[CSS.203.1]: Computational Complexity (2026-I)

Summary scribes

git info: cba76d7 , (2026-02-12 08:20:27 +0530)

Lectures

A dummy's guide

Theorem 1. *This is a cool theorem*

In fact we can refer to theorems using Theorem 1.

You could also define lemma, corollary etc. (take a look at `thmmacros.tex` for the environments).

Other useful macros are present in `lazy_macros.tex` and `common_macros.tex`. You can add more to them if required.

One pet-peeve: There are many times when people have to define a function called 'blah'. There are multiple ways of doing this:

- (worst) `$blah$` which renders as $blah$
- (bad) `$\text{blah}$` which renders as blah
- (better, but not ideal) `$\mathrm{blah}$` which renders as blah
- (right) `$\operatorname{blah}$` which renders as blah

Here is a place where you can see the difference between these:

$$sin\theta \quad \sin\theta \quad \sin\theta \quad \sin\theta$$

And the same within an emphasised text

$$sin\theta \quad \sin\theta \quad \sin\theta \quad \sin\theta$$

Here is a general guide for deciding which to use:

> If you just want to use text within a math block, then use `\text`. For examples such as defining a set called 'PRIMES' (which are not used as functions or operators), you may use `\mathrm{PRIMES}`. If you are defining functions or operators, then use `\operatorname{blah}` as it adds the right spacing around it.
>
> Using `$blah$` should only be used when you are multiplying four variables called b, l, a and h.

Lecture 1

Starting with circuits

Scribe: Ramprasad Saptharishi

Topics covered in this lecture

- General introduction to the course
- Different circuits for Th_2

1.1 Introduction

Complexity is about the study of *computational tasks* that can be performed by *computational models* under *certain resource contraints*. This takes multiple avatars and we will see several of them throughout the course.

To begin with, we are going to deal with Boolean functions $f : \{0,1\}^n \to \{0,1\}$. The computational model we will start with will just be Boolean circuits (made up of $\bigwedge, \bigvee$ gates (of arbitrary fan-in) and $\neg$ gates (of fan-in 1, obviously)).

1.1.1 An example to get comfortable with circuits

Consider the following simple function: $\mathrm{Th}_2 : \{0,1\}^n \to \{0,1\}$ that is 1 whenever the input has at least two 1s in it. There are several ways we can try to compute this function via a circuit.

- **The obvious method:** Run through every pair of variables and check if both are one:

  $$\bigvee_{i \neq j \in [n]} \bigwedge(x_i, x_j)$$

  This circuit has $O(n^2)$ gates and wires, and depth 2.

- **Linear cuts:** Run through each $i = 2, \ldots, n-1$ and check if there is a 1 among $\{1, \ldots, i-1\}$

and $\{i, \ldots, n\}$.

$$\bigvee_{i=2}^{n-1} \left(\left(\bigvee_{j=1}^{i-1} x_j \right) \wedge \left(\bigvee_{j=i}^{n} x_j \right) \right)$$

This has $O(n)$ gates, $O(n^2)$ wires and depth 3.

- **Hiding approach:** For each i, let x_{-i} be the $(n-1)$-bit string obtained by removing the i-th coordinate. If the input has at least 2 ones if and only if $\bigvee(x_{-i}) = 1$ for all i.

$$\bigwedge_{i=1}^{n} \bigvee(x_{-i})$$

(and a few more such examples)

1.2 Counting functions, and circuits

It is easy to note that the number of functions $f : \{0,1\}^n \to \{0,1\}$ is exactly 2^{2^n}. How do we count the number of circuits of a given size (where size is say the number of gates)? For simplicity, let us just assume that the circuit has fan-in bounded by 2 (if you need larger fan-in, just split that into multiple gates of the same type of fan-in 2 each).

To count this, we will just try to describe each circuit as a string (its encoding), and we will just count the number of such strings. To describe a circuit entirely, we merely need to describe each gate — what is its type, and what are the children of that gate. If we index the gates as $1, \ldots, s$ (including the literals $x_1, \ldots, x_n$ as gates as well), each gates needs 2 bits to describe its type (AND, OR or NOT), and $2 \log s$ bits to describe the two children feeding into it. Thus, every circuit of size s can be described using about $2s \log s + 2s$ gates. Thus, we have the following bound on the number of circuits of a certain size.

Observation 1.1. *The number of fan-in 2 circuits of size s is at most $2^{2s \log s + 2s}$.*

As a corollary, we immediately get the following.

Corollary 1.2 (Existence of 'hard' functions)**.** *For all n large enough, there exists Boolean functions $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by circuits of size at most $2^n/10n$.*

Proof. Set $s = 2^n/10n$ in Corollary 1.2 and we see that the number of circuits is way smaller than 2^{2^n}. □

To contrast this, note that *every* function can be computed using a circuit of about 2^n gates using the *truth-table*.

1.3 Dealing with varying lengths

One drawback of circuits is that the input length is fixed. Thus, if the idea was to compute functions $f : \{0,1\}^* \to \{0,1\}$, we cannot hope to compute it using a single circuit. Thus, we

will often be dealing with families of circuits $\{C_n : n \in \mathbb{N}\}$ such that C_n handles inputs of length in.

For such a family, we can now talk about its size as a function $s : \mathbb{N} \to \mathbb{N}$ where $s(n)$ is just the size of C_n. Now that we have such a function, we can talk about a circuit family being $O(n^2)$ size etc. to mean that $s(n) = O(n^2)$.

Thus, if we define $\mathsf{SIZE}(s)$ as

$$\mathsf{SIZE}(s) = \left\{ f : \{0,1\}^* \to \{0,1\} \ : \ f \text{ is computable by a size } s \text{ circuit family} \right\},$$

then the discussion above can be phrased as saying that $\mathsf{SIZE}(2^n \cdot n)$ contains *all* functions, and there *are* functions that are not in $\mathsf{SIZE}(2^n/10n)$. In fact, we will soon see that something stronger is true — for every 'reasonable' s, we have that $\mathsf{SIZE}(s) \subsetneq \mathsf{SIZE}(10s)$. "With more gates comes more computation"

Lecture 2

Size Hierarchy and Universal Circuits

Scribe: Aindrila Rakshit

Topics covered in this lecture

- Circuit size hierarchy
- Circuits as strings
- Universal circuits

2.1 Recap

Recall that a Boolean circuit C_n computes a function $f : \{0,1\}^n \to \{0,1\}$. To compute functions on inputs of varying lengths, we consider *families of circuits* $\{C_n\}_{n\in\mathbb{N}}$, where C_n handles inputs of length n.

The size of a circuit family is measured by a function $s : \mathbb{N} \to \mathbb{N}$, where $s(n)$ denotes the number of gates in C_n.

From the previous lecture:

- The number of Boolean functions on n bits is 2^{2^n}.
- The number of circuits of size s is at most $2^{O(s \log s)}$.
- Hence, there exist functions requiring circuit size at least $2^n/(10n)$.
- Every Boolean function is computable by a circuit of size $O(2^n \cdot n)$.

We now show that allowing more gates strictly increases the computational power of circuits.

2.2 Size Hierarchy

Theorem 2.1 (Size Hierarchy Theorem)**.** *For any* $0 \leq s \leq 2^n/(10n)$,

$$\mathsf{SIZE}_n(s) \subsetneq \mathsf{SIZE}_n(s + O(n)).$$

Proof. From the counting argument last lecture, we know that there exists a hard function $f^* : \{0,1\}^n \to \{0,1\}$ that requires circuit of size at least $2^n/(10n)$.

Order all n-bit strings lexicographically.

Define a sequence of functions $\{f^{(\bar{a})}\} : \{0,1\}^n \to \{0,1\}$ by

$$\{f^{\bar{a}}\}(x) = \begin{cases} f^*(x) & \text{if } x \leq a, \\ 0 & \text{otherwise.} \end{cases}$$

Observe that $\{f^{(\bar{0})}\}(x)$ is the constant zero function, while $\{f^{(\bar{1})}\}(x) = f^*(x)$ forall x. Moreover, $f^{(i)}$ and $f^{(i+1)}$ differ on exactly one input.

A circuit for $f^{(i+1)}$ can be obtained from a circuit for $f^{(i)}$ by adding a small equality test for the differing input and combining the outputs using the circuit in Q1 of the quiz. This increases the circuit size by $O(n)$ gates. So, size$((f^{(i+1)})) \leq$ size$((f^{(i)}))$

Thus, the circuit size increases gradually from $O(1)$ to $O(2^n/(10n))$, implying that for any $s(n)$ in this range there exists a function computable with size $s(n) + O(n)$ but not with size $s(n)$. □

2.3 Circuits as Strings

A Boolean circuit of size s can be encoded as a binary string of length $2s + 2\log s$ by describing, each gate and type using 2 bits and its two children by $\log s$ bits each.

This allows us to treat circuits themselves as inputs to other circuits.

2.4 Universal Circuits

A *universal circuit* $U_{s,n}$ takes as input:

- a description $\langle C \rangle$ of a circuit C of size at most s with n inputs, and
- an input $x \in \{0,1\}^n$,

and outputs $C(x)$.

Theorem 2.2 (Universal Circuit)**.** *For all* $n, s \in \mathbb{N}$*, there exists a Boolean circuit* $U_{s,n}$ *of size* $O(s \log s)$ *such that for every Boolean circuit* C *of size at most* s *with* n *inputs and every input* $x \in \{0,1\}^n$,

$$U_{s,n}(\langle C \rangle, x) = C(x),$$

where $\langle C \rangle$ *denotes a binary description of* C.

Proof-sketch. A Boolean circuit C of size s can be encoded using $2s + 2\log s$ bits for each gate, its type and the 2 children of each gate.

The universal circuit $U_{s,n}$ simulates C gate by gate. Let $g_1, g_2, \ldots, g_s$ be the gates of C, and let G_i denote the value of gate g_i on input x. The description $\langle C \rangle$ specifies, for each gate g_i, its gate type and its 2 children.

We can imagine that we have wires $g_1, \ldots, g_s$, which are initialised to zeroes, and wire g_i is supposed to hold the value of the gate g_i. If we wish to update the value of gate i (after having computed all the previous gates), build a circuit of the form:

$$g_i^{(\text{new})} = \text{Lookup}_2\left(c_{i,1}c_{i,2}, \text{AND}(g_{i_1}, g_{i_2}), \text{OR}(g_{i_1}, g_{i_2}), \text{NOT}(g_{i_1}), x_{i_1}\right)$$

where $c_{i,1}c_{i,2}$ is the two bits that determine the type of the gate i, and i_1 and i_2 are the two children feeding in. Of course, we also need to *lookup* g_{i_1}, g_{i_2} etc. (all the parts above in blue) from the circuit description, but they are also appropriate lookup circuits.

One can check that the size of the universal circuit is $O(s^2 \log s)$ (since there are s phases that computes one gate each, and the lookups are of size $O(s \log s)$). □

2.5 Summary

- There is a strict hierarchy of circuit size classes.
- Circuits can efficiently simulate other circuits.
- Circuit complexity is inherently non-uniform.

Lecture 3

Turing Machines

Scribe: Ananya Ranade

Topics covered in this lecture

- What are Turing Machines?
- Alphabet and Tape Reductions
- Universal Turing Machines
- Time Complexity

In this lecture we understood what are Turing Machines, and how they correctly capture the idea of computation.

3.1 Turing Machines

A Turing machine is a machine which is composed of an infinite input tape, several work tapes and a finite state machine (automaton). The input to the machine is written on the one-sided infinite input tape starting from it's left most end. It is followed by the special symbol # which is followed by $\sqcup$ all the way to infinity. The machine is allowed to make changes to the input tape once it starts running.

The tape alphabet Γ is a finite set of symbols. Here we assume it is some set containing $\{0, 1, \#, \sqcup\}$ with $\#, \sqcup$ special symbols only used to indicate end of input and space respectively.

The work tapes are initially blank one-sided infinite tape on which we can read, write and erase one symbol at a time.

The finite state machine has heads (pointers) to the input tape and each of the work tapes, and the heads can read, write or erase symbol in the current cell it is pointing at. It can also move left or right by 1 place at a time. These instructions are given by the current state of the finite state machine after reading the symbols the heads are currently reading. The finite state machine has 3 special states : $q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}$. The machine starts from initial state q_{start} with

all heads at leftmost ends of the respective tapes. It accepts the input and stops running once it reaches q_{accept} state, and it rejects the input and stops running once it reaches q_{reject} state.

Definition 3.1 (Basic aspects of Turing machines)**.**

1. *Language computed by TM M is denoted $L(M)$ which is $\{x \ : \ M \text{ on } x \text{ accepts}\}$.*
2. *M is a halting TM if it halts on all inputs.*
3. *M computes $f : \Sigma^* \to \{0,1\}$ if $f(x) = 1 \iff x \in L(M)$.*
4. *Time Complexity of M : Let $T_{M,x}$ be the number of steps M took to run on input x. Then the time complexity of M denoted T_M is a function from $\mathbb{N} \to \mathbb{N}$ where $T_M(n) = \max_{x \in \Sigma^n} T_{M,x}$. Time complexity of $f : \Sigma^* \to \{0,1\}$ is the lowest time complexity among all possible halting TM's M computing f.*

$\Diamond$

After defining so many things, it is a natural question to ask that does the time complexity change a lot based on the number of tapes or alphabet size, since in the definition of TM we allowed it to have any finite alphabet size and work tapes.

To address this issue we will prove the following theorems.

3.2 Alphabet Reduction

Theorem 3.2. *If M is a TM with tape alphabet Γ, then there is an equivalent TM M' with tape alphabet $\{0,1,\#,\sqcup\}$. Furthermore $T_{M'} = O(T_M)$ where the constant depends only on $|\Gamma|$..*

Proof-idea. In order to compensate for a smaller alphabet size, we make the finite state machine larger. Basically we will encode all symbols inΓ using roughly $log|\Gamma|$ size binary encoding. The states in the automaton will slowly read the $log|\Gamma|$ size encoding one step at a time and basically simulate one move of M in around $\log|\Gamma|$ steps. Thus, we can always reduce the tape alphabet. However, in doing so, we incur $\log|\Gamma|$ factor extra time. This is a constant which depends only on the alphabet size and not on other parameters like input and hence it is not a bad blow up. $\square$

3.3 Tape Reduction

Theorem 3.3. *If M is a TM with k tapes then there is an equivalent TM M' with just 2 tapes. $T'_M = O(T_M \log(T_M))$.*

Proof Idea : We will interleave the content on all work tapes in a single tape. We will need a symbol α' for each symbol α in the alphabet of M. This is basically to tell the i^{th} tape head pointer would have been at this position. Then, we just scan across all the k tape heads and then move. Since on input of length n it is spending atmost $T_M(n)$ time, the maximum it has to traverse to simulate 1 step of M is $T_M(n)$. So, to simulate M on length n input, it needs atmost $T_M(n)^2$ time. We can also reduce the alphabet size as done in previous part. Thus, $T_{M'} = O(T_M^2)$ where the constant depends on k, $|\Gamma|$.

We can do this in a better way. Assume that the alphabet size is $|\Gamma|^k$, and we have a 2 sided infinite work tape. From the starting head position, we split the left and right side into consecutive blocks $L_1, L_2, ...$ and $R_1, R_2, ...$ respectively, where $|L_i| = |R_i| = 2^i$. Further, at any point for all i, L_i and R_i are either full or empty and atmost one of them is full. Now, we try to simulate all k head movements by thinking of it as moving tapes instead of moving heads, and then doing the corresponding changes to the work tape. Since there are a lot of blank spaces, the amortised time to do this T_M times is $\log(T_M)$. We can convert it to smaller alphabet size and single sided infinite tape by incurring constant factor blow up. Thus, $O(T_M \cdot \log(T_M))$ time suffices.

Any TM M can be fully described by providing the description of states, transitions, tape alphabet, and number of tapes. All these can be described as binary strings using some convention. Thus, each TM has a finite encoding as a binary string. We denote it as $\langle M \rangle$. Further, we will say that any string of the form $\langle M \rangle \#\alpha$ (where # is a special symbol and α is any finite string) is also an encoding of M denoted $\langle M \rangle_\alpha$. Thus, any TM M has infintely many encodings.

3.4 Universal TM's

We can build a TM $\mathcal{U}$ which takes as input $(\langle M \rangle, x)$ and does what M does on x. Furthermore, $T_{\mathcal{U}}(\langle M \rangle, x) = O(T_{M,x} \log(T_{M,x}))$, where the constant depends only on Γ, k and not the padding α. The idea is the same as in tape reduction proof.

3.5 Time Complextiy classes

Let $f : \mathbb{N} \to \mathbb{N}$ be a function. Then, we define the following "deterministic time complexity classes":

$$\mathsf{DTIME}(f) := \{L \subseteq \{0,1\}^* \ : \ \text{There is a det. TM } M \text{ with } L(M) = L \text{ with } T_M = O(f)\}$$

Once we have the above, we can define the class P (of polynomial time computable functions) as

$$\begin{aligned}\mathsf{P} &:= \bigcup_{c \geq 1} \mathsf{DTIME}(n^c) \\ &= \{L \ : \ \exists\, c \text{ with } L \in DTIME(n^c)\}\end{aligned}$$

Bibliography