CS 121: Introduction to Theoretical Computer Science Fall 2019

Section 2— September 2019

Prof. Boaz Barak *Pranay Tankala, Laura Pierson*

1 Universal NAND-CIRC Program

1.1 Conceptual Overview

- Since each program (e.g. NAND-CIRC program) is finite, we can encode it in the same way we can encode inputs to it (as a sequence of 0s and 1s). Therefore we can treat representations of code (in the form of bit sequences) as input to code.
- Every finite function can be computed by some NAND-CIRC program.
- Since NAND-CIRC programs are finite (after fixing the size of inputs and outputs), so too is the function that executes a given input on a given NAND-CIRC program. Thus, this function can also be computed by some NAND-CIRC program.

1.2 NAND-CIRC Program Representation

We will represent a NAND-CIRC program P with n inputs, m outputs, s lines, and t distinct variables as the triple (n, m, L) where L is a list of s triples of the form (i, j, k) for $i, j, k \in [t]$.

For every variable of P, we assign a number in $[t]$ as follows:

- For every $i \in [n]$, the variable `X[i]` is assigned the number i.
- For every $j \in [m]$, the variable `Y[j]` is assigned the number $t - m + j$.
- Every other variable is assigned a number in $n, n+1, \dots, t-m-1$ in the order of which it appears.

Each triple (i, j, k) represents a line `u = NAND(v, w)`, where i, j, k are the numbers we assigned to the variables `u, v, w`, respectively.

Moreover, one can show that if the list L corresponds to an s-line NAND-CIRC program, then L can be encoded as a binary string of length $S(s) = 3s\lceil \log(3s) \rceil$.

1.3 Formal statement of universality

Let us define the function $EVAL_{s,n,m} : \{0,1\}^{S(s)+n} \to \{0,1\}^m$ as follows: if $p \in \{0,1\}^{S(s)}$ represents a size s NAND-CIRC program P with n inputs and m outputs, and if $x \in \{0,1\}^n$ is any string, then $EVAL_{s,n,m}(px)$ is the output of P on input x.

We have the following important theorem regarding the $EVAL_{s,n,m}$ function:

Theorem 1 (Efficient bounded universality of NAND-CIRC programs)**.** *For all $s, n, m \in \mathbb{N}$ with $s \geq m$, there exists a NAND-CIRC program of at most $O(s^2 \log s)$ lines that computes $EVAL_{s,n,m}$.*

1.4 Counting Arguments

Our representation of NAND-CIRC programs as strings leads to two other important results:

Theorem 2 (Counting programs)**.** *For every n, m, s with $n, m \leq 3s$, $|SIZE_{n,m}(s)| \leq 2^{O(s \log s)}$.*

Theorem 3 (Counting argument lower bound)**.** *There is a function $F : \{0,1\}^n \to \{0,1\}$ such that the shortest NAND-CIRC program to compute F requires $2^n/(100n)$ lines.*

1.5 Practice Problem

Problem 1. Let p be a string representation of

$$(3, 1, ((3,0,0), (4,1,1), (5,2,2), (6,3,4), (7,6,6), (8,5,7)))$$

Compute $EVAL_{6,3,1}(p, (1,0,1))$.

Solution. String p represents a NAND-CIRC program 6 lines, 3 inputs, and 1 output. Explicitly, the program is

```
u3 = NAND(X[0], X[0])
u4 = NAND(X[1], X[1])
u5 = NAND(X[2], X[2])
u6 = NAND(u3, u4)
u7 = NAND(u5, u5)
Y[0] = NAND(u5, u7)
```

We can check that the output of running this program on input $x = 101$ is $y = 1$. □

2 Programs with Loops

2.1 Conceptual Overview

- A finite NAND-CIRC program can only compute a finite function. This is a significant drawback, since we would like a universal notion of computation to inputs of arbitrary length.

The solution is to extend the NAND-CIRC language to handle inputs of every possible size, including inputs larger than the program itself.

- A function is **computable** if there is a NAND-TM program that computes it.
- NAND-TM is a **uniform** model of computation, in contrast to NAND-CIRC, which is **non-uniform**.
- NAND-TM = NAND-CIRC + loops + arrays
- A **total** function is one that can evaluate the entire domain. A **partial** function only computes over some subset of the domain.
- NAND-TM, Turing machines, and essentially all programming languages have equivalent computational power.

2.2 Turing Machines

The components of a Turing Machines are:

- k states
- A tape of infinite length
- A head keeping track of current location on tape
- Some alphabet Σ, which we will define in our explanation to be $\{0, 1, \triangleright, \varnothing\}$

In addition, to fully define a Turing machine, we also need a function $M : [k] \times \Sigma \to \Sigma \times [k] \times \{L, R\}$.

A Turing machine computes a function $F : \{0,1\}^* \to \{0,1\}^*$ if when the Turing Machine runs on input x, the Turing machine halts with only $F(x)$ written on the tape.

The Turing Machine starts with only x written on the tape and the head pointing to the first cell of the tape. At each step, the Turing Machine reads the value at the cell the head points to. This value, along with the current state $s \in [k]$ are inputs to the function M. M outputs a new state s', a new value $v' \in \{0, 1\}$ and L or R. The Turing machine will then write v' to the tape, remember the new state s' and move the head left or right. When the Turing machine reaches state $s = k - 1$, it halts, at which point $F(x)$ is written on the tape.

2.3 NAND-TM Progamming Language

NAND-TM programs add the following features on top of NAND-CIRC:

- We add a special integer valued variable `i`
- We support arrays by allowing variable identifiers to have the form `Foo[i]`. `Foo` is an array of Boolean values, and `Foo[i]` is the value of this array at index equal to the current value of the variable `i`.

- Input `X` and output `Y` are now considered arrays with values of zeroes and ones. We also add two new arrays `X_nonblank` and `Y_nonblank` to mark their length, where `X_nonblank[i]`$= 1$ iff `i` is smaller than the length of the input, and `Y_nonblank[j]`$= 1$ iff `j` is smaller than the length of the output.
- We add a special `MODANDJUMP` instruction in the last line that takes two boolean variables a, b as input and does the following:
  - If at least one of a or b is 1, `MODANDJUMP` jumps to the first line of the program. If $a = b = 1$ it increments `i` by 1, if $a = 0, b = 1$ it decrememts `i` by 1, and if $a = 1, b = 0$ it keeps `i` the same.
  - If $a = b = 0$ then `MODANDJUMP`(a, b) halts execution of the program.

2.3.1 Practice Problems

Problem 2. What function does the following NAND-TM program compute?

```
temp_0 = NAND(X[0],X[0])
Y_nonblank[0] = NAND(X[0],temp_0)
temp_1 = NAND(X[i],X[i])
temp_2 = NAND(Y[0],Y[0])
Y[0] = NAND(temp_1,temp_2)
MODANDJUMP(X_nonblank[i], X_nonblank[i])
```

Solution. It computes the OR function on inputs of arbitrary length. □

Problem 3. Write a NAND-TM program P that computes that parity of the number of 1s in a given string.

Solution. One possible solution is

```
temp_0 = NAND(X[0],X[0])
Y_nonblank[0] = NAND(X[0],temp_0)
temp_1 = NAND(X[i],Y[0])
temp_2 = NAND(X[i], temp_1)
temp_3 = NAND(Y[0], temp_1)
Y[0] = NAND(temp_2,temp_3)
MODANDJUMP(X_nonblank[i], X_nonblank[i])
```

□

2.4 NAND-TM Syntactic Sugar

Like for NAND-CIRC, we can add additional syntactic sugar to NAND-TM to make programs easier to read, e.g.:

- Syntactic sugar from NAND-CIRC; `if/then` conditionals
- Arrays with multiple indices
- Other index variables besides `i`
- The `GOTO` command that allows us to jump to a specific line of code instead of the start
- Nested loops and other loops such as `while` and `for` loops

How would we go about implementing each of these things in NAND-TM?

2.5 NAND-TM Computation

Let P be a NAND-TM program. For every input $x \in 0, 1^*$, we define the output of P on input x (denotes as $P(x)$) to be the result of the following process:

- Initialize the variables `X[i]`$= x_i$ and `X_nonblank[i]`$= 1$ for all $i \in [n]$ (where $n = |x|$). All other variables (including `i` and `loop`) default to 0.
- Run the program line by line. At the end of the program, if `MODANDJUMP` has at least one input 1, then increment/decrement `i` according to the inputs of `MODANDJUMP` and go back to the first line.
- If `MODANDJUMP` has both inputs 0 at the end of the program, then we halt and ouptput `Y[0]` , . . ., `Y[`$m-1$`]` where m is the smallest integer such that `Y_nonblank[`m`]`$= 0$.

If the program does not halt on input x, then we say it has no output, and we denote this as $P(x) = \perp$.

Let $F : \{0,1\}^* \to \{0,1\}^*$ be a function and let P be a NAND-TM program. We say that P **computes** F if for every $x \in \{0,1\}^*$, $P(x) = F(x)$. We say that a function F is **NAND-TM computable** if there is a NAND-TM program that computes it.