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

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

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

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. $D : \{0,1\}^* \rightarrow O$ and $D(E(x)) = x$

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

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

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

- 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 $\overline{ZtoB} : Z \rightarrow \{0,1\}^*$ 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: TODO

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

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

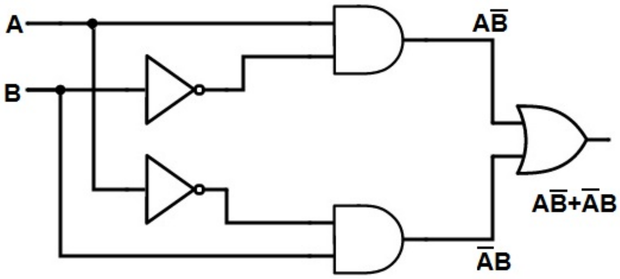
Boolean implementation of XOR3:

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

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

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:


A
B
$A\overline{B}$
$A\overline{B}+\overline{A}B$
$\overline{A}B$