CS 181:
Introduction to Theoretical Computer Science

Einar Balan

Contents

1 | Computation and Representation

Major Idea: What can we compute? What does it mean to compute?

There are two aspects of computing:

1. Data (Representations of Objects)
   - We can represent numbers in many ways i.e. roman numerals or the place-value system we are familiar with today
   - Some representations are easier to work with than others – to convey the distance to the moon in Roman numerals would fill a small book
   - Clearly choosing the right representation can have a *dramatic* effect on computation
2. Algorithms (Operations on Data)
   - In general, there is more than one way to accomplish the same task; some better than others
   - To multiply, we can either perform repeated additions or the typical gradeschool multiplication algorithm
   - Gradeschool multiplication is far more efficient as an $O(n^2)$ algorithm compared the to the exponential nature of repeated addition
   - Choosing the right algorithm is also incredibly important to computation

Takeaway: It is important to utilize both a good data represntation and a good algorithm in all computing tasks.

Representations

- In general we can represent many objects as a sequence of 0's and 1's. Anything from images, text, video, audio, databases, etc. can be encoded in binary.
- BIG IDEA: We can compose representations in order to represent any object i.e. if you can represent objects of type T, then you can also represent lists of that object

Definition Representation, where E is one-to-one

$$E : O \to \{0,1\}^*$$

Example Represent natural numbers $E : N \to \{0,1\}^*$

Couple options:

- Unary: E(n) = 0000...0 (n + 1 zeroes)
- Binary: Standard binary encoding defined as follows

$$NtoB(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ NtoB(\lfloor \frac{n}{2} \rfloor) \circ (n\%2) & \text{else} \end{cases} \tag{1.1}$$

- We know an encoding E is valid iff there exists a decoding function s.t. $D : \{0,1\}^* \to O$ and $D(E(x)) = x$

Example $E : Z \to \{0,1\}^*$

$$ZtoB(n) = \begin{cases} 0 \circ NtoB(n) & \text{if } n >= 0 \\ 1 \circ NtoB(-n) & \text{else} \end{cases} \tag{1.2}$$

Example Represent rational numbers $E : Q \to \{0,1\}^*$

- We know we can represent any rational number with a fraction i.e. a pair of numbers
- So if we can find a way to encode two numbers in a one-to-one way, we can represent any rational number
- Suggestions
  1. Unary(p) ∘ 1 Unary(q)
  2. Encode length of numerator
  3. Add padding to shorter number to match lengths
  4. Utilize new $\overline{ZtoB} : Z \to \{0,1\}^*$ where each bit is duplicated and a 01 is added to the end
     - $QtoS(p/q) = \overline{ZtoB}(p) \circ \overline{ZtoB}(q)$

Big Idea: We can represent objects and lists of objects as compositions of representations.

2 | Prefix Free Encoding & Models of Computation

Nonformally, a prefix free encoding is one that is easy to decode if there are encodings of several objects concatenated together. These lists of objects are easy to decode because, as implied by the name, the prefix free encoding of an object will never be a prefix within the encoding of another distinct object using the same encoding function.

Definition Prefix Free Encoding

$$E : O \to \{0,1\}^*$$

E is prefix free if $\forall x \neq y \in O, E(x)$ is not a prefix of E(y)

Example $NtoB$ (binary encoding of natural numbers)

This encoding is not prefix free. Consider NtoB(2) = 10 and NtoB(5) = 101. 10 is a prefix of 101, so it does not satisfy the PFE property.

Example $\overline{ZtoB} : O \to \{0,1\}^*$

This is the same function as in the last lecture, in which bits are duplicated and 01 is added to indicate the end. It IS prefix free because the 01 end symbol will never be found in an encoding before the end of the encoding.

Theorem Suppose we have a prefix-free encoding $E : O \to \{0,1\}^*$.

Define $\overline{E}((x_1, x_2, ..., x_3)) = E(x_1) \circ E(x_2) \circ E(x_3) \circ ... \circ E(x_n)$

Then $\overline{E}$ is a valid encoding of O^*.

Proof Suppose someone gave us the binary sequence $E(x_1) \circ E(x_2) \circ E(x_3) \circ ... \circ E(x_n)$.

We can decode it as follows:

- Keep reading from left to right until the sequence matches an encoding
- Once we find it, chop it off to recover the first object and proceed

Thus, since our encoding has a decoder, it is a valid encoding.

A quick remark regarding effiency of PFE:

- length of PFE(x) is 2|E(x)| + 2, which leads to exponential growth in nested encodings (i.e. lists of lists)
- instead, we can get a conversion where the new encoding has length $|E(x)| + 2log_2(|E(x)|) + 2$ by encoding the length of the objects instead of the data itself

Additionally, concrete code of this encoding and decoding in action can be found here.

Summary We can view all inputs (images, videos, strings, graphs, etc.) as binary strings.

Algorithms

Informally, algorithms are a series of steps to solve some problem, or a way to transform inputs to a desired output. How can we formalize this?

Definition Specification

Function $f : \{0,1\}^* \to \{0,1\}^*$

i.e. $Mult : N \times N \to N$

Additionally, we can define steps as "some basic operations."

Boolean Circuits

A boolean circuit uses AND/OR/NOT as basic operations. For concision, we will omit their definitions. AND is often denoted $\wedge$, OR is $\vee$, and NOT is $\neg$.

Example MAJ3: $\{0,1\}^3 \to \{0,1\}$

$$MAJ3(a,b,c) = \begin{cases} 1 & \text{if } a+b+c \geq 2 \\ 0 & \text{else} \end{cases} \tag{2.1}$$

In terms of boolean operations, this can be defined as follows:

$$MAJ3(a,b,c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

Example

XOR2: $\{0,1\}^2 \to \{0,1\}$

$$XOR2(a,b) = (a \wedge \neg b) \vee (\neg a \wedge b)$$

XOR3: $\{0,1\}^3 \to \{0,1\}$

$$XOR3(a,b,c) = \begin{cases} 1 & \text{if odd number of a, b, c are 1} \\ 0 & \text{else} \end{cases} \tag{2.2}$$

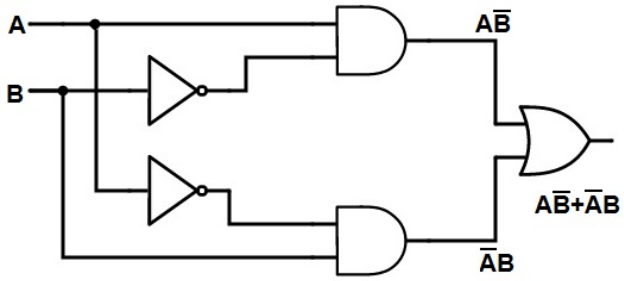
Boolean implementation of XOR3:

$$XOR3(a,b,c) = XOR2(XOR2(a,b),c)$$

$$XOR3(a,b,c) = a \oplus b \oplus c$$

In the case of a boolean circuit, "solving the problem" means computing the function and our "basic steps" are our boolean operations.

Boolean circuits can be represented using DAG's (Directed Acyclic Graphs) in circuit diagrams as follows:



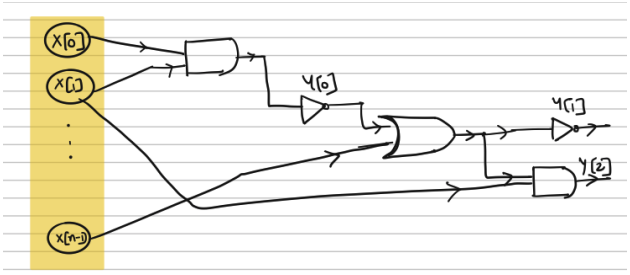
3 | Boolean Circuits

Continuing on from last time, Boolean Circuits use AND/OR/NOT as basic operations and can be represented as DAG's. Let's formally define them.

Definition Boolean Circuits

A (n, m, s)-Boolean Circuit is a DAG with $n+s$ vertices. n refers to the number of input variables, m is the number of output variables, and s is the size of the circuit.

- Exactly n of these vertices are labeled as inputs $x[0], x[1], ..., x[n-1]$
- The other s vertices are AND/OR/NOT gates
  - Each AND and OR gate has exactly two incoming edges
  - Each NOT gate has one incoming edges
- m of the gates are labeled as outputs $y[0], y[1], ..., y[m-1]$



We know that DAG's can be topologicallys sorted (i.e. there is a layering of the vertices such that for every edge (i, j), h(i) < h(j)), so Boolean Circuits can be topologically sorted too. This is important for computation of circuit outputs. How can we compute the outputs?

- layer the DAG via topological sort so that all input vertices are in layer 0
- we have computed in layers 0, 1, ..., k - 1
- now for each vertex in layer k, assign it the value of its operation on its wires (i.e. AND gate should be w1 $\wedge$ w2)
- output $y \in \{0,1\}^m$ s.t. $y = y[0] \circ y[1] \circ ... \circ y[m-1]$

Choosing Basic Operations

We stated in our definition of a boolean circuit that the basic operations it uses are AND/OR/NOT. Why did we pick these operations in particular? Could we have picked some other operations?

Definition NAND

$$\text{NAND: } \{0,1\}^2 \to \{0,1\}$$
$$\text{NAND}(a,b) = \neg(a \wedge b)$$

NAND Circuits are defined similarly to Boolean Circuits, the only difference being the basic operations available. In NAND Circuits, only NAND is used. Is there a meaningful difference between the two? Is one more powerful than the other?

We can show that NAND circuits can be easily simulated using Boolean gates.

- simply convert NAND gates to and AND followed by a NOT
- NAND $\to$ AND + NOT

Is the opposite true? Can we express Boolean Circuits only in terms of NAND gates?

- NOT(a) = NAND(a, a)
- AND(a, b) = NOT(NAND(a, b)) = NAND(NAND(a, b), NAND(a, b))
- OR(a, b) = NAND(NOT(a), NOT(b)) = NAND(NAND(a, a), NAND(b, b))

So yes, we can simulate a Boolean Circuit with a NAND circuit.

Theorem Boolean circuits are **equivalent** to NAND circuits in computational power.

Definition Equivalent

f is computable by a Boolean Circuit $\iff$ f is computable by a NAND circuit.

Theorem

Every function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a Boolean circuit of size $O(n * m * 2^n)$.

Proof (for m = 1)

Given $f : \{0,1\}^n \to \{0,1\}^m$, an arbitrary function mapping boolean strings to binary, define the set

$$S = \{\alpha : f(\alpha) = 1\}$$

For each binary string $\alpha \in 0, 1^n$, define

$$E_\alpha : \{0,1\}^n \to \{0,1\}$$
$$E_\alpha = \begin{cases} 1 & \text{if } x = \alpha \\ 0 & else \end{cases} \tag{3.1}$$

Example So if $\alpha = (1, 1, ..., 1)$, we can construct a circuit using a chain of AND gates in order to compute E_α. The same is true if $\alpha = (1, 1, ..., 1, 0)$, except that we must add in a NOT gate after the final AND gate.

If we extend this logic to the entire domain of $\{0,1\}^n$, it is clear that we can construct a circuit for arbitrary values of α in the domain.

We can use E_α to compute arbitary f.

Consider the set S from before, which is simply the set of binary strings in which f is 1. We have shown that it is possible to create a circuit for arbitary E_α (which is equal to 1 if the input x = α), so it is easy to create a circuit that computes f. It is simply

$$f(x) = \text{OR}(E_{\alpha_0}(x), E_{\alpha_1}(x), ..., E_{\alpha_{n-1}}(x)))$$

which follows from the logic that if x is equal to any one of the binary strings that cause f(x) to be 1, then the circuit should output 1. By chaining together all the circuits we constructed before with OR gates, we can construct the final circuit for arbitary f.

How many gates will we use?

$$\begin{aligned} \#\text{ gates used } &\leq |S|(2n-1) + (|S|-1) \\ &= O(n * 2^n) \end{aligned}$$

We can extend this logic beyond the $m = 1$ by using the above to compute each bit of y.

Remark: We can do better with a circuit of size $O(\frac{2^n}{n})$, but this goes beyond the scope of this class.

Some functions, such as addition and multiplication, are frequently used. As a result, engineers work to make far more efficient circuits for them than guaranteed above. Addition can be computed in an $O(n)$ circuit while multiplication can be computed in an $O(n^2)$ circuit. However, some functions require an exponential number of gates to compute. We will show this.

Big Idea We can encode circuits as binary strings.

Two corollaries to this idea:

1. Some functions need exponential size circuits
2. Universal circuits exist (i.e. general computers)

Theorem Every (n, m, s) NAND Circuit can be represented by a binary string of length O((n + s)log(n + s)).

Proof We define $\text{size}_{n,m}$(s) to be all circuits on n inputs, m outputs, with at most s gates. The goal is to find an encoding E from this function to binary strings.

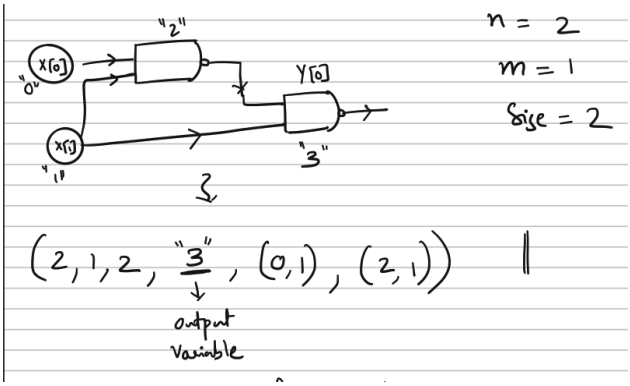
What do we need to specify in a NAND circuit?

- how many inputs? n
- how many outputs? m
- how many gates? $s_0 \leq s$
- which nodes correspond to output variables (list of indices corresponding to y[i])

- links between gates (pairs of nodes whose outputs are inputs to current index node)

We can store these features as a tuple and encode it in the standard nested PFE.

Example



How exactly to get valid encoding of list shown above?

- for n, m, and size we encode them with PFE
- same for node output numbers (we know there are m of them)
- same for pairs (simply read two of them at a time when decoding, we know there will be 2 * s_0 integers b/c 2 integer inputs for every gate)

How many bits are we using?

- recall that the PFE of an integer a takes $\leq 2log_2(a)$ bits
- $2log_2(n) + 2log_2(m) + 2log_2(s_0)$ for first 3 integers
- $m * 2log_2(n + s_0)$ for m integers in output list
- $2s_0 * 2log_2(n + s_0)$ for $2s_0$ integers in pairs
- adding these terms together we determine that this is $\leq 12(n + s)log_2(n + s)$ as stated in the theorem

---

Corollary 1. Some Functions Require Exponential Size Circuits

Theorem There exists functions $f : \{0, 1\}^n \to \{0, 1\}$ that require circuits of size $\frac{c*2^n}{n}$ for c > 0.

Proof

First some definitions.

$$\text{All}_n = \{f : \{0, 1\}^n \to \{0, 1\}\}$$

The above function is the class of all functions from n-length binary strings to 0 or 1.

$$SIZE_n(s) = \{\text{All functions with 1 bit output computable by circuits of size} \leq s\}$$

We want to show that $|All_n| > |SIZE_n(\frac{c*2^n}{n})|$.

- Counting All_n
  - we know there are 2^n possible inputs
  - each of these has 2 possible outputs
  - so we have 2^{2^n} possible functions (2 possible outputs over 2^n rows)
- Counting $SIZE_n(s)$
  - idea: count using encodings
  - # circuits $\leq$ number of binary strings of length $12(n+s)log_2(n+s)$
  - we know the number of strings of length $\leq l$ is $2^0 + 2^1 + 2^2 + 2^3 + ... + 2^l = 2^{l+1} - 1$ (geometric series)
  - so the number of circuits $\leq 2^{12(n+s)log_2(n+s)+1} - 1$

To finish we need to compare $|ALL_n|$ to $|SIZE_n(s)|$. We will skip the algebraic manipulation (see lecture notes for that), but the conclusion is that $|ALL_n| > |SIZE_n(s)|$ where $s = \frac{2^n}{24n}$. Therefore $\exists$ functions on n bits that require $\frac{2^n}{24n}$ gates to compute.

Corollary 2: Universal Circuits

We know that we can encode a circuit as a binary string. Let's define a a function to take advantage of this fact.

Definition EVAL: $\{0,1\}^{S(n,m,s)} \times \{0,1\}^n \to \{0,1\}^m$

$$\text{EVAL}(C, x) = \begin{cases} C(x) & \text{if C is a valid circuit} \\ 0^m & \text{else} \end{cases} \tag{3.2}$$

Theorem There is a circuit for $\text{EVAL}_{n,m,s}$ of size $O(s^2 logs)$.

One last major idea: Circuits are efficient.

Theorem Physical Extended Church-Turing Thesis(PECTT)

If a function can be computed using s physical resources, then it can be computed by a circuit that uses roughly s gates. In other words, circuits are about as efficient as we can get in terms of computation.

Summary

- circuits can be implemented on physical devices
- every functin can be computed by circuits
- some functions require exponential size circuits
- universal circuits of size cs^2logs can simulations all size s circuits

4 | Deterministic Finite Automata

Circuits are a great model for bounded input lengths. But what if we want to compute on some unbounded input? We know we can determine a finite answer to an infinite class of questions using an algorithm. Can we use this to create a model of computation for unbounded inputs?

Consider XOR:

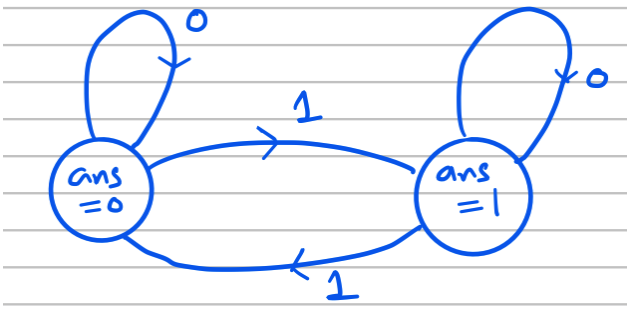
$$\text{XOR} : \{0,1\}^* \to \{0,1\}$$

$$\text{XOR}(x) = \begin{cases} 1 & \text{if number of inputs equal to 1 is odd} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

We can define an algorithm for XOR as follows:

- def XOR(x):
  - ans = 0
  - for i in range(len(x)):
    - ans = (ans + x[i]) % 2
  - return ans

This is an exmaple of aa "Single Pass Constant Memory Algorithm." We can create a diagram to represent it as follows:



Definition Deterministic Finite Automaton (DFA)

DFA with c states over {0, 1} is a pair $D \equiv (T, S)$, where $T : [c] \times \{0,1\} \to [c]$ and $S \subseteq [c]$. T is known as the transition function and defines the inputs that cause a transition in state. For example, if the current state is 0 and the input bit is 1, the transition function might output 1 to indicate a change in state from 0 to 1. S is the acceptor. 1 is output if the final state is a state $\in$ S.