ITCS:CCT09 : Computational Complexity Theory	Apr 11, 2009
Lecture 9	
Lecturer: Jayalal Sarma	*Scribe: Bangsheng Tang*

In this lecture, three different non-uniform models of computation: *boolean formulas*, *decision tree* and *branching program* are introduced, and their relation with other time or space classes are discussed in details.

Recall that in the non-uniform models, we allow various combinatorial objects to abstract out the computation. That is an instance of the computational device(or combinatorial object) is allowed for for computations on all inputs of a particular length. Hence the model is really an infinite family of such objects. This note is divided into three sections, each section deals with one of these models.

1 Boolean Formula

The Boolean formula model restricts the reusability of a wire in a circuit which is a natural constraint based on the out-degree of each node in the underlying graph.

Definition 1 (Boolean Formula) *A Boolean formula is a Boolean circuit with the restriction that every gate can have at most one gate which it is feeding to. That is the underlying DAG is an (un)directed tree rooted at the root gate.*

The parameters for these restricted class of circuits is again the same as the case of general circuits : depth and size. A formula can be presented as a tree with inner nodes representing operators and leaves representing variables. Depth is maximal depth of leaves, and size is the number of inner nodes.

We can define the following complexity class, straight away.

Definition 2 (BF) *A language L is in* BF *if and only if it can be computed by* poly *size Boolean Formulae.*

As we said formulae are restricted Boolean circuits. That is, formulae does not allow the output of a gate to be used more than once. Thus, a natural way to convert a circuit into a formulae is: whenever the fanout of a gate g is k which is more than 1, replicate the subtree computing g, $k-1$ times to feed in to each outgoing wire from g.

This clearly blows up the size. by how much? It could be exponential. When will it remain within polynomial bound? Well, if fan-in of each gate of the circuit is bounded the depth is only log, the blow up is still poly. This gives a natural lower bound for BF in terms of classes we have seen before. In fact, more is true, as we show below.

Theorem 3 $\mathsf{BF} = \mathsf{NC}^1$.

Proof The direction, $\mathsf{NC}^1 \subseteq \mathsf{BF}$ follows from the above arguemnt. Now we need to prove the other direction. That is $\mathsf{BF} \subseteq \mathsf{NC}^1$. A formula by definition is a circuit. But the circuit might have depth of order $\Omega(n)$, e.g. $(x_1 \wedge (x_2 \wedge \cdots (x_{n-1} \wedge x_n)) \cdots)$. Thus, we need to derive an equivalent circuit with lower depth. This is called *depth reduction problem* in general. Almost always, the idea will be to do a rebalancing of the circuit structure, in order to eliminate all long paths.

In this particular case, the idea is to cut the circuit at a certain edge (which we will soon describe how to choose), and get two subcircuits. Denote the part with the output node as B, and the other part M. Clearly B has a cut wire (y, see the figure), and we have to feed in some value to it. Let B^0 be the circuit by replacing y with 0 in B, and B^1 by replacing y with 1 in B.

Without loss of generality, assume that the variables that are used by B are $x_1, \cdots, x_k$, the value output by M is denoted as y.

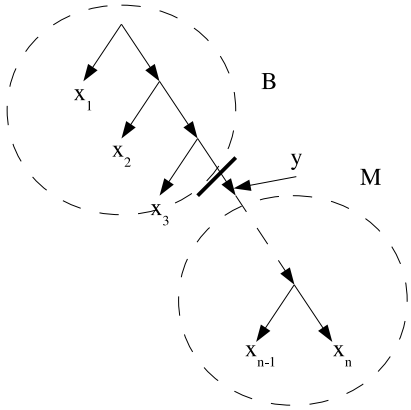

Figure 1: Transformation

And then, construct the new circuit $C(x)$ as follows:

$$C(x) = B^0 \text{ if } M(x_{k+1}, \cdots, x_n) = 0$$
$$C(x) = B^1 \text{ if } M(x_{k+1}, \cdots, x_n) = 1$$

The construction of C by selecting between B^0 and B^1 by M can be done by a constant number of gates (implementing the selector function).

The depth of the new circuit is the max of the depth of B and M. Thus, if one can find a good cut point, such that the depth of the new circuit can be reduced by a constant fraction, then after applying this many times transformation, we are guaranteed to get a circuit with polynomial size and logarithmic depth.

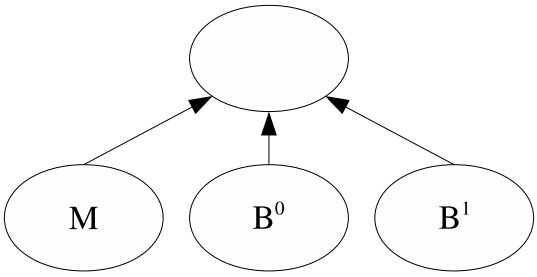

Figure 2: A depth reduced equivalent circuit

■

But now how do we find the good cutting point? In fact, we can show that it exists. This will show that there exists and NC^1 circuit. We will not argue the uniformity contraints (about how to find this NC^1 circuit) in this lecture. In particular, we will show the following lemma.

Lemma 4 *For any rooted binary tree T of size s, there is a node r such that the tree T' rooted at r has size in the range $[\frac{s}{3}, \frac{2s}{3}]$.*

Proof For a node v denote by T_v the size of the tree rooted at v. We start traversing down in search of r starting from the root node. Consider the two child nodes a and b of the root. Choose the heavier child (say a). That is $|T_a| \geq |T_b|$. Clearly has $|T_a| \geq \frac{s}{3}$. Consider the two children of a and traverse down choosing the heavier child until you hit the first node r such that both its children have subtrees of size at most $\frac{s}{3} - 1$. Clearly, r is the required node that is claimed in the lemma. ■

2 Decision Tree

Decision tree is a model ensembles when a person making a decision, every time he sees a condition, he adjusts his strategy a little bit until have seen all the conditions and finally

make a decision. Notice that this is adaptive, we could choose to query different inputs based on different settings of the previous inputs that we have seen.

2.1 Definitions

A formal definition of a decision tree is as follows,

Definition 5 (Decision Tree) *A decision tree for a function* $f : \{0,1\}^n \to \{0,1\}$ *is a tree for which each node is labelled with an index* i *of input string, and has two outgoing edges, labelled* 0 *and* 1*. Each tree leaf is labelled with an output value* 0 *or* 1*.*

The computation on input $x = x_1 x_2 \cdots x_n$ proceeds at each node by inspecting the input bit x_i indicated by the node's label. If $x_i = 0$, then continues with the 0-edge, and with 1-edge otherwise, until a leaf is reached, and the value of the leaf is returned.

The natural parameters that can be associated with the decision tree are, again, depth(which we will call height) and size denoted by D_h and D_s. The definitions of depth and size are in the usual sense. Now we give some examples of a decision tree.

Remark There is a close connection betweek decision trees and DNFs (disjunctive Normal Forms, or $\wedge$ or $\vee$s.). From a decision tree, we can directly write DNF based on each path that makes the function 1. The term size of the DNF (the number of variables in each $\wedge$) will be exactly the depth of the decision tree. Similarly, from a DNF representation we can directly write a decision tree. We will see this in more detail in the coming lecture.

A first example is the $\wedge$ function. Clearly, the decision tree will have to look at entire set of variables to decide the output. The same is the case with $\vee$.

A little more complicate example might be the parity function,

$$\text{PARITY}_n(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if the number of 1s in the input is odd} \\ 0 & \text{otherwise} \end{cases}$$

Here is a decision tree for PARITY function. At the root, it decides based on input x_1 whether to remain 0 or 1. At the i^{rmth} level of the tree, the decision tree examines the variable x_i and decides to remain 0 or 1 based on what it has seen. Finally, when it has seen all the variables, it outputs the current decision. This decision tree clearly has depth n, and size 2^n. But for parity function, these are essnentially optimal (this is crucial fact that will be made use of, when we see circuit lower bounds for PARITY fuction in the upcoming lectures). We show this below.

Theorem 6

$$D_h(\text{PARITY}_n) = n$$

$$D_s(\text{PARITY}_n) = 2^n$$

Proof Clearly every variable has to be inspected while computing parity. This gives a lower bound of n for $D_h(\text{PARITY}_n)$. We will now show a slightly different way of arguing the same and it provides a combinatorial characterisation on the depth.

We can also think of decision trees as partitioning the Boolean cube (the cube where each corner is indexed by an n-bit string, hence 2^n of them.). To see this: color the Boolean n-cube with two colors, black and white (depending on if the function is taking a value 0 and 1). Now, the decision tree, on each path inspects a set of variables, and makes a decision based on them. The settings of variables that were not inspected forms a subcube of the n cube, and the value remains same on this subcube.

Clearly, for the the parity function, the boolean cube is uniformly bichromatic. That is any node of the n-cube has all its neighbours colored with a color different from it. Thus there cannot be a monochromatic subcube of size more than 1. Hence the claim.

Every variable inspected does give two non-trivial possibilities because of the above argument, and this gives rise to the size lower bound. ■

3 Branching Program

In this section, we will characterize the log space classes using computational model named branching programs.

3.1 Definitions

Branching program(BP) can be seen as a generalization of decision tree model, in the sense that the decision process need not to be a tree-like, it is allowed to be a DAG.

However, we will state in the following, an equivalent definition, which is more convenient to work with for what we are going to do in the rest of this lecture.

Definition 7 (Branching Programs) *A branching program is a layered DAG such that each node has two outgoing edges, of which one is labelled x_i where i is an index into the input string, and the other as $\overline{x_i}$.*

Notice first that instead of having two outgoing edges with labels 0 and 1 for a node which looks at the input bit x_i we could just have those two edges labelled with the variable and

its negation itself. This is obviously equivalent to the model where we have nodes labelled by input variables.

If we consider one particular input setting for the input x, this will give an instantiated sugraph(which is a DAG) of G, where the edges which are labelled x_i(resp. $\neg x_i$) are retained if and only if $x_i = 1$ (resp. $x_i = 0$).

Similarly, some parameters are of interest. The *depth* of a branching program is defined as the longest path from one input node to a output node; the *size* is defined as the number of nodes in it; the *width* is the maximal number of vertices in each layer(if the program is layered, that is, transitions can only be between layers).

Branching programs have more computational power than decision trees, since PARITY can be computed by a width 2 poly size branching program, in contrast to an exponential lower bound of a decision tree mentioned in previous section.

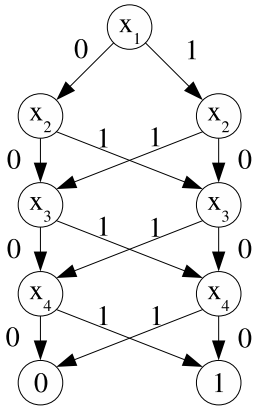

Figure 3: A branching program computing parity with width 2

Now we define various versions of the model. If nondeterminism is allowed when branching (that is more than one outgoing edge with the same label), then the branching program is called Non-deterministic Branching Program(NBP). If the width of graph is a constant(independent of n), then it is called a Bounded Width Branching Program(BWBP). We call a branching program a *permutation branching program*(PBP) if the functions associated with each layer are permutations. To be more specific, the computation at any layer can be represented by two boolean functions $f_0, f_1 : \{1, 2, \cdots, w\} \to \{1, 2, \cdots, w\}$, where $f_0(i) = j$ if and only if the edge labeled 0 from the ith node in layer l goes to jth node in the next layer, similar for f_1. In PBP, such f_0 and f_1 are permutations. Figure 3 shows a PBP. Another point to note is that permutation branching programs are deterministic by definition.

3.2 Reachability

Before going to more details about branching programs, let's first state some results about log space classes. Some of them you must have seen while solving problem set 1.

Given a directed graph G and two nodes (s, t), the problem of deciding whether there is a path from s to t is called REACHABILITY PROBLEM. There are many varieties of reachability problem. These problems are important since they characterize the classical log space classes. In particular, the following two theorems hold.

Theorem 8 REACHABILITY *of* (G, s, t) *is* NL*-complete*

Reachability can be seen to be in NL, because if we start at s and guess a path using nondeterminism, only the current node is needed to be stored. To see the hardness, given an NL machine, one can construct the state graph of the machine, and check whether the initial state is reachable to the accept state. We have roughly seen the details of this construction in one of the previous lectures, but never formalised it. We will leave the details of this formalisation as an exercise.

If the graph G does not have cycles, then the problem is much easier and in fact characterises L. The hardness follows from above construction itself (Argue that the graph we obtain is a forest !). And why can we solve this problem in log space (Thats another exercise !). But we state,

Theorem 9 *If the graph* G *is an undirected forest,* UNDIRECTED-FOREST-REACHABILITY *of* (G, s, t) *is* L*-complete*

3.3 Characterizing Log Space and Circuit Classes

We will see an easy proposition.

Proposition 10 NBP=NL

To see the above claim, it is clear that NBP is contained in NL because the model involves reachability testing in special kind of graphs (layered) to decide whether an input is accepted or not. Notice that the graph here is the instantiated graph G where the edges are present if the corresponding input bit or its negation is 1. Now to show the hardness we will again refer to the details of the NL-hardness of the reachability problem in directed graphs and notice that the output graph can easily be made layered.

Now we will state a result about permutation branching programs.

Proposition 11 PBP=L

Notice that reachability testing is simply multiplication of the bipartite adjacency matrices of each layer of the branching program. Now for a permutation branching program, these matrices are simply permutation matrices. Now we use the fact that multiplying these can be done in in log space. We skip the details again.

Theorem 12 (Barrington) BWBP=NC^1

Proof

($\subseteq$) Given a BWBPP computing function f, we will construct an NC^1 circuit C simulating it. We use notation of computing in a stronger sense, that is, P accepts x, if $P(x)$ is a certain function from $[w]$ to $[w]$(not necessarily a permutation). Such a function can be represented by w^2 boolean functions telling whether $f(i) = j$ for each i, j. Two of such functions can be computed by a fixed circuit with size depends only on w. Then we build C by constructing a binary tree of composition circuits.

($\supseteq$) We shall prove that, any circuit in NC^1 can be simulated by a bounded-width branching program of width 5.

We make some more observations on permutation branching programs with width w. Layer i is a permutation decided by ith bit of input x, denoted as (x_i, σ_i, π_i), if $x_i = 0$ the permutation is σ_i, and π_i otherwise. The whole program can be seen as the product of all the permutations and thus is a permutation decided by the input.

A program P δ-computes a function f, where δ is a non-identity permutation of $\{1, 2, \cdots, w\}$, if

$$f(x) = 0 \Leftrightarrow \Pi(x_i, \sigma_i, \pi_i) = \delta$$
$$f(x) = 1 \Leftrightarrow \Pi(x_i, \sigma_i, \pi_i) = e(identity)$$

This definition of program can be shown equivalent to what we defined as permutation branching programs. In the following constructions, we will take this as guaranteed.

Fix a cyclic permutation δ over $\{1, 2, 3, 4, 5\}$. Given a depth d circuit, we show how to construct a width 5 branching program of length 2^{2d} that δ-computes the same function as the circuit. The necessity of cyclic permutation is that, for two cyclic permutations α and β, there exists a permutation δ such that $\alpha = \delta^{-1}\beta\delta$(since $\delta^{-1}(\beta_1, \cdots, \beta_n)\delta = (\delta(\beta_1), \cdots, \delta(\beta_n))$). If we have a program P β computes f, then by replacing first permutation (x_1, σ_1, π_1) by $(x_1, \delta^{-1}\sigma_1, \pi_1)$ and last permutation (x_l, σ_l, π_l) by $(x_l, \sigma_l\delta, \pi_l\delta)$, one will get a program P' α computes f.

The proof is by induction on d. When $d = 1$, the circuit outputs one of its input bit, say, x_i, then one may construct a two-layer program, (x_i, δ, e).

For induction step, assume that for all depth $< d$ the claim is true, and we will prove for depth d. It suffices to see how to simulate OR and NOT gates(AND gates can be simulated by OR and NOT gates).

For a NOT gate, suppose a circuit C with depth d has a NOT gate on the top, by induction, the lower $d-1$ depth subcircuit can be simulated by a program P of length $2^{2(d-1)}$. Suppose P δ-computes language L, we construct a P' δ^{-1}-computes language $\bar{L}$, and therefore satisfies the conditions. Suppose last layer of P can be represented as (x_l, σ_l, π_l), P' is constructed by replace the last layer with $(x_l, \sigma_l\delta^{-1}, \pi_l\delta^{-1})$. One may check that, on a fixed input, if P outputs δ P' outputs e, and if P outputs e, P' outputs δ^{-1}. This is exactly what we need. Since δ and δ^{-1} are both cyclic permutations, we can further modify P' to a program δ-computes $\bar{L}$.

For a OR gate, consider the elements $\alpha = (15324)$, $\beta = (12534)$, and $\gamma = (12345)$. $\gamma = \alpha\beta\alpha^{-1}\beta^{-1} \neq e$. By induction, assume that we have width 5 branching programs P_1 α-computes some function f, and P_2 β-computes some function g. Just as in the previous paragraph, we can construct programs P_1', P_2' compute $\bar{f}$ and $\bar{g}$ respectively. Then compute the four programs in order, $P = P_1P_2P_1'P_2'$. It can be proved that the constructed program γ-computes $f \wedge g$, and can be further modified to δ-computes $f \wedge g$. The length of the four programs is $\leq 2^{2(d-1)} \times 4 = 2^{2d}$. This completes the inductions.

Therefore, for each NC^1 circuit, one can find a width 5 program δ-computes it.

■