Ph/CS 219A
Quantum Computation

Lecture 9. Classical Circuits

Today we begin the discussion of computation. To pave the way for quantum computing, we'll first review some notions from the theory of classical computation, including universal gates, complexity classes, randomized computation, and reversible computation.

For students with a computer science background this lecture may seem superfluous, but it provides a good launching point for other less familiar topics we wish to cover later.

See Chapter 5 of the Lecture Notes.

Boolean Functions

We can build up any classical computation as a *circuit* of *universal* logic gates. The same is true for a quantum computation.

A deterministic computation is a function mapping an *n*-bit input to a *m*-bit output.

$$f : \{0,1\}^n \rightarrow \{0,1\}^m$$

Regard this a *m* Boolean functions:

$$f : \{0,1\}^n \rightarrow \{0,1\}$$

There are many such functions. Think of *f* as a string of 2^n bits; the number of possible strings is

$$2^{2^n}; \quad e.g., \quad 2^{32} = 4.3 \times 10^9 \quad for \quad n = 5.$$

The function *f* separates the n-bit strings into two complementary sets, the strings that *f accepts* (maps to 1), and the strings that *f rejects* (maps to 0),

$$\Sigma_f = \{x \in \{0,1\}^n \mid f(x) = 1\}; \quad \bar{\Sigma}_f = \{x \in \{0,1\}^n \mid f(x) = 0\}.$$

Universal Gates

$$\Sigma_f = \{x \in \{0,1\}^n \mid f(x) = 1\}; \quad \bar{\Sigma}_f = \{x \in \{0,1\}^n \mid f(x) = 0\}.$$

$$\Sigma_f = \{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\} \Rightarrow f(x) = f^{(1)}(x) \vee f^{(2)}(x) \vee f^{(3)}(x) \vee \ldots$$

where $f^{(a)}(x) = \begin{cases} 1 & x = x^{(a)} \\ 0 & \text{otherwise} \end{cases}$

Build $f^{(a)}$ from $\wedge$ and $\neg$

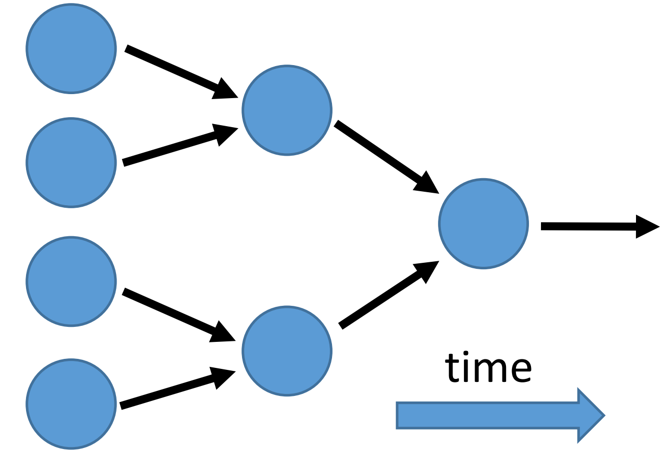
Example: $x^{(a)} = (1001) \Rightarrow f^{(a)}(x) = x_3 \wedge (\neg x_2) \wedge (\neg x_1) \wedge x_0$

Disjunctive Normal Form (DNF): f expressed in terms of $\neg, \wedge, \vee$.

Other operations: Input a variable bit (a bit from the string *x*), or input a constant (0 or 1) --- we might want some scratch space for carrying out a computation.

Circuits

The circuit we have constructed is a *directed acyclic graph*. Each vertex is a gate, and the direction indicates the order in which the gates are applied. No directed closed loops.



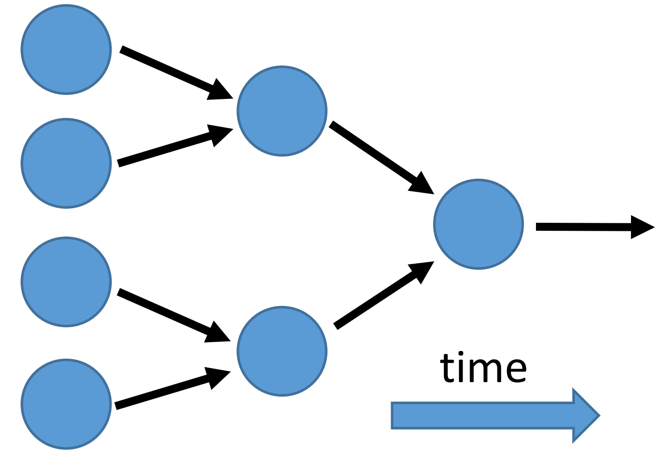
How large a circuit from DNF? No more than

$$2^n \vee, \quad n2^n \wedge, \quad n2^n \neg, \quad n2^n \ \text{inputs} \Rightarrow (3n+1)2^n \text{ total gates.}$$

In general, the number of gates is exponential in *n*, though for some functions many fewer gates suffice. Anyway, it is a remarkable fact that we can build up very complex functions from very simple components.

For *most* functions, a number of gates exponential in *n* really is necessary. A counting argument shows this.

Circuits



For *most* functions, a number of gates exponential in *n* really is necessary. A counting argument shows this.

There are *n*+5 types of gates: NOT, AND, OR, INPUT(0), INPUT(1), or INPUT on of the *n* bits of *x*. Each gate acts on outputs from earlier gates, chosen in fewer than G^2 ways, if the total number of gates is *G.* Upper bound on the number of circuits with *G* gates:

$$N_{\text{circuit}}(G) \le \left((n+5)G^2\right)^G \Rightarrow \log_2 N_{\text{circuit}}(G) \le G\left(\log_2(n+5) + 2\log_2 G\right)$$

$$= c2^n\left(1 + \frac{1}{2n}\log_2\left(\frac{c^2(n+5)}{4n^2}\right)\right) \le c2^n, \quad \text{if} \quad G = c\frac{2^n}{2n} \qquad \Rightarrow \log_2\left(\frac{N_{\text{circuit}}(G)}{N_{\text{function}}(n)}\right) \le (c-1)2^n$$

This means that for $c < 1$, the number of functions is far greater than the number of circuits. The trouble is that the number of functions is doubly exponentially large in *n*, while the number of circuit is roughly exponential in *n*.

Most functions have no special structure, so there is no way to evaluate the function which is much more efficient than a look-up table, which is essentially what DNF does.

Languages

For the purpose of discussing the hardness of decision problems, we consider not a Boolean function with a fixed input size, but rather a family of related functions with variable input size.

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

Example: FACTORING problem: $$f(x,y)=\begin{cases}1 & \text{if integer } x \text{ has a divisor } z \text{ such that } 1<z<y,\\ 0 & \text{otherwise;}\end{cases}$$

Knowing *f(x,y)* for all $y < x$ means knowing the least nontrivial factor of *x* (if there is one). In this case, the size of the input is the number of bits needed to specify *x* and *y*.

In general, the set *L* of strings accepted by a function family is called a *language.*

$$L=\{x\in\{0,1\}^* : f(x)=1\}$$

Quantifying hardness of a decision problem: How do the resources (circuit size, depth, width, …) needed to solve the problem scale with the input size? However, it would not be fair to say that a problem is easy if there is a small circuit that solves the problem, if the circuit is hard to find. Therefore we admit only *uniform* circuit families (informally, uniformity means it is easy to build the circuit with (n+1)-bit input once we have constructed the circuit with n-bit input).

P and NP

For function family $\{f_n\}$, suppose $\{C_n\}$ is a uniform circuit family that computes the functions (for all possible values of the input). The circuit family is polynomial size, if the size $|C_n|$ of C_n (the number of gates) grows no faster than a power of n.

$$\text{size}\,(C_n) \le \text{poly}(n)$$

$$\text{P} \;=\; \{\text{decision problems solved by polynomial-size uniform circuit families}\}$$

Problems in P (polynomial time) are considered *easy*, and problems not in P are *hard*.

Although the exact circuit size may depend on how we choose our universal gate set, whether the circuit size scales polynomially or not does not depend on that choice. One universal gate set can simulate another efficiently.

A bit arbitrary. $|C_n| \sim n^{1000}$ is not so easy, and $|C_n| \sim n^{\log\log\log n}$ is not so hard. And even if the power of n is modest, the constant multiplying that power could be large. But such cases are rare.

For some problems, though finding the solution to the problem may be hard, it is easy to verify the solution once it has been found. FACTORING is an example. If someone kindly provides a divisor z of x which is less than y, we can easily check that it really is a divisor, i.e. that $f(x,y)=1$.

P and NP

More formally, a language L is in NP if and only if there is a polynomial size verifier $V(x,y)$ such that

If $x \in L$, then there exists y such that $V(x,y) = 1$ (completeness),

If $x \notin L$, then, for all y, $V(x,y) = 0$ (soundness).

The verifier is the efficient uniform circuit family that checks the answer.

Completeness means that for each input in the language there is a "witness" such that the verifier accepts the input is the witness is provided.

Soundness means that that for each input not in the language the verifier rejects the input no matter what witness if provided.

NP: "Nondeterministic polynomial time" --- a bit confusing, but we're stuck with it.

Obvious that P is contained in NP --- in that case no witness is needed to verify efficiently.

A fundamental conjecture: **P ≠ NP**.

The conjecture is that the mere existence of a succinct proof of a statement does not ensure that we can find the proof by any systematic procedure in a reasonable amount of time.

NP-Completeness

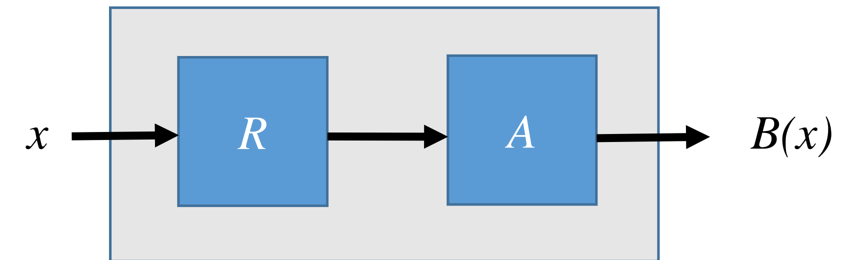
Example: CIRCUIT-SAT. The input is a Boolean circuit *C*, the problem is to determine whether any input *x* is accepted by *C*.

$$f(C) = \begin{cases} 1 & \text{if there exists } x \text{ with } C(x) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

In NP because *x* is a witness, and poly-size circuit *C* is the verifier.

Furthermore, *every problem in NP is efficiently reducible to* CIRCUIT-SAT.

B reduces to *A*: there is an efficiently computable function family *R* such that *B*(*x*) = *A*(*R*(*x*)).



B accepts *x* iff *A* accepts *R(x).*
A machine that solves *A* efficiently can be used to solve *B* efficiently.

If *B* is in NP, then it has a poly-size verifier *V(x,y)* such that *B* accepts *x* iff there exists *y* such that *V(x,y)*=1.

For each *x,* asking whether such a witness *y* exists is an instance of CIRCUIT-SAT. Thus a poly-size circuit family that solves CIRCUIT-SAT can be used to solve *B*.

NP-Completeness

Every problem in NP is efficiently reducible to CIRCUIT-SAT.

A problem *A* in NP is *NP-complete* if every problem in NP is reducible to *A*. Hence CIRCUIT-SAT is NP-complete.

If we can solve efficiently *any* NP-complete problem, then we can solve efficiently *all* problems in NP.

To show *A* is NP-complete: Show that *B* reduces to *A* where *B* is NP-complete.

Hundreds of "natural" problems are NP-complete. (But not FACTORING, or so we believe.)

Another complexity class is co-NP. A language is in co-NP if the complementary language is in NP. That is, while for NP problems the witness testifies that *x is* in the language, for co-NP problems the witness testifies that *x* is *not* in the language.

So it depends on how you ask the question. "Is there a Hamiltonian path?" is in NP (and believed not to be in co-NP); the complementary question "Is there no Hamiltonian path?" is in co-NP, and believed not to be in NP.

FACTORING is in both NP and co-NP. For NP the witness is a factor of x. For co-NP the witness is the prime factorization of x (one can verify efficiently that a number is prime).