Lecture 2: Barrington's Theorem

Anup Rao

April 3, 2020

Natural models often have unexpected connections between them. Let us take a brief interlude to explore one such unexpected connection between branching programs and circuits, that was discovered by Barrington. Barrington showed:

Theorem 1. *If $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of depth d, then it can be computed by a branching program of width 5 and length $O(4^d)$.*

The theorem is not known to hold with width 4.

This is a really powerful statement. It is especially useful if you want to prove lower bounds — if you want to show that a function cannot be computed in small depth, you can try to prove that the function cannot be computed using a small width branching program of small length. It is much easier to show the converse of the theorem:

Theorem 2. *If $f : \{0,1\}^n \to \{0,1\}$ can be computed by a branching program of width $O(1)$ and length 2^d, then it can be computed by a circuit of depth $O(d)$.*

Sketch of Proof Every width w branching program can be thought of as computing a function $g_x : [w] \to [w]$, where x is the input to the program. We shall prove inductively that you can compute the function g_x in depth Cd, for some large constant C.

The idea is to break up the program into the first half of the program, which computes h_x, and the second half, which computes q_x. Then $g_x = h_x \circ q_x$ is composition of these two functions. We recursively compute h_x and q_x. This computation should take depth $C(d-1)$. Then we use a constant number of gates to compute g_x from the descriptions of the two functions. Since the width is just a constant, this takes depth C for some constant C. Our final depth is $C(d-1) + C = C(d)$. ■

Now, let us turn to proving Theorem 1.

Proof We are given a circuit of depth d computing f and need to compute the same function using a width 5 branching program. We shall restrict our attention to width 5 branching programs that compute permutations of $[5] = \{1,2,3,4,5\}$. Before we give the construction, we need to describe some nice properties of *cyclic* permutations.

A cyclic permutation is a permutation π with the property that if you start at 1, and keep applying the permutation, you eventually

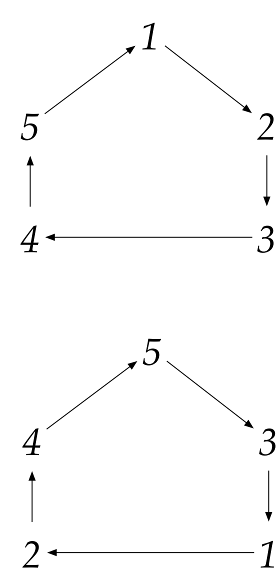

Figure 1: Two cyclic permutations.

visit all elements of [5]. For example, the permutation shown in Figure 1 are cyclic. Here are some nice properties of cyclic permutations. These are all easy to verify, but we leave it as an exercise to do it:

- If π, σ are cyclic, then so is $\pi \circ \sigma = \pi\sigma$.
- If π is cyclic, then so is π^{-1}.
- There are two cyclic permutations of [5], π, σ with the property that $\pi\sigma\pi^{-1}\sigma^{-1}$ is another cyclic permutation. This will be called the *commutator property* below. For example, set $\pi = (12345), \sigma = (13542)$, and then the composition is (13254).
- For any two cyclic permutations π, σ, there is a permutation τ (not necessarily cyclic), such that $\tau\pi\tau^{-1} = \sigma$. This we be called conjugation.

Now, for the purpose of carrying out the proof, we shall design a branching progam that on input x computes a permutation π_x, such that if $f(x) = 0$, then π_x is the identity permutation, but if $f(x) = 1$, then π_x is a fixed cyclic permutation, say $\gamma = (12345)$. This branching program computes $f(x)$.

Suppose we have already made a program computing π_x that represents $g(x)$, and we want to compute $\neg g(x)$. To do this, we simply add a layer that computes γ^{-1}. The new program computes $\pi_x\gamma^{-1}$. Call the new program σ_x. If $g(x)$ is 0, $\sigma_x = \gamma^{-1}$, and if $g(x) = 1$, σ_x is the identity permutation. Now, by conjugation, there is another permuation τ such that $\tau\gamma^{-1}\tau^{-1} = \gamma$. We apply two more layers to implement this, and so recover the program that corresponds to $\neg g(x)$.

Suppose the final gate of the circuit is a $\wedge$ gate. So, the final output is $f(x) = g(x) \wedge h(x)$. Then, by induction we have two programs, one computing π_x that corresponds to $g(x)$, and the other computing σ_x that corresponds to $h(x)$. After doing some conjugation, we can ensure that if $g(x) = h(x) = 1$, then π_x, σ_x satisfy the commutator property. If either of them is the identity, then we have $\pi_x\sigma_x\pi_x^{-1}\sigma_x^{-1}$ is also the identity. So, we get that $\pi_x\sigma_x\pi_x^{-1}\sigma_x^{-1}$ is cyclic if and only if $f(x) = 1$. Applying another conjugation gives us back the final program.

Gates that compute $\vee$ can be handled using the above methods, since $g(x) \vee h(x) = \neg(\neg g(x) \wedge \neg h(x))$.

We see that the length of the program generated in the above process satisfies $\ell_d \leq 4\ell_{d-1} + O(1)$. The solution to this recurrence is $\ell_d \leq O(4^d)$. ■