31251 – Data Structures and Algorithms

Week 11, Autumn 2020

Xianzhi Wang

Everyone's gonna love this:

- Computational Complexity
- P
- NP
- NP-hardness and NP-completeness

Algorithmic to Computational Complexity

So far we've seen a variety of algorithms for problems.

- We can compare them via their asymptotic running time.

What about the problems themselves?

- An algorithm is but may not be the best solution.
- How can we get some idea of whether we're doing well with our algorithms?

We need some formalisation to talk about the complexity of problems.

Formalising Computational Problems

A (basic) computational problem consists of two parts:

- The input.
- The ouput.

SORT

Input: An array A of length n.

Output: An sorted array A' that contains the same elements as A and satisfies $\forall 1 \leq i \leq j \leq n$, $A'[i] \leq A'[j]$.

Formalising Computational Problems

The same computational problem can come in several varieties:

- `Optimisation` - we want to find the best solution according to some optimisation criterion.
- `Search` - we want to find any solution that solves the problem.
- `Decision` - we just want to know *if* there is a solution.

We're interested in *decision* problems (what Turing Machines do).

The decision version of `sorting` looks like:

SORT
Input: An array A of length n, two integers k, p.
Question: Is element $A[k]$ at position p when A is put into ascending order?

Are these always equivalent to the others?

The Complexity of a Problem

Now that we have at least some notion of a formal problem, we define what we mean by its complexity:

Definition

The complexity of a problem is the complexity of the (asymptotically) most efficient algorithm that correctly solves the problem.

- Most of the time, when we say "most efficient", we mean fastest.
- Any algorithm that solves the problem at least gives an upper bound – but we also need a lower bound to be precise.

Algorithms so far

Using Turing Machine, we can say things like:

- Sort has complexity $\Theta(n \log n)$.
- Depth First Search has complexity $\mathcal{O}(|V| + |E|)$.
- Minimum Spanning Tree has complexity $\mathcal{O}(|E| \cdot \log |V|)$.
- Shortest Path(s) has complexity $\mathcal{O}(|E| + |V| \log |V|)$.

These are fast, practical algorithms - we would like to group these together...

Definition (Polynomial-time or "P")

Given a decision problem Π, we say that Π is polynomial-time solvable, or $\Pi \in \mathbf{P}$, if and only if there exists a Turing Machine ($\approx$ algorithm) $\mathcal{A}$ such that for each instance I of Π the Turing Machine $\mathcal{A}$ halts and correctly answers $\textsc{Yes}$ or $\textsc{No}$ in time bounded by $\mathcal{O}(|I|^c)$ for a fixed $c \in \mathbb{N}$.

Alternatively, if we think of Π as a language, $\mathcal{A}$ decides $I \in \Pi$ in polynomial time.

Cobham Edmonds Thesis

Computational problems can be feasibly computed on some computational device only if they can be computed in polynomial time.

- This is the basic rule-of-thumb that standard complexity theory works with.
- Why is this a reasonable thing to say?
  - A finite number of Yes/No decision problems.
  - Each solved in polynomial time.

$$c_1 O(|I|^{c_2}) = O(|I|^{c_2})$$

Excursion 1 - Measuring Things

- Problem size ($|I|$): the length of encoding of an instance over the input alphabet Σ of Turing Machine.
- We can reasonably assume $\Sigma = \{0, 1\}$, so the encoding is in binary.
- But what does it mean when we start using $|V|$ or $|E|$, or other measures for the complexity?
  - We can encode a graph in at most $\mathcal{O}(|V|^2 \log |V|)$ bits.
    - Why? Each node $\log |V|$; number of edges related to the node $|V|$; number of nodes $|V|$.
  - So $|V|$ is within a polynomial factor of $|I|$, so it's "okay" to use it as a proxy of $|I|$.

Is Everything in **P**?

- So 'everything' we've seen so far is in **P**. Is *everything* in **P**?
- Of course not! We can easily make up problems that aren't, some things aren't even decidable!
- What about things that we might want to solve reasonably, but can't quite seem to get a polynomial-time algorithm?

Excursion 2 - Non-determinism

- Turing Machines can be non-deterministic.
- What does this mean?
  - A non-deterministic machine can guess cleverly to solve a problem (but we still have to check the solution).

P again

- The class **P** is actually the class of problems that can be solved in polynomial time by a `deterministic` Turing Machine.
- What happens if we switch deterministic for non-deterministic?
- We get **NP**!

Non-deterministic Polynomial Time, or **NP**

Definition (**NP**)

Given a decision problem Π, we say that Π is non-deterministically polynomial-time solvable, or $\Pi \in$ **NP**, if and only if there exists a *non-deterministic* Turing Machine ($\approx$ algorithm) $\mathcal{A}$ such that for each instance I of Π the Turing Machine $\mathcal{A}$ halts and correctly answers $\textsc{Yes}$ or $\textsc{No}$ in time bounded by $\mathcal{O}(|I|^c)$ for a fixed $c \in \mathbb{N}$.

- Obviously **P** $\subseteq$ **NP** - just don't do any guessing.
- Do we get any more power from non-determinism (i.e., is **P** $\subset$ **NP**?)
- We'll come to that later.

An Alternate Characterisation of **NP**

NP can also be thought of as the class of problems that we can verify in polynomial-time:

Definition (Polynomial-time verifiable.)

Given a decision problem Π, we say that Π is polynomial-time verifiable, if and only if there exists a deterministic Turing Machine ($\equiv$ algorithm) $\mathcal{A}$ such that for each instance I of Π, with a witness W that I is a YES instance where $|W| \leq |I|^k$ for some $k \in \mathbb{N}$, the Turing Machine $\mathcal{A}$ halts and correctly answers whether the witness is correct in time bounded by $\mathcal{O}(|I|^c)$ for a fixed $c \in \mathbb{N}$.

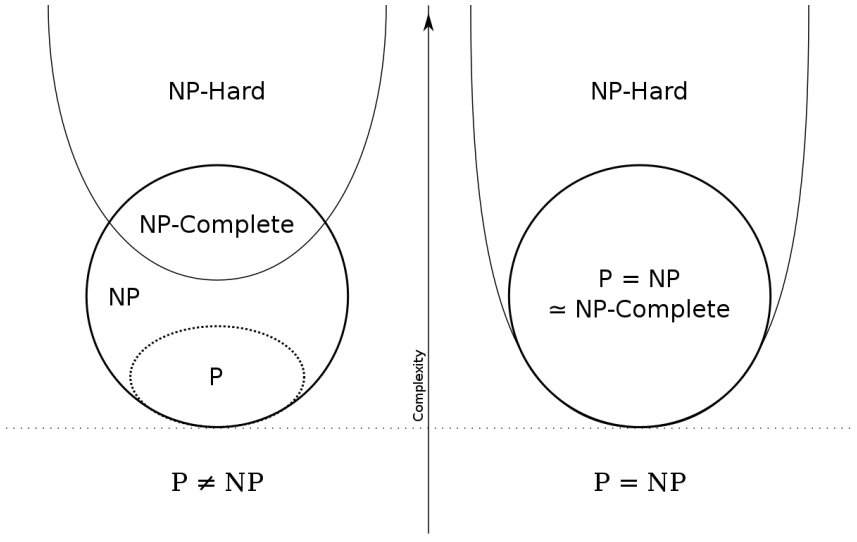
i.e., given a solution, we can check if it is correct quickly. It turns out that this is the same as **NP**

Putting Things in **NP**

For example,

- SORT is **NP** - check the final array in linear time.
- MINIMUM SPANNING TREE (MST) is **NP** - add up the weights of the solution and check.

But there are things that are in **NP** for which we don't know any polynomial-time algorithm.



NP Seems Bigger

It seems **NP** contains a 'lot more' than **P** - many problems that have no obvious **P**-time algorithm are in **NP**.

- No one knows whether **P** = **NP** or not though.
- The general appraisal is that **P** $\neq$ **NP**, but there is notable disagreement.

If we can't implement a non-deterministic algorithm, what is the point of **NP**?

- **NP** gives us a way to show that a problem has no polynomial-time algorithm unless **P** = **NP**.

Before we can say how hard things are, we need to be able to compare them. We do this via *polynomial-time many-one mapping reductions* (or ptime reductions, or Karp reductions):

Definition

Given two decision problems A and B, we say A is (ptime many-one) reducible to B (denoted by $A \leq_m B$) *iff* there exists a deterministic algorithm that maps an instance I_A to an instance I_B such that I_A is a YES-instance *iff* I_B is a YES-instance, where the algorithm runs in time $\mathcal{O}(|I_A|)$.

Suppose A is reducible to B. Then,

- B is "at least as hard as" A, up to some polynomial factor.
- if we can solve B in **P**-time, then we can solve A in **P**-time.

NP-hardness and **NP**-completeness

A problem Π' is **NP**-hard if every NP problem in NP can be reducible to Π'.

$$\forall \Pi' \in \mathbf{NP}, \exists \leq_m, \text{ such that } \Pi' \leq_m \Pi.$$

- If A is **NP**-hard and $A \leq_m B$, then B is also **NP**-hard.
- If Π is also in **NP**, then Π is **NP**-complete.

So the **NP**-complete problems are the 'hardest' problems in **NP**.

- If there's a polynomial time algorithm for any of them, then $\mathbf{P} = \mathbf{NP}$.

The Cook-Levin Theorem, or, SAT is **NP**-complete

The first **NP**-complete problem is Satisfiability (SAT):

> SAT
> *Instance: A set of variables V and a set of clauses C over the literals of V (i.e. a CNF formula).*
> *Question: Is there an assignment of* True *and* False *to V such that the entire formula evaluates to* True*?*

For example,

$$\neg B \wedge \neg C(A \vee C) \wedge (B \vee C)(A \vee C) \wedge (B \vee C)$$

$$A \wedge (B \vee D) \wedge (B \vee E).A \wedge (B \vee D) \wedge (B \vee E).$$

Theorem (Cook-Levin Theorem)

SAT *is* **NP***-complete.*

And from there...

- Then from SAT, we can reduce to thousands of other problems.

- Meaning that they are all **NP**-complete:
  - Vertex Cover.
  - Hitting Set.
  - Dominating Set.
  - Independent Set.
  - Clique.
  - Travelling Salesman Problem.
  - Graph Colouring.

What do we do with all this?

- So now we know that some things are **NP**-complete. What next?
- This gives strong evidence that looking for a polynomial-time algorithm is a waste of time.
- Even worse, lots of problems are **NP**-complete.
  - In comparison, there are very limited problems in **P**.
- We have to develop methods for coping with intractability.

Methods for Coping with Intractability

- Approximation — often, we're happy with sub-optimal solutions.
- Randomisation — maybe we get lucky most of the time.
- Heuristics and metaheuristic — use what seems to work in practice.
- Relaxation — sometimes exponential is better than polynomial.
- Build a different type of computer — quantum computing?.