1 Branching programs

In the last lecture we introduced braching programs and described the relationship with space bounded complexity classes.

**Theorem 1.** Let $S(n) \geq \log n$. Then $\text{DSPACE}(S(n)) \subseteq \text{BPSIZE}(2^{O(S(n))})$.

Conversely, given as input a branching program of size $S$ together with a $x$, we can in space $O(\log S)$ compute the output of the program on input $x$. Thus we can obtain the following characterization of $L$ in terms of logspace uniform branching programs of polynomial size.

**Theorem 2.** $L = U_L - \text{BPSIZE}(n^{O(1)})$

### 1.1 Constant width

We are next going to consider restricting the class of polynomial size branching programs. We will see (an example of) that one can obtain characterizations of subclasses of $L$ in this way.

A digraph $D = (V,A)$ is a **layered** directed graph if there is a partition $V = V_0 \cup V_1 \cup \ldots V_h$ of the nodes into **layers**, in such a way that for all edges there is an $i$ such that the edge goes from layer $V_i$ to the next layer $V_{i+1}$.

We define the **width** of a layered graph $D$ as $\max_i |V_i|$. This naturally generalized to the width of a branching program: We say that a branching program is of width $W$ if there is a layering of the nodes in the underlying graph $D$ of the branching program, such that under this layering $D$ is of width $W$.

With these definitions in place, we can now state the following result, characterizing constant width polynomial size branching programs. The result was at the time it appeared very surprising. Note at each layer of the branching program can only encode $O(1)$ bits of information, but despite that fact it is possible to compute the majority function of $n$ bits, which would intuitively seem to require having an accumulator of $O(\log n)$ to be efficiently computable.

**Theorem 3** (Barrington’s Theorem). Let $L$ be a language. Then $L$ can be computed by a family of $O(1)$-width BP of size $n^{O(1)}$ if and only if $L \in \text{NC}^1$ (i.e. $L$ can be calculated by a family of boolean circuits of depth $O(\log n)$ and size $n^{O(1)}$)

**Remark 4.** We can in fact ensure width 5 in the above equivalence. Also, the theorem holds both in the nonuniform case, as stated above, but in the uniform cases, such as logspace uniformity, or even stronger notions.

One direction of the theorem is easy to obtain. Namely, given a layered braching program $P$ computing a language, we can split the program into the layers. We evaluate the input for each layer, and view the resulting bipartite graph as a matrix of constant size. These matrices can then
be multiplied together in a tree-wise fashion using Boolean matrix multiplication. This computes
the transitive closure between the first layer and the last layer, and we can then directly see if there
is a path from the start vertex to an accepting vertex.

For proving the other direction of the theorem, it is convenient to restrict the kinds of branching
programs to a special form. More precisely, we consider branching programs of the following form:

- All layers have the same number of nodes.
- In each layer a specific variable is read at all nodes.
- Between two layers, the edges that are followed when the variable read has value 0 form a
  permutation, and the edges that are followed when the variable read has value 1 form another
  permutation.

We will rephrase this as follows.

**Definition 5.** A program $P$ over $S_5$ is given by a sequence of instructions. Every instruction is of
the form $< i, g, h >$, where $i \in \{1, \ldots, n\}$ is an index, and $g$ and $h$ are permutations of
$\{1, \ldots, 5\}$. The length of the program is the number of instructions.

The output of instruction $< i, g, h >$ is $g$ if $x_i = 1$, $h$ if $x_i = 0$. The output of the program $P$
the product of the output of all the instructions.

(This corresponds to a BP med width = 5, except for not stating start vertex and output
vertices)

**Definition 6.** $P$ 5-cycle recognizes a language $L$ on input length $n$ if there exists a 5-cycle $\sigma$, so
that:

- If $x \in L$ then the output of $P$ on input $x$ is $\sigma$.
- If $x \notin L$ then the output of $P$ on input $x$ is $e$ (the identity permutation).

We then also say that $P$ $\sigma$-recognizes the language $L$.

Suppose now we have a program $P$ over $S_5$ that $\sigma$-recognizes $L$. We can convert this into a
branching program of width 5 that computes $L$ as follows. Each instruction of the program gets
turned into edges between layers in the branching program. The nodes of layer $j$ all read the variable
read by instruction $j$, and the edges correspond to the permutations $g$ and $h$ of instruction $j$.

Now, choose $i \in \{1, \ldots, 5\}$ so that $\sigma(i) \neq i$. We can then let $i$ be the start node, and in the final
layer we let node $\sigma(i)$ have output 1 and all other nodes have output 0.

Before we begin the proof of Barrington’s theorem, we establish the following lemmas.

**Lemma 7.** There exists 5-cycles $g, h$ so that $ghg^{-1}h^{-1}$ is a 5-cycle.

Proof. $g = (1 2 3 4 5), h = (1 3 5 4 2)$
$ghg^{-1}h^{-1} = (1 2 3 4 5) (1 3 5 4 2) (5 4 3 2 1) (2 4 5 3 1) = (1 3 2 5 4)$

**Lemma 8.** Let $\sigma$ and $\tau$ be 5-cycles. If $P$ $\sigma$-recognizes $L$, there exists $P'$ of the same length
that $\tau$-recognizes $L$. 

2
Proof. Since \( \sigma \) and \( \tau \) are both 5-cycles, there exists a permutation \( \varphi \) such that \( \tau = \varphi \sigma \varphi^{-1} \). Now in the program \( P' \), replace instruction \( <i,g,h> \) of \( P \) with the instruction \( <i,\varphi g \varphi^{-1},\varphi h \varphi^{-1}> \).

Notice then, that \((\varphi \sigma_1 \varphi^{-1})(\varphi \sigma_2 \varphi^{-1}) \cdots (\varphi \sigma_k \varphi^{-1}) = \varphi \sigma_1 \cdots \sigma_k \varphi^{-1} \).

From this we see that when \( P \) outputs \( \sigma \), \( P' \) outputs \( \varphi \sigma \varphi^{-1} = \tau \), and when \( P \) outputs \( e \), \( P' \) outputs \( \varphi e \varphi^{-1} = e \).

Proof of Barrington' Theorem. Let \( C \) be a circuit of depth \( d \), that (without loss of generality) consists exclusively of AND and NOT gates. We then make a program \( P \leq 4d \) inductively in \( d \):

Base case, \( d = 0 \) : We have that \( C \) is a constant or a variable. If \( C = x_i \) we take the instruction \( < x_i,\sigma,e > \), if \( C = 1 \) we take the instruction \( < x_1,\sigma,\sigma > \), and if \( C = 0 \) we take the instruction \( < x_1,e,e> \).

Next for the general case, \( d + 1 \).

- \( C = \neg C' \).

By induction there exists \( P' \), that \( \sigma^{-1} \)-recognizes the language that \( C' \) calculates. Then simply add the instruction \( < x_1,\sigma,\sigma > \) after \( P' \). Then the language that \( C \) calculates is \( \sigma \)-recognized.

- \( C = C_1 \land C_2 \).

Again by induction, there exist \( P_1 \) and \( P_2 \) of length \( \leq 4d \), so that \( P_1 \ g \)-recognizes the output of \( C_1 \) and \( P_2 \ h \)-recognizes the output of \( C_2 \).

In the same manner there exist \( P'_1 \) and \( P'_2 \) of length \( \leq 4d \), so that \( P'_1 \ g^{-1} \)-recognizes output of \( C_1 \) and \( P'_2 \ h^{-1} \)-recognizes output of \( C_2 \).

We then form \( P \) as the concatenation of \( P_1, P_2, P'_1, P'_2 \). To see that \( P \) computes the correct function, we just check the truthtable.

<table>
<thead>
<tr>
<th>( C_1(x) )</th>
<th>( C_2(x) )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P'_1 )</th>
<th>( P'_2 )</th>
<th>( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>g</td>
<td>e</td>
<td>( g^{-1} )</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>e</td>
<td>h</td>
<td>e</td>
<td>( h^{-1} )</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>h</td>
<td>h</td>
<td>( g^{-1} )</td>
<td>( h^{-1} )</td>
<td>( 13254 )</td>
</tr>
</tbody>
</table>

\[
\]

1.2 Width 3

We will now consider programs over \( S_3 \) (Permutation of \( \{1, 2, 3\} \)), corresponding to a special case of width 3 branching programs. We will see that these are fully correspond to a class of Boolean circuits.

Definition 9. A \( \text{MOD}_3 \circ \text{MOD}_2 \) circuit is a boolean circuit of depth 2, with \( \text{MOD}_3 \) gate as output, which in turn take \( \text{MOD}_2 \) gates as inputs.

Theorem 10. Let \( L \) be a language. Then \( L \) can be computed by a family of programs over \( S_3 \) of polynomial length if and only if \( L \) can be computed by a family of \( \text{MOD}_3 \circ \text{MOD}_2 \) circuits of polynomial size.

Fact 11. \( S_3 \) is generated by \( g \) and \( h \) where \( g = (123) \) , \( h = (12)(3) \).
Lemma 12. $hgh^{-1}g^{-1} = g$

Proof. $hgh^{-1}g^{-1} = (12)(3)(123)(12)(3)(321) = (123) = g$ \)

Proof of Theorem. First we show how to convert a $\text{MOD}_3 \circ \text{MOD}_2$ circuit $C$ into a program over $S_3$. Let $C_1, \ldots, C_k$ be the subcircuits of $C$ consisting of single $\text{MOD}_2$ gates. We will assume that $(\sum_i C_i(x)) \mod 3 \in \{0, 1\}$, for all $x$. This can be assumed at a cost of at most squaring the size of the circuit (see the proposition below).

First we construct a program for each $\text{MOD}_2$ gate $C_j$. This program consists of a sequence of instructions for each input to the $\text{MOD}_2$ gate. If $x_i$ is an input for the $\text{MOD}_2$ gate we include the instruction $<i, h, e>$. If the constant 1 is an input for the $\text{MOD}_2$ gate, we include the instruction $<1, h, h>$. Now this program $h$-computes the output of the $\text{MOD}_2$ gate. Denote this program by $P$. Consider now the program $P'$ given by the list of instructions $P < x_i, g, g > P < x_i, g^{-1}, g^{-1} >$. By the lemma, we obtain that $P'$ $g$-computes the output of the $\text{MOD}_2$ gate.

Next to form a program for the full circuit $C$, take programs $P'_1, \ldots, P'_k$ for each $\text{MOD}_2$ gate as constructed above, and concatenate all these. Now in case $C(x) = 1$ the number of subprograms that output $g$ is 1 modulo 3, and hence the output of the program is $g$. In case $C(x) = 0$ the number of subprograms that output $g$ is 0 modulo 3, and hence the output of the program is $e$.

Next we show how to convert a program $P$ into an equivalent $\text{MOD}_3 \circ \text{MOD}_2$ circuit. Similarly to the case of $S_5$ we can assume that $P$ $g$-computes $L$. Next, by a simple transformation we can assume $P$ satisfies the following properties: All instructions are of the form $<y, g, e>$ or $<y, h, e>$, where $y$ is a variable or its negation.

Imagine having evaluated all instructions, and now we need to compute a product of $h, g$ and $e$'s. The insight we require is that $hg = g^2h$ (and thus also $gh = hg^2$). In other words, we can “shift” an $h$ to the other side of an $g$ by squaring the $g$. We will imagine now that we shift all the $h$ outputs to the right side of the product, changing the $g$'s appropriately.

Note that all these $h$ elements will cancel out, since the full product is $g$ or $h$. In this shifted product we will for each position of the program (that can output $g$) compute whether there is contributed $g$, $g^2$ or $e$. This then depends on the parity of the number of $h$'s produced in previous positions. Let $r$ denote this parity, and let $<x_i, g, e>$ be the instruction we consider. Then the value that must be fed to the output gate is then $x_i(r + x_i) = (2^{x_i} - 1)(2^r + 2^{x_i} - 2)$. Following the argument of the proposition below, one can see that this can be obtained by feeding a constant number of $\text{MOD}_2$ gates to the output.

Proposition 13. Let $C$ be a $\text{MOD}_3 \circ \text{MOD}_2$ of size $S$. Then there is a $\text{MOD}_3 \circ \text{MOD}_2$ circuit $C'$ of size $O(S^2)$ equivalent to $C$, and such that the number of $\text{MOD}_2$ subcircuits that evaluate to 1 on input $x$, is always congruent to 0 or 1 modulo 3, for any input $x$.

Proof. Let $C_1, \ldots, C_S$ be the $\text{MOD}_2$ subcircuits of $C$. Each $C_i$ can be written on the form

$$C_i(x) = (2^{a_{i1}x_1+\cdots+a_{in}x_n+c_i} - 1 \mod 3)$$

and thus the output of $C$ is 1 if and only if

$$\sum_i (2^{a_{i1}x_1+\cdots+a_{in}x_n+c_i}) - S \not\equiv 0 \mod 3$$
By use of Fermat’s little theorem, we then have

\[ C(x) = \left( \sum_i (2^{a_i}x_1 + \cdots + a_{i_n}x_n + c_i) - S \right)^2 \mod 3 \]

Expanding this sum, we may convert each of the \( O(S^2) \) terms to a \( \text{MOD}_2 \) gate and obtain the desired circuit \( C' \) by feeding all these to a \( \text{MOD}_3 \) output gate.