Since the number of 3-colorings is exponential in the number of nodes in the graph, this algorithm will not run in polynomial time. However, it will use only a polynomial amount of space, so we conclude that 3-colorability is in PSPACE.

7. Exercises

1. Show that if a language can be decided by a circuit family of logarithmic depth and fan-in 2, then it is in NC^1.

2. Encode pairs of binary numbers by interleaving the bits in their binary representations in *reverse order*. In other words, if $a_{n-1} \cdots a_0$ and $b_{n-1} \cdots b_0$ are the binary representations of a and b, then the pair $\langle a, b \rangle$ is encoded as $a_0 b_0 a_1 b_1 \cdots a_{n-1} b_{n-1}$. Show that under this encoding, the addition problem (with the output produced in reverse order) can be solved by a DFA and thus is in DSPACE(1).

3. Show that the following problem can be solved by a DFA: given a binary number, determine whether it is divisble by 3.

4. Given two n-bit numbers, the *comparison* problem is to determine if the first number is greater than the second one. Under the encoding described in the text, show that this problem can be solved in L.

5. Suppose that two functions can be computed by machines using a logarithmic amount of space. Show that the composition of these two functions can also be computed in L. (Thus the class of functions in L is closed under composition.)

6. Complete the details in the construction of the AC^0 addition circuit. In particular, show how to compute the ith bit of the output from a_i, b_i and c_i and show that the circuit has size $O(n^2)$.

7. Show that addition can be computed by circuits of linear size.

8. Show that any language decided by a DFA can be decided by circuits of linear size.

9. Show that the reachability algorithm will mark every node that can be reached from s.

10. Show that the circuit evaluation algorithm will evaluate every gate in the circuit.