In order to prove that P $\neq$ NP, we need to design an efficient nondeterministic Turing Machine that accomplishes some task that no efficient deterministic Turing Machine can do. In the last lecture we saw the method of diagonalization, which led to proofs of the related results P $\neq$ EXP and NP $\neq$ NEXP. Can diagonalization be adapted to prove P $\neq$ NP? It is believed that the answer is no, and later in the course we will see some concrete evidence as to why we do not expect diagonalization to resolve the P versus NP question. (The current evidence leaves open the possibility that diagonalization could play some role in an eventual proof that P $\neq$ NP, but such a proof ought to have a radically different structure from the proofs of the time hierarchy theorems.)

Intuitively, perhaps the P versus NP question is so difficult because it requires us to find a task that is difficult for *all* polynomial-time Turing Machines. To do this we need to understand the properties (and limitations) of all these Turing Machines. But many interesting properties of Turing Machines are undecidable, which makes it difficult to reason about them.

To overcome these obstacles, in the 1980s proposed studying a different model of computation – *boolean circuits*. A boolean circuit (with appropriate parameters) is thought to be roughly about as powerful as an efficient deterministic Turing Machine, yet circuits can be described in a fairly simple way. This gave computer scientists hopes that one can prove interesting theorems about the computational limitations of boolean circuits, which might eventually lead to a resolution to the P versus NP question.

The field of (boolean) circuit complexity studies the computational limitations of boolean circuits. Some very promising results in circuit complexity were obtained in the 1980s. Unfortunately, since then, there has not been as much progress. Although certain limitations of the circuit complexity approach to the P versus NP question were uncovered in the 1990s, the theory of circuit complexity is perhaps the most powerful tool we have in understanding the limits of efficient computation.

1 Circuits

Informally, a boolean circuit is some contraption with gates and wires that takes in boolean-valued inputs, runs them through some wires and gates (AND, OR, NOT) and produces an output. To formalize the definition we need to make several choices, like: Which gates do we allow in the circuit? How many wires go into each gate? Initially, these choices will not make much of a difference in the issues we want to understand. But later on we will talk about specific types of circuits, so we will choose a definition which will be most convenient for our discussion.

Definition 1. A *(boolean) circuit* with *input length* n is a directed acyclic graph with $2n$ source nodes and one sink node. Each node can have arbitrary in-degree and out-degree. The source nodes are labeled by literals $x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n$. The other nodes are labeled either "AND" or "OR".

The nodes of a circuit are called *gates*. The source nodes are called *input gates* and the sink is the *output gate*. A circuit with n inputs computes a boolean function from $\{0,1\}^n$ to $\{0,1\}$ in the

obvious way. Initially, the input is assigned to the gates $x_1, \ldots, x_n$, while the gates $\overline{x}_1, \ldots, \overline{x}_n$ get the opposite values. For any other gate, if all the gates that connect to it are assigned values, we can assign it the value which is their AND for an "AND" gate, and their OR for an "OR" gate. Eventually we obtain the value at the output gate. This is the value of the function. We will write C both for the circuit itself and for the function $C\colon \{0,1\}^n \to \{0,1\}$.

The most important complexity measure of a circuit is its size: We define the *size* of a circuit to be the total number of gates in it.

While a circuit looks nothing like a Turing Machine, we will soon show that they are intricately related. However there is one important difference: The input to a Turing Machine can be an arbitrary string of an arbitrary length, while the input to a circuit is a string of a fixed length n. To make a circuit decide a decision problem L, we have to find a way to feed it arbitrarily long instances of L. However, if we we want to design a single circuit that handles arbitrarily long inputs, its size cannot be finite. Instead we will do the following: For every input length n, we will design a separate circuit that decides L only on inputs of that particular length. So instead of getting a single circuit for L, we get an infinite family of circuits $C_0, C_1, C_2, \ldots$, where the circuit C_n decides only those inputs of L that have length n.

Definition 2. Let L be a decision problem. A *circuit family* for L is a collection of circuits $C = \{C_0, C_1, C_2, \ldots\}$, where C_n has input length n and for every $x \in \{0,1\}^*$, $C_n(x) = L(x)$, where n is the length of x.

For Turing Machines, our notion of efficient computation was polynomial time. For circuits, it is polynomial size. We say the circuit family $C = \{C_0, C_1, C_2, \ldots\}$ has *polynomial size* if there exists a polynomial p such that C_n has size at most $p(n)$ for every n. We denote the class of all decision problems that have circuit families of polynomial size by P/poly.

The reasons that polynomial size is a useful complexity bound for circuits is similar to the reason polynomial time is important for Turing Machines. First of all, polynomial-size is robust to reasonable changes in the definition of boolean circuits: For example, the size of a circuit family remains polynomial under any of the following changes in the definition:

- Instead of using AND and OR gates, suppose we use some other "basis" of gates, like AND and XOR.
- Instead of counting the number of gates towards the size of the circuit, suppose we count the number of edges (i.e. wires).
- Instead of allowing that each gate has arbitrary in-degree, suppose we insisted on in-degree two.

When we defined polynomial-time Turing Machines, we said that the reason we care about polynomial time is that it appears to describe all realistically efficient computations, and not much more. In particular, in the context of the P versus NP question, solving an NP problem in polynomial time means an improvement over exhaustive search.

We will see in a minute that $\mathrm{P} \subseteq \mathrm{P/poly}$. So polynomial-size circuits also encompass all computations that appear to be efficient in practice. However, unlike polynomial time, circuits of polynomial

size can also perform some computations that are blatantly unrealistic: For instance, polynomial size circuits can solve undecidable problems. This is related to the fact that circuit families are infinite objects – there is one for every input length – while Turing Machines are finite.

However, in the context of NP problems it is not believed that polynomial-size circuits can do much more than polynomial-time Turing Machines. In fact it is quite plausible that P $\neq$ NP *if and only if* NP $\not\subseteq$ P/poly. While we do not know how to show this, we will prove a related result later in the course. For now let us just state the conventional wisdom that while circuits are more powerful than Turing Machines, it is generally believed that this additional power cannot be harnessed for solving NP-complete problems.

One useful thing we can do with circuits is *hardwiring*: Given a circuit $C\colon \{0,1\}^n \to \{0,1\}$, we can get a new circuit $C'\colon \{0,1\}^{n-1} \to \{0,1\}$ by fixing one of the input bits of C to a specific value (0 or 1). It is easy to see that after "simplifying" C' will be at most as large as C. We can repeat the operation several times to hardwire an arbitrary subset of the input bits of C.

2 Circuits versus algorithms

We now show the reason why circuits are relevant to the P versus NP question: Polynomial-size circuit families can simulate polynomial-time Turing Machines.

Theorem 3. P $\subseteq$ P/poly.

This theorem corresponds to the intuition that we can implement any computation in hardware (a circuit) in a way that preserves the efficiency of the computation: Polynomial time computations give rise to polynomial size circuits.

Proof sketch. Let $L \in$ P, so L is recognized by a TM M that runs in time $p(n)$ time on inputs of length n for some polynomial p. Let Σ be the alphabet that M uses for its tape (which we extend so it can also describe the state of the TM). Given an input length n, we look at the *computation tableau* of M on length n inputs: This is a table of dimensions $t(n) \times t(n)$ with entries in Σ that describes the computation history of M. The cell i, j of this tableau contains the contents of the jth cell of the tape at time i (including a special marker that describes the state of the TM if the head of the tape happens to be there).

We associate a variable z_{ij} taking values in Σ with cell (i, j) of the tableau. The value in cell i at time j is then determined by a function

$$z_{i,j} = C_{i,j}(z_{i-1,j-1}, z_{i-1,j}, z_{i-1,j+1})$$

that determines the contents of cell i, j as a function of the contents of the three cells above it (only two if $i = 1$ or $i = t$). Since $C_{i,j}$ is a function from Σ^3 to Σ, it can be represented by a circuit of size $O_\Sigma(1)$. We now create a circuit C by "cascading" the circuits $C_{i,j}$ so that the input they receive comes from the top row $z_{1,j}$ and its output is the element $z_{t(n),0}$. We initialize the top row with the input x. The resulting circuit has size $O(t(n)^2)$. □

The idea in this proof is useful for establishing the NP-hardness of SAT. We will reduce from bounded halting by way of the following problem called CSAT:

INPUT: The description of a boolean circuit $C\colon \{0,1\}^n \to \{0,1\}$.

PROBLEM: Is there an $x \in \{0,1\}^n$ such that $C(x) = 1$?

Clearly CSAT $\in$ NP. We now show CSAT is NP-hard by reducing from bounded halting (BH). The key observation we will use is that the way of building a circuit from a Turing Machine described in the proof of Theorem 3 is "uniform": Given a description of a machine M an input length n, and a time bound t, we can construct in time poly$(\langle M\rangle, n, t)$ a circuit $C : \{0,1\}^n \to \{0,1\}$ such that for every x of length n, $C(x) = M(x)$.

Now we reduce BH to CSAT as follows. Given an instance $(\langle N\rangle, x, 1^t)$ of BH, we think of the nondeterministic TM N as an NP-verifier that takes input x, $|x| = n$ and witness y, $|y| = m(n)$ and verifies that y is a witness for x. We then compute a circuit $C : \{0,1\}^{n+m(n)} \to \{0,1\}$ such that $C(z,y) = N(z,y)$ for every pair (z,y) of the appropriate length. We then hardwire $z = x$ in C to obtain a new circuit $C'(y)$. By construction $C'(y)$ accepts for some y if and only if $N(x,y)$ accepts in t steps for some y, so that $(\langle N\rangle, x, 1^t) \in$ BH if and only if $C' \in$ CSAT.

Finally, we can reduce from CSAT to SAT along the following lines: Given a circuit C, apart from the input gates $x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n$, we label the other gates of the circuit by new variables y_j and write CNF formulas for each of these gates that sets y_j to an appropriate function of the inputs. Finally we add the clause y_m where y_m is the variable corresponding to the output gate. The resulting CNF is satisfiable iff C has an assignment that evaluates to 1.

3 Polynomial versus exponential size circuits

In the last lecture we said that while we cannot prove that P $\neq$ NP, we have the weaker separation P $\neq$ EXP. The analogous statement for circuits would be that exponential size families of circuits can compute more functions than polynomial-size families. Is this true? We will see that the answer is yes, but for very different reasons. This is a good illustration of the differences between Turing Machines and circuits.

Let us first consider the question of what an exponential-size family of circuits can do. It turns out that exponential-size circuit families can solve *all* decision problems! In contrast, exponential-time Turing Machines admit some hard problems, for instance the undecidable ones.

Theorem 4. *Every function $f\colon \{0,1\}^n \to \{0,1\}$ can be computed by some circuit of size $4 \cdot 2^n$.*

Proof. Let $s(n)$ be the minimal size of a circuit necessary to compute all $f\colon \{0,1\}^n \to \{0,1\}$. Assume $n > 0$. Given a function $f\colon \{0,1\}^n \to \{0,1\}$, consider the functions $f_0, f_1\colon \{0,1\}^{n-1} \to \{0,1\}$ where $f_b(x_1, \ldots, x_{n-1}) = f(x_1 \ldots, x_{n-1}, b)$. Then

$$f(x_1, \ldots, x_n) = (\overline{x}_n \wedge f_0(x_1, \ldots, x_{n-1}, 0)) \vee (\overline{x}_n \wedge f_1(x_1, \ldots, x_{n-1}, 1)).$$

Therefore, $s(n) \leq 2 \cdot s(n-1) + 4$. Solving this recursion yields $s(n) \leq 2^n(s(0)+4) - 4 \leq 4 \cdot 2^n$. $\square$

Thus exponential-size circuit families can compute all functions. In contrast, polynomial-size circuit families cannot compute all functions. The reason is that there are simply not enough circuits to handle all the functions.

Theorem 5. *There is a function $f\colon \{0,1\}^n \to \{0,1\}$ such that no circuit of size $n^{\log n}$ computes f for all sufficiently large n.*

We will prove this theorem via *the probabilistic method*: We will show that with nonzero probability, a random f fails to be computed by any circuit of size n. Therefore such an f must exist.

Proof. Let n be a sufficiently large number and C a circuit with input length n. Consider a random function $f\colon \{0,1\}^n \to \{0,1\}$, chosen by picking the value $f(x)$ uniformly and independently at random for every $x \in \{0,1\}^n$. Since the values $f(x)$ are independent and uniform, we have

$$\Pr_f[C \text{ computes } f] = \Pr_f[C(x) = f(x) \text{ for all } x] = \prod_{x\in\{0,1\}^n} \Pr_f[C(x) = f(x)] = 2^{-2^n}.$$

Now let us look at all possible circuits of size s on n inputs. (For convenience let's order the gates of the circuit so that the $2n$ input gates come first in the order $x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n$ and the output gate comes last.) How many such circuits are there? To specify such a circuit it is enough to describe

- For each non-input gate, the type of gate (AND or OR)
- For each pair of gates (u, v), whether u is connected to v.

There are 2^{s-2n} choices that specify the type of all the gate. There are 2^{s^2} ways to specify the connections between pairs of gates. So there are at most 2^{s^2+s-2n} choices of different circuits of size s. By a union bound, we have that

$$\Pr_f[\text{some } C \text{ of size } s \text{ computes } f] \leq \sum_{C \text{ of size } s} \Pr[C \text{ computes } f] \leq 2^{s^2+s-2n} \cdot 2^{-2^n} = 2^{s^2+s-2^n-2n}.$$

When n is sufficiently large, $(n^{\log n})^2 + n^{\log n} - 2^n - 2n < 0$. Then,

$$\Pr_f[\text{some } C \text{ of size } n^{\log n} \text{ computes } f] < 1.$$

Therefore, there exists some f that is not computed by any circuit of size $n^{\log n}$. □

It follows that P/poly does not contain all decision problems: If it did, then for every function $f\colon \{0,1\}^n \to \{0,1\}$ there would exist a polynomial-size that computes f. Since $n^{\log n}$ is eventually larger than any polynomial, we obtain a contradiction with Theorem 5. What about decision problems in NP? What Theorem 5 roughly says is that a random decision problem is unlikely to be in P/poly. But it is not clear if this has any bearing for NP problems, which (for all we know) do not "look" especially random.