6.966 Algebra and Computation	November, 16 1998
Lecture 19	
Lecturer: Madhu Sudan	*Scribe: Adam Klivans*

1 Introduction

In this lecture we will introduce several algebraic models of computation including algebraic circuits, decision trees, branching programs and straight line programs. We will focus on three central questions:

1 How can we transfer known results in the boolean world to the algebraic setting?

2 How are lower bound techniques applicable to the algebraic case?

3 What is the relationship of algebraic complexity classes to their boolean counterparts?

Furthermore, we will prove a simple result due to Ben-Or and Cleve showing how to simulate a circuit of depth d with a straightline program of size 4^d using only 3 registers.

1.1 Preliminaries

We will consider non-uniform models of algebraic computation over some ring R. The model of computation computes a function $\phi\colon R^n \to R^m$. I.e. there will be n inputs and m outputs.

2 Algebraic Circuits

Definition 1 *An algebraic circuit is a directed acyclic graph. Each node, or gate, computes some ring operation (i.e., $+, -, *$) of the incoming edges. The output of each node "contains" the value of the ring operation applied to the values in the incoming edges.*

In the case where the ring where are working over is a field and we allow the division operation, our circuits correspond precisely to the set of all rational functions over the field.

In the boolean case, these circuits compute all boolean functions (assuming we allow an input to be hardwired to 1 for negations). This is not so in the general algebraic case. For example, we have no equality or inequality operator. We could allow equality gates, i.e., a gate that outputs 1 if and only if its input is equal to 0. Or, if we are working over an ordered field, we could add inequality gates, i.e., a gate that outputs 1 if and only if its input is greater than or equal to 0.

3 Algebraic Decision Trees

Another frequently studied model of computation is the algebraic decision tree:

Definition 2 *An algebraic decision tree is a binary tree with two types of nodes: query nodes and leaf nodes. Each query node is labelled with a polynomial in the input variables $p(x_1, \ldots, x_n)$. We "branch" depending upon whether or not $p(x_1, \ldots, x_n) = 0$. Each leaf node is then labelled as reject or accept. One of the nodes is designated as the start node. The decision tree accepts (rejects) an input if the corresponding path in the tree leads to an accept (reject) leaf.*

One way of measuring the complexity of a decision tree is to look at the maximum over all paths from root to leaf of the sum of the degrees of the polynomials at each node. To measure a function's algebraic decision tree complexity, we then take the minimum of this quantity over all decision trees computing the function. Notice that if each polynomial is a linear polynomial then this measure of complexity coincides with the standard notion of boolean decision tree complexity.

4 Straight Line Programs

Another interesting model of algebraic computation is the straight line program:

Definition 3 *A straight line program is a sequence of instructions using the variables (registers) $x_1, \ldots, x_n$, $A_1, \ldots, A_l, O_1, \ldots, O_m$ Each x_i corresponds to an input variable and each O_i corresponds to an output.*

Each instruction must be one of the following:

1 $A_i = A_j$ **op** A_k where **op** $\in \{+, -, *\}$

2 $A_i = A_j$ **op** x_k where **op** $\in \{+, -, *\}$

3 $A_i = r$ where r is some ring element.

4 $O_i = A_j$ for some i and j.

The **time** complexity of a straight line program corresponds to the number of instructions and the **space** complexity corresponds to the number of registers used.

Before we begin looking at straight line programs, we introduce another model of computation: the branching program.

5 Branching Programs

Definition 4 *A branching program is a directed acyclic graph where all nodes are labelled by variables, except for the output nodes which are labelled either* 1 *or* 0*. The nodes labelled by variables are called query nodes. Each query node has a label x_i and has two outgoing edges, one labelled* 1 *and one labelled* 0 *(corresponding to whether or not x_i equals* 0 *or* 1*).*

5.1 Branching Programs and Straight Line Programs

Here is a simple example which relates straight line programs and branching programs:

Example: Assume that we are given a straight line program over $GF(2)$ using registers $A_1, \ldots, A_\ell$ and having d instructions. Notice that there are exactly 2^ℓ different "configurations" that the registers could be in. We can simulate the straight line program by a branching program divided into d "levels." Each level corresponds to one instruction in our straight line program. Level i contains 2^ℓ nodes, each one corresponding to possible configuration of our registers. We connect node k in level i to node j in level $i+1$ if the ith instruction in our straight line program would change the configuration of our registers in node k to the configuration represented by node j given that the input in the ith instruction equals 1 or 0 (if the input is not looked at in the ith instruction then both edges will point to the same node).

Definition 5 *Let B be a branching program. We say B is levelled if it can be partitioned into sets of nodes $S_1, \ldots, S_n$ such that all of the edges leaving S_i enter only the nodes in S_{i+1}.*

Definition 6 *We say a levelled branching program has width w if the size of the largest set in the partition is less than or equal to w.*

Proposition 7 *¿From the above example we see that any straight line program using ℓ registers and having d instructions can be simulated with a levelled branching program of width 2^ℓ.*

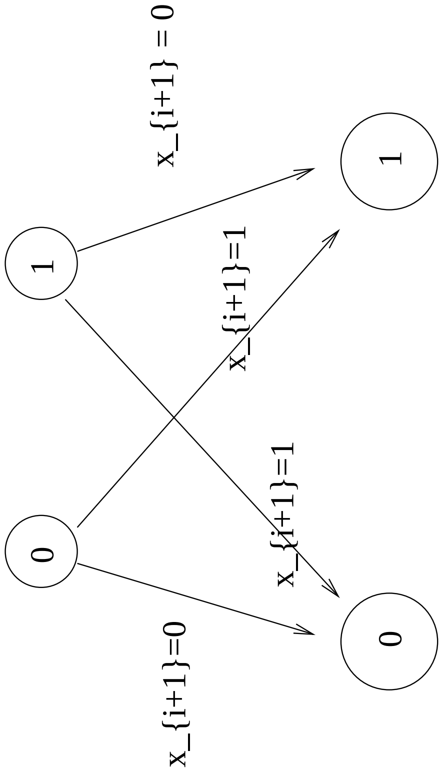

Figure 1: The ith level of a branching program computing Parity

We can use levelled branching programs to give an interesting non-uniform measure of the complexity of a function by looking at the smallest width over all levelled branching programs computing a function. For example, the parity function has a width 2 branching program:

Considerable effort was put into understanding what functions cannot be computed with small width, polynomial size levelled branching programs. **Put reference here** proved that the majority function on n variables requires exponential size width-3 branching programs. Finally Barrington proved that width-5 branching programs correspond precisely to non-uniform NC_1.

6 Algebraic Circuits and Straight Line Programs

In this section we will prove the following result of Ben-Or and Cleve:

Theorem 8 *If a function f is computable by an algebraic circuit of depth d with the $+, -, *$ operations then f is also computed by a straight line program of size 4^d with 3 registers*

Proof: We will assume that every instruction in our straight-line program will have the form

$$A_i \Leftarrow A_j + (A_k * x_l)$$

Let $f(x_1, \ldots, x_n)$ be some mapping from R^n to R that can be computed by a circuit of depth d. Assume for a moment that using registers A_1, A_2 and A_3 we can compute (stored in our output registers) $A_1 + f(x_1, \ldots, x_n)A_2, A_2, A_3$. Then with $A_1 = 0, A_2 = 0$, and $A_3 = 0$ given as input we would have as output $f, 1, 1$. We will prove that we can achieve this construction by induction on the depth of the circuit computing the function.

- **Base Case:** This can clearly be done for a depth 0 circuit.
- **Inductive Case:**

Assume $f = f_1 + f_2$. Then by induction we have a straight line program with three variables A_1, A_2, A_3 such that after our straight line program executes, these registers take on the values $A_1 + f_1A_2, A_2, A_3$. Then we use these values as inputs to our straight line program for f_2 and we receive as output $A_1 + f_1A_2 + f_1A_2 = A_1 + (f_1 + f_2)A_2, A_2, A_3$.

So far we have not used the full power of our inductive hypothesis. The case for multiplication will be a little more involved:

Assume $f = f_1 \times f_2$.

1 Use, via our inductive hypothesis, a straight line program which on input A_1, A_2, A_3 produces as output $A_1, A_2, A_3 + f_1A_2$.

2 Now feed these registers into our straight line program for f_2 to obtain as output $A_1 + f_2(A_3 + f_1A_2), A_2, A_3 + f_1A_2$.

3 Use the output from (2) as input to our SLP for $-f_1$ to get as output $A_1 + f_2(A_3 + f_1A_2), A_2, A_3 + f_1f_2 - f_1f_2 = A_3$.

4 Finally use the output from (3) as input to our SLP for $-f_2$ to get as output $A_1 + f_1f_2A_2, A_2, A_3$.

7 Uniform Models for Algebraic Computation

In the next lecture we will introduce the Blum-Shub-Smale model of algebraic computation and look at the corresponding complexity classes.