Intractability

Huan Long

Shanghai Jiao Tong University

Acknowledgements

Part of the slides comes from a similar course given by Prof. Yijia Chen.

`http://basics.sjtu.edu.cn/~chen/`

Textbook

Introduction to the theory of computation
Michael Sipser, MIT
Third edition, 2012

Outline

Main objective

Proving the existence of problems that are decidable in principle but not in practice.
– that is, problems that are decidable but intractable.

Hierarchy Theorems

Intuitively, giving a Turing machine more time or space should increase the class of problems that it can solve.

Yes, the hierarchy theorem.

Space constructible

Definition

A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is at least $\mathcal{O}(\log n)$, is called space constructible if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $\mathcal{O}(f(n))$.

In other words, f is space constructible if some $\mathcal{O}(f(n))$ space TM exists that always halts with the binary representation of $f(n)$ on its tape when started on input 1^n.

Examples for space constructible functions

- n^2
- $n \log n$
- $\log n$
- $\dots$

Note that when showing functions $f(n)$ that are $o(n)$ to be space constructible, we use a separate read-only input tape.

Space hierarchy theorem

Theorem

For any space constructible function $f : \mathbb{N} \to \mathbb{N}$*, a language* A *exists that is decidable in* $\mathcal{O}\left(f(n)\right)$ *space but not in* $o\left(f(n)\right)$ *space.*

Proof idea

We demonstrate a language A which is decidable in $\mathcal{O}(f(n))$ space while not in $o(f(n))$ space.

We describe A by giving an algorithm D that decides it. D runs in $\mathcal{O}(f(n))$ space and it ensures that A is different from any language that is decidable in $o(f(n))$ space.

Suppose M is a TM that decides a language in $o(f(n))$ space,D will implement the diagonalization method: D ensures that A different from M's language in at least one place. i.e., the place corresponding to $\langle M \rangle$.

D runs M on input $\langle M \rangle$ within the space bound $f(n)$. If M halts within that much space, D accepts iff M rejects. (If M doesn't halt, D just rejects.).

Two technical details

1. Even when M runs in $o\left(f\left(n\right)\right)$ space, it may use more than $f\left(n\right)$ space for small n. (When the asymptotic behavior hasn't 'kicked in' yet).

   Sol: Padding. $\langle M\rangle 10^{*}$ give additional opportunities to avoid $M's$ language.

2. When D runs M on some string, M may get into an infinite loop, while D should be a decider.

   Sol: Counting. Let D counts the number of steps used in simulating M and reject if the counter ever exceeds $2^{f(n)}$.

Proof

The following $\mathcal{O}(f(n))$ space algorithm D decides a language A that is not decidable in $o(f(n))$ space.

D : On input w

1. Let n be the length of w.
2. Compute $f(n)$ using space constructibility and mark off this much tape. If later stages ever attempt to use more, *reject*.
3. If w is not of the form $\langle M \rangle 10^*$ for some TM M, *reject*.
4. Simulate M on w while counting the number of steps used in the simulation. If the count ever exceeds $2^{f(n)}$, *reject*. (Note here may have a constant factor overhead: if M runs in $g(n)$ space, then D uses $d \cdot g(n)$ space to simulate M for some constant d that depends on M.)
5. If M accepts, *reject*. If M rejects, *accept*.

Space hierarchy

Corollary

For any two functions $f_1, f_2 : \mathbb{N} \to \mathbb{N}$*, where* $f_1(n)$ *is* $o\left(f_2(n)\right)$ *and* f_2 *is space constructible, SPACE*$(f_1(n)) \subset$ *SPACE*$(f_2(n))$*.*

Corollary

For any two real numbers $0 \leq \epsilon_1 < \epsilon_2$

$$\mathit{SPACE}(n^{\epsilon_1}) \subset \mathit{SPACE}(n^{\epsilon_2}).$$

Space hierarchy

Corollary

NL $\subset$ *PSPACE*.

Proof.

- By Savitch's theorem NL $\subseteq$ SPACE$(\log^2 n)$,
- While SPACE$(\log^2 n) \subset$ SPACE(n).

$\square$

Corollary

TQBF $\notin$ *NL*.

Intractable problem

Definition

EXPSPACE $= \bigcup_k$ SPACE(2^{n^k}).

Corollary

PSPACE $\subset$ *EXPSPACE*.

Proof.

SPACE$(n^k) \subseteq$ SPACE$(n^{\log n}) \subset$ SPACE$(2^n) \subseteq$ EXPSPACE. $\square$

An EXPSPACE complete problem

Extend the regular expression $(a, \epsilon, \emptyset, R_1 \cup R_2, R_1 \circ R_2, R^\star)$ with $\uparrow$: the exponentiation operation defined as:

$$R^k = R \uparrow k = \overbrace{R \circ R \circ \cdots \circ R}^{k}$$

Definition

A language B is EXPSAPCE-complete if

1. $B \in$ EXPSPACE, and
2. every A in EXPSPACE is polynomial time reducible to B.

Theorem

$EQ_{REX\uparrow} = \{\langle Q, R \rangle \mid Q$ and R are equivalent regular expressions with exponentiation.$\}$ is EXPSAPCE-complete.

Time constructible

Definition

A function $t : \mathbb{N} \to \mathbb{N}$, where $t(n)$ is at least $\mathcal{O}(n \log n)$, is called time constructible if the function that maps the string 1^n to the binary representation of $t(n)$ is computable in time $\mathcal{O}(t(n))$.

In other words, t is space constructible if some $\mathcal{O}(t(n))$ time TM exists that always halts with the binary representation of $t(n)$ on its tape when started on input 1^n.

Examples for time constructible functions

- $n \log n$
- $n\sqrt{n}$
- n^2
- 2^n
- $\ldots$

Time hierarchy theorem

Theorem

For any time constructible function $t : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $\mathcal{O}(t(n))$ space but not decidable in $o(t(n)/\log t(n))$ time.

Proof idea

We construct a TM D which decides a language A in time $\mathcal{O}(t(n))$, whereby A cannot be decided in $o(t(n)/\log t(n))$ time. Here, D takes an input w of the form $\langle M \rangle 10^*$ and simulate M on input w, making sure not to use more than $t(n)$ time. If M halts within that much time, D gives the *opposite* output.

For time complexity, the above simulation introduces a logarithmic factor overhead.

Proof

The following $\mathcal{O}(t(n))$ time algorithm D decides a language A that is not decidable in $o(t(n)/\log t(n))$ time.
D : On input w

1. Let n be the length of w.
2. Compute $t(n)$ using time constructibility and store the value $\ulcorner t(n)/\log t(n) \urcorner$ in a binary counter.Decrement this counter before each step used to carry out stage 4,5. If the counter ever hits 0, *reject*.
3. If w is not of the form $\langle M \rangle 10^*$ for some TM M, *reject*.
4. Simulate M on w.
5. If M accepts, *reject*. If M rejects, *accept*.

Time hierarchy

Corollary

For any two functions $t_1, t_2 : \mathbb{N} \to \mathbb{N}$*, where* $t_1(n)$ *is* $o\left(t_2(n)\right) / \log t_2(n)$ *and* t_2 *is time constructible,* $\mathit{TIME}(t_1(n)) \subset \mathit{TIME}(t_2(n))$*.*

Corollary

For any two real numbers $1 \leq \epsilon_1 < \epsilon_2$*,*

$$\mathit{TIME}(n^{\epsilon_1}) \subset \mathit{TIME}(n^{\epsilon_2}).$$

Corollary

$P \subset \mathit{EXPTIME}$.

Relativization

Oracle model

An oracle for a language A is a device that is capable of reporting whether any string w is a member of A. An oracle Turing machine M^A is a modified Turing machine that has the additional capability of querying an oracle for A. Whenever M^A writes a string on a special oracle tape, it is informed whether that string is a member of A in a single computation step.

Let P^A be the class of languages decidable with a polynomial time oracle Turing machine that uses oracle A. Define the class NP^A similarly.

Examples

- $\mathrm{NP} \subseteq \mathrm{P}^{\mathrm{SAT}}$
- $\mathrm{coNP} \subseteq \mathrm{P}^{\mathrm{SAT}}$
- $\overline{\text{MIN-FORMULA}} \subseteq \mathrm{NP}^{\mathrm{SAT}}$

Limits of the diagonalization method

At its core, the diagonalization method is a simulation of one TM by another: TM M_1 simulates TM M_2 and then behave differently.

Give them the same oracle O, M_1^O can simulate M_2^O just as before.

Thus any theorem proved about TM by using only the diagonalization method will still hold if both machines were given the same oracle.

The same works for ‘P versus NP’.

Limits of the diagonalization method

'P versus NP'

Theorem

1. *An oracle A exists whereby $\mathsf{P}^A \neq \mathsf{NP}^A$.*
2. *An oracle B exists whereby $\mathsf{P}^B = \mathsf{NP}^B$.*

Proof: $\exists B$ ($\mathsf{P}^B = \mathsf{NP}^B$)

Let B be TQBF will do the job.

$$\mathsf{NP}^{\mathsf{TQBF}} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{PSAPCE} \subseteq \mathsf{P}^{\mathsf{TQBF}}$$

Proof: $\exists A$ ($\mathsf{P}^A \neq \mathsf{NP}^A$) (1)

For an oracle A, define language

$$L_A = \{w \mid \exists x \in A\ [\ |x| = |w|\ ]\}.$$

Obviously, for any A, $L_A \in \mathsf{NP}^A$.

We will build an A such that $L_A \notin \mathsf{P}^A$.

Proof: $\exists A$ ($\mathsf{P}^A \neq \mathsf{NP}^A$) (2)

Construct A s.t. $L_A = \{w \mid \exists x \in A \, [\, |x \models |w| \,]\} \notin \mathsf{P}^A$.

Let $M_1, M_2, \ldots$ be a list of *all* polynomial time oracle TMs. Assume M_i runs in time n^i. Language A will be constructed in stages. Each stage determines the status of only a finite number of strings.

Stage i will ensure that M_i^A does not decide A.

Proof: $\exists A$ ($\mathsf{P}^A \neq \mathsf{NP}^A$) (3)

Build A s.t. $L_A = \{w \mid \exists x \in A\ [\ |x| = |w|\ ]\} \notin \mathsf{P}^A$. Let $M_1, M_2, \ldots$ be a list of *all* polynomial time oracle TMs. Assume M_i runs in time n^i. Stage i will ensure that M_i^A does not decide A.

- **Initialize** $A = \emptyset$;
- **Stage** i ($i \geq 1$): So far A is finite and suppose string $\ell \in A$ is of maximal length so far.
  Choose n such that both $n > |\ell|$ and $2^n > n^i$.
  Run M_i on input 1^n, if M_i queries a string y, respond to its oracle queries as follows:
  - y's status has been *determined*: respond consistently;
  - y's status is *undetermined*: respond NO, declare $y \notin A$.

  – If M_i *accepts* 1^n:declare A does not contain any string of length n.
  – If M_i *rejects* 1^n:declare an un-queried string of length n to be in A.
  – Declare any string of length at most n, whose status remains undetermined at this point, is out of A.
  Proceed with **Stage** $i+1$.

In summary

The relativization method tells us that to solve the P versus NP question, we must *analyze* computation, not just simulate them.

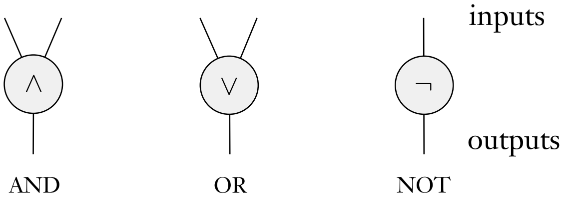
Circuit Complexity

Computers are built from electronic devices wired together in a design called a *digital circuit*. The theoretical counterpart to digital circuit is Boolean circuit.

Boolean circuit

Definition

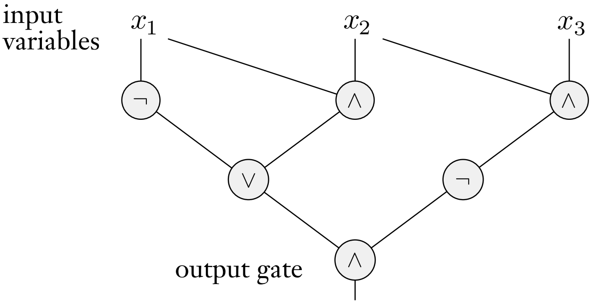
A boolean circuit is a collection of gates and inputs connected by wires. Cycles are not permitted. Gates take three forms: AND gates, OR gates, and NOT gates, as shown schematically in the following figure.



Boolean circuit

The inputs are labeled $x_1, \ldots, x_n$. One of the gates is designated the output gate.

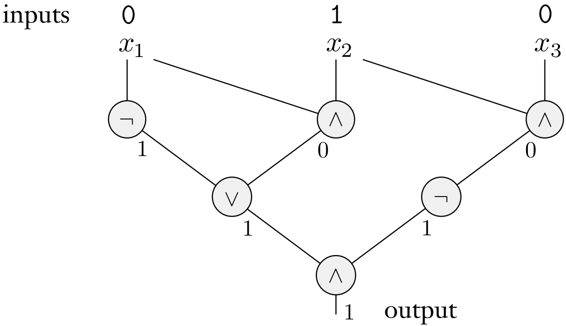
Example



Boolean circuit

A Boolean circuit computes an output value from a setting of the inputs by propagating values along the wires and computing the function associated with the respective gates until the output gate is assigned a value.

Example



Boolean circuit

To a Boolean circuit C with n input variables, we associate a function $f_C : \{0,1\}^n \to \{0,1\}$, where if C outputs b when its inputs $x_1, \ldots, x_n$ are set ti $a_1, \ldots, a_n$, we write $f_C(a_1, \ldots, a_n) = b$. We say that C computes the function f_C.

Example

The $n-$input Parity function *parity*$_n : \{0,1\}^n \to \{0,1\}$ outputs 1 if an odd number of 1s appear in the input variables.

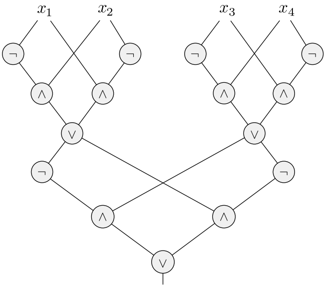


Figure: *parity*$_4$

Circuit family

As any particular circuit can handle only inputs of some fixed length, whereas a language may contain string of different lengths. So instead of using a single circuit to test language membership, we use an entire family of circuits: one for each input length.

Definition

A circuit family C is an infinite list of circuits, $(C_0, C_1, C_2, \ldots)$ where C_n has n input variables. We say that C decides a language A over $\{0, 1\}$ if for every string ω,

$$\omega \in A \text{ iff } C_n(\omega) = 1,$$

where n is the length of ω.

Circuit complexity

The size of a circuit is the number of gates that it contains. Two circuits are equivalent if they have the same input variables and output the same value on every input assignment. A circuit is size minimal if no smaller circuit is equivalent to it. The size complexity of a circuit family $(C_0, C_1, C_2, \ldots)$ is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the size of C_n.

The depth of a circuit is the length of the longest path from an input variable to the output variable.

Similarly, we have depth minimal circuits and circuit families, and the depth complexity of circuit families.

Circuit complexity

Definition

The circuit complexity of a language is the size complexity of a minimal circuit family for that language. The circuit depth complexity of a language is defined similarly, using depth instead of size.

Example

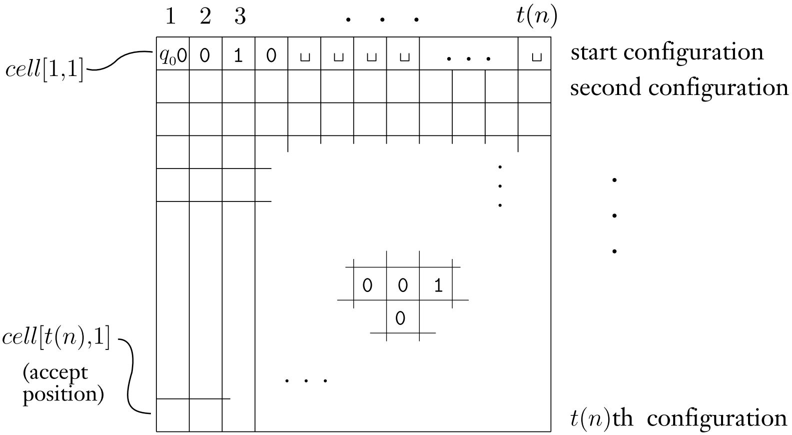
parity$_n$ has circuit complexity $\mathcal{O}(n)$.

Circuit complexity

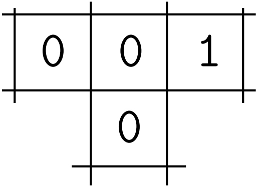
Theorem

Let $t : \mathbb{N} \rightarrow \mathbb{N}$ be a function, where $t(n) \geq n$. If $A \in$ TIME$(t(n))$, then A has circuit complexity $\mathcal{O}\left(t^2(n)\right)$.

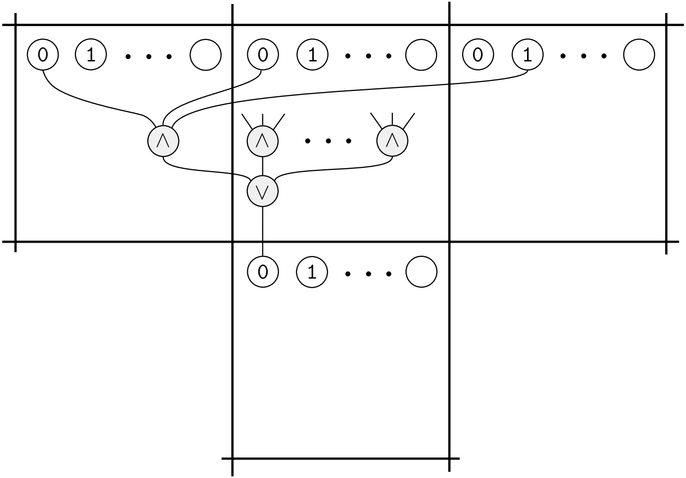
Proof (1)


1 2 3 . . . t(n)
cell[1,1]
q0 0 0 1 0 ⊔ ⊔ ⊔ ⊔ . . . ⊔
start configuration
second configuration
0 0 1
0
cell[t(n),1]
(accept
position)
t(n)th configuration

Proof (2)



Proof (3)



CIRCUIT-SAT

We say that a Boolean circuit is satisfiable if some setting of the inputs causes the circuit to output 1. The circuit-satisfiability problem tests whether a circuit is satisfiable. Let

$$\text{CIRCUIT-SAT} = \{\langle C \rangle \mid C \text{ is a satisfiable Boolean circuit}\}.$$

Theorem

CIRCUIT-SAT is NP-complete.

3SAT

Theorem

3SAT is NP-complete.

Proof.

We give a polynomial time reduction f from CIRCUIT-SAT to 3SAT. □

Proof

Let C be a circuit containing inputs $x_1, \ldots, x_l$ and gates $g_1, \ldots g_m$. We will build a formula ϕ from C. Each of ϕ's variables corresponds to a wire in C. The x_i variables corresponds to the input wires, and the g_i variables correspond to the wires at the gate outputs. We relabel ϕ's variables as $w_1, \ldots, w_{l+m}$.

- $w_j = \text{NOT}(w_i)$: $(\overline{w_i} \rightarrow w_j) \wedge (w_i \rightarrow \overline{w_j})$;
- $w_k = \text{AND}(w_i, w_j)$: $((\overline{w_i} \wedge \overline{w_j}) \rightarrow \overline{w_k}) \wedge$ $((\overline{w_i} \wedge w_j) \rightarrow \overline{w_k}) \wedge ((w_i \wedge \overline{w_j}) \rightarrow \overline{w_k}) \wedge ((w_i \wedge w_j) \rightarrow w_k)$;
- $w_k = \text{OR}(w_i, w_j)$: $((\overline{w_i} \wedge \overline{w_j}) \rightarrow \overline{w_k}) \wedge$ $((\overline{w_i} \wedge w_j) \rightarrow w_k) \wedge ((w_i \wedge \overline{w_j}) \rightarrow w_k) \wedge ((w_i \wedge w_j) \rightarrow w_k)$.