Theory of Computation Part III - Complexity Classes

Ethan Graham

22nd of June 2023

Lecture 7 - P Complexity Class

We established in *part II* that deciders are stronger than recognizers. We would now like to be able to rank these deciders.

Definition: Running Time / Time Complexity

Let M be a TM that halts on all inputs (*it's a decider*). Its running time is the function $t : \mathbb{N} \to \mathbb{N}$ where

$$t(n) = \max_{w \in \Sigma^* \text{ s.t. } |w|=n} \textit{number of steps that } M \textit{ takes on } w$$

The class gives definitions for Big-O and Small-o here, but I'll leave them out because we have seen these definitions a few times

Definition: Time Complexity Class

$$\text{TIME}(t(n)) = \{L \subset \Sigma^* \mid L \textit{ is decided by a TM with running time } O(t(n))\}$$

This next property follows from this definition:

$$\text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq ... \subseteq \textit{TIME}(2^n) \subseteq ...$$

Definition: P Complexity Class

$$P = \bigcup_{k=1}^{\infty} \text{TIME}(n^k)$$

This model extends to all models of computation, whether that be a computer program or a Turing Machine.

Extension of Church-Turing Thesis

This extended version of the Church-Turing thesis states that class P corresponds to all problems that are *realistically* solvable. This is a little controversial since randomisation methods (such as Monte Carlo methods) or quantum computing may have the potential to compute things faster than the turing machine model of computation.

Definition: Verifier

A verifier for a language L is a turing machine M such that for each $x \in \Sigma^*$

$$x \in L \Rightarrow \exists C \ s.t. \ M \ accepts \ \langle x, C \rangle$$
$$x \notin L \Rightarrow \forall C, \ M \ rejects \ \langle x, C \rangle$$

Where C is called a **ceritificate** or **witness**

Definition: Polynomial Time Verifier

This is simply a verifier s.t. on input $\langle x, C \rangle$ runs polynomially in x

Definition: NP Complexity Class

NP is the class of languages that have polynomial-time verifiers. Naturally,

$$P \subseteq NP$$

With potential equality since we don't know if they are equal. Since P has a polynomial time Decider, we can simply make a verifier out of the decider that runs in polynomial time (ignoring the certificate C).

A few NP Problems and their verifiers

SAT and SAT-Verify

Let φ be a *conjunctive normal form* expression (sub-expressions using OR and NOT connected together using AND).

We can easily verify that an assignment for expression φ is satisfiable

$$\text{SAT-verify} = \{\langle \varphi, C \rangle : C \ is \ a \ satisfying \ assignment \ for \ \varphi\}$$

This is clearly in class P since the problem just consist of replacing the literals in φ

We now define the following much more difficult language

$$\text{SAT} = \{\langle \varphi \rangle : \varphi \ is \ satisfiable\}$$

We can solve this with the following *exponential-time* decider:

```
def sat(phi):
    for C in possible_assignments: # loop over all possible assignments for phi
        if sat_verify(phi, C) == True:
            return True
    return False
```

Since this algorithm runs in exponential time, $SAT \notin P$. However, we have that $SAT \in NP$

GI and GI-verify

A graph isomorphism is a bijection $f : V(G_1) \to V(G_2)$ which preserves adjacency, *i.e.*

$$(u, v) \in E(G_1) \Leftrightarrow (f(u), f(v)) \in E(G_2)$$

Two graphs are isomorphic if there exists at least one graph isomorphism between them, *i.e.* we can relabel the vertices of one to get the other.

We define:

$$\text{GI-verify} = \{\langle G_1, G_2, C\rangle | C : V(G_1) \to V(G_2) \textit{ is a graph isomorphism}\}$$

This is indeed $\in P$ since it has a polynomial time decider. We just need to check

$$(u, v) \in E(G_1) \Leftrightarrow (C(u), C(v)) \in E(G_2)$$

Which can be done simply by iterating over all edges in $O(E) = O(V^2)$.

Now consider the more difficult language

$$\text{GI} = \{\langle G_1, G_2\rangle \mid \exists C \textit{ such that } \langle G_1, G_2, C\rangle \in \text{GI-verify}\}$$

This problem is a lot more difficult to solve. Naively we could try all possible isomorphisms in exponential time. The quickest solution that we have is in *quasi polynomial-time* $n^{O(log(n))}$ which still isn't polynomial. We have that $GI \in NP$

The course also defines INDSET. I will leave that out for now but may add later.

Lecture 8 - Non-Deterministic Turing Machines

Definition: Non-deterministic Turing Machines

To extend our definition of Turing Machines to be non-deterministic, it suffices to modify the transition function. Here, we will allow for several transitions from a given state.

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

These aren't computationally efficient at all as we would need an exponential amount of parallel threads in practice.

Definition: Non-deterministic Decider

A non-deterministic decider for L is an NTM N such that for each $x \in \Sigma^*$, every computation of N on x halts. Furthermore, if $x \in L$, then a non-zero amount of threads of N will accept and it $x \notin L$ then no threads will accept (*all threads reject*).

Definition: Polynomial Time Non-deterministic Turing Machine

This is when N's longest running time on x is polynomial in $|x|$

Theorem:

Let $L \in \Sigma^*$ be an arbitrary language.

L has a non-deterministic polynomial time decider $\Leftrightarrow L$ has a poly-time verifier

Proof idea $\Rightarrow$

By hypothesis there exists an NTM N that decides L. We construct a verifier V for L using N. On input $\langle x, C\rangle$ simulate $N(x)$ with non-deterministic choices given by C. We know that there exists a path with polynomial-in-$|x|$ length in our decision tree (by definition of NTM). Thus we know that this path can be encoded by a certificate with polynomial length.

Proof idea $\Leftarrow$

We know by hypothesis that L has a polynomial-time verifier V. We construct a nondeterministic polynomial time decider using V. We do this by trying all possible certificates in parallel (which we are allowed to do with an NTM). If there exists a certificate that works, we will find it in polynomial time.

Definition: $NTIME$ Complexity Class

$$\text{NTIME}(t(n)) = \{L \mid L \text{ has a nondeterministic } O(t(n)) \text{ time decider}\}$$

Definition: NP Class

$$\text{NP} = \bigcup_{k=1}^{\infty} \text{NTIME}(n^k)$$

Lecture 8 - Polynomial-time Reductions

Definition: Poly-time Computable Function

$f : \Sigma^* \to \Sigma^*$ such that $\exists M$ a turing machine that on every input w halts with just $f(w)$ on its tape.

Definition: Poly-time Reduction (***Poly-time mapping reducible***)

$A \leq_p B$ if $\exists f$ a poly-time computable function s.t. $w \in A \Leftrightarrow f(w) \in B$

Theorem:

If $A \leq_p B$ and $B \in P$, then $A \in P$

proof idea: Compute $y = f(w)$ on $w \in A$ in $O(n^q)$, and then run B's decider on y in $O(n^p)$

If A reduces to B and B is easy, then A is easy too.

Corollary

If $A \leq_p B$ and $A \notin P$, then $B \notin P$

Transitivity

If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$

Definition: NP-Hardness

A language L is NP-hard if $L \in NP$ and if every language $L' \in NP$ is such that

$$L' \leq_p L$$

Definition: NP-Complete

Language L is NP-complete if $L \in NP$ and L is NP-hard.

Observation

If any NP-complete happened to have a poly-time decider, then this would imply that every NP problem has a poly-time decider. Thus implying $P = NP$. NP-complete problems are the hardest NP problems.

Cook-Levin Theorem

SAT is NP-complete

In practice to show that a language L is NP-complete...

- **NP membership:** $L \in NP$, we find a poly-time verifier
- **NP-hardness:** $H \leq_p L$ for some known NP-complete language H

Propositions: (*inplying NP-completeness of left-hand side*)

$$\text{SAT} \leq_p \text{INDSET}$$

$$\text{SAT} \leq_p k\text{SAT}$$

Theorems: (*implying NP-completeness of left-hand side*)

$$\text{INDSET} \leq_p \text{CLIQUE}$$

$$\text{INDSET} \leq_p \text{VERTEX COVER}$$

$$\text{VERTEX COVER} \leq_p \text{SET COVER}$$

$$\text{SAT} \leq_p \text{PERFECT-3-MATCHING}$$

$$\text{PERFECT-3-MATCHING} \leq_p \text{SUBSET-SUM}$$

I'm going to omit the proofs here because it would take ages to write them and the idea of this document is for it to be *shorter* than the other one. I suggest you read the equivalent chapter in Joachim Favre's document