IITM-CS6845: Theory Toolkit 16 April, 2012

Lecture 54 : Layered Branching Programs

Lecturer: Jayalal Sarma M.N. *Scribe: Prashant Vasudevan*

THEME: Layered Branching Programs
LECTURE PLAN:Build up to Barrington's Theorem

Previously we defined *Branching Programs* that are built on Directed Acyclic Graphs and work by starting at a source vertex and testing the values of the variables that each vertex is labeled with and following the appropriate edge till a sink is reached, and accepting or rejecting based on the identity of the sink. Note that below we speak only of branching programs of size polynomial in the number of variables.

Formally, the language computed by a branching program P is given by:
$L(P) = \{x \mid P|_x \text{ has a path from } s \text{ to } t \in T\}$, where T is the set of accepting sinks.

Now we consider a constrained version of Branching Programs:

Definition 1. *Layered Branching Programs* are branching programs where the nodes are partitioned into a number of *layers*, and edges go only from nodes in one layer to nodes in the next. The start node is in the first layer and the sink nodes in the last.

Definition 2. The *length* of a Branching Program is the length of the longest path from the start node to any of the sinks. In case of layered programs, it is also the number of layers, as the start node is in the first layer and the sinks are in the last.

Definition 3. The *width* of a Layered Branching Program is the maximum number of nodes in any of its layers.

The width of a layered branching program corresponds in a rough sense to the amount of memory the program carries - this is the information that one layer can receive from the previous one and pass on to the next.

We had seen earlier that branching programs in general are NL-Complete. But what about layered branching programs? These are no lesser as one can take any branching program, perform a topological sort and get an equivalent layered branching program by adding a few extra vertices to satisfy the requirement of edges to go only between adjacent layers - all without affecting the length in any way and keeping the increase in number of nodes polynomial.

Next we wonder about layered branching programs in which the width - the 'memory' - is constrained. Constrained to how much? We start at the very bottom, restricting our

programs to constant width. At first glance this model seems quite weak and so did it seem to most till Barrington came along and proved it to be quite otherwise.

We start with noting that the branching program for $PARITY$ seen earlier is layered, with length n and width 2. This shows a function in $\mathsf{NC}^1 \backslash \mathsf{AC}^0$ that can be computed by a constant width branching program. Are there other such functions? Before going there, we observe the following - a claim in somewhat the other direction:

Claim 4. *All languages computed by constant width layered branching programs are in* NC^1.

Proof. To see this, notice that, once the input (values of variables) has been decided, computing the program reduces to finding reachability in the DAG that results from retaining the edges corresponding to the input, from the start vertex s to one (w.l.o.g) other sink vertex t. Let (u, v) denote the boolean function corresponding to reachability from u to v in this graph. Let M be the layer that is exactly half-way between the first (which has s) and the last (which has t). We have:

$$(s, t) = \bigvee_{v \in M} ((s, v) \wedge (v, t))$$

For (s, v) and (v, t), we proceed recursively in the same manner until we get to functions involving vertices in adjacent layers that we may evaluate directly. As length of the program is at most polynomial in the number of variables n, the depth of this recursion is bounded by $O(\log n)$, each level of recursion introducing two levels of gates into the circuit. Thus the depth of the circuit is $O(\log n)$ and as the width of the program is constant, fan-in of the gates is upper-bounded by a constant, making this an NC^1 circuit computing the same function as the branching program. □

Notice that programs very similar to the one for $PARITY$, but of width k, may be used to compute MOD_k for any constant k. Also, we can decide the union and intersection (OR and AND) of the languages of any two constant width programs in another constant width program by concatenating the programs appropriately. This suggests computing power comparable to ACC^0. How much more powerful could these programs be? This question was settled by the following theorem from David Barrington's Ph.D. thesis.

Theorem 5 (Barrington's Theorem)**.** *All languages in* NC^1 *can be computed by polynomial-sized layered branching programs of constant width.*

They can, in fact, be computed by a special class of branching programs, called permutation programs, of width 5. Before getting to the proof itself, we list a few preliminaries.

Definition 6. A *k-Permutation Branching Program* is a layered branching program in which:

1. Each layer has k nodes.
2. Each layer observes only one variable.
3. For any setting of the variables, the edges going between any pair of consecutive layers form a permutation of the vertices.

Definition 7. A program over a group G is given by:

$$P = ((x_1, a_1, b_1), (x_2, a_2, b_2), \ldots, (x_n, a_n, b_n)),\ a_i, b_i \in G$$

On input $x = (x_1, x_2, \ldots, x_n)$,

$$P(x) = \prod_i \alpha_i,\ \alpha_i = \begin{cases} a_i,\ if\ x_i = 0 \\ b_i,\ if\ x_i = 1 \end{cases}$$

We shall be considering programs over the symmetric group S_k for some k. These programs may be realised by k-permutation branching programs without the acceptance conditions. We instead use the notion of σ-acceptance.

Definition 8. A Program P over S_k is said to σ-accept a language A if:

$$P(x) = \begin{cases} \sigma,\ if\ x \in A \\ I,\ if\ x \notin A \end{cases}$$

Now for something about permutations. We shall be using two notations to represent permutations:

1. *Pointwise*: $\sigma = ((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n))$ implies that $\sigma(x_i) = y_i$.
2. *Cyclic*: $\sigma = ((x_{11}, x_{12}, \ldots, x_{1k_1}), (x_{21}, x_{22}, \ldots, x_{2k_2}), \ldots, (x_{n1}, x_{n2}, \ldots, x_{nk_n}))$ implies that $\sigma(x_{ij}) = x_{i[(j+1)(mod\ k_i)]}$. In words, we list down the cycles in the permutation.

A *cyclic permutation* is one that has only one cycle.

Lemma 9 (Cycle Conjugacy Lemma)**.** *If $\sigma, \tau \in S_k$ are cyclic permutations, then they are conjugate, i.e., $\exists \gamma \in S_k : \sigma = \gamma\tau\gamma^{-1}$.*

To see this, suppose $\sigma = ((s_1, s_2, \ldots, s_k))$ and $\tau = ((t_1, t_2, \ldots, t_k))$ in the cycle representation. One can then verify that $\gamma = ((t_1, s_1), (t_2, s_2), \ldots, (t_k, s_k))$ (in the pointwise representation) fits the bill like a charm.