31251 – Data Structures and Algorithms

Week 11

Luke Mathieson

Everyone's gonna love this:

- Moving from Algorithmic Complexity to Computational Complexity.
- **P**.
- **NP**.
- **NP**-hardness and **NP**-completeness.

From Algorithmics to 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 a solution for a problem, but it may not be the best one.
- How can we get some idea of whether we're doing well with our algorithms?

We can talk about the complexity of *problems* too, but we need some formalisations...

Formalising Computational Problems

- A (basic) computational problem consists of two parts:
  - The input.
  - The ouput.
- Both should be specified *precisely*.

For example, we can express sorting an array as:

> SORT
> *Input: An array A of length n.*
> *Output: An array A' with the same elements as A where $\forall 1 \leq i \leq j \leq n$ we have $A'[i] \leq A'[j]$.*

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.

For a variety of reasons, we're going to be most interested in *decision* problems. (Are these always equivalent to the others?)

So the decision version of sorting would look 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?*

For those who like formal languages (ha!) a decision problem can be equivalently thought of as the language containing the strings encoding the YES instances.

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.

"Most Efficient" on what?

To make this definition well-formed, we need to specify how we're going to measure efficiency.

- What is the machine model that we're going to use?
- RAM vs. Turing Machine? (How many tapes?)
- What's a "unitary" operation?

We're going to take Turing Machines as our basic model. (Why?)

Algorithms so far

So using this model, 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.

The Class **P** & the Cobham Edmonds Thesis

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? (*Is it* reasonable?)

Excursion 1 - Measuring Things

- Notice that in the definition of **P**, we talk about the size of the instance $|I|$, what does this mean?
- The formal definition is that $|I|$ is the length of a reasonable encoding of I over some alphabet of symbols Σ (the input alphabet of the Turing Machine).
- We can reasonably assume that $\Sigma = \{0, 1\}$, so the encoding is in binary. (Why?)
- But what does it mean when we start using $|V|$ or $|E|$, or other measures for the complexity?
  - These measures are called *proxy* measures.
  - Note given a graph, we can encode it in at most $\mathcal{O}(|V|^2 \log |V|)$ bits. (Why?)
  - So $|V|$ is within a polynomial factor of $|I|$, so it's "okay" to use it as a proxy.

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? Can we say anything here?

Excursion 2 - Non-determinism

- You might remember a brief mention of non-determinism several weeks ago.
- Turing Machines can be non-deterministic too.
- What does this mean?
  - Non-determinism is a piece of theoretical "magic".
  - A non-deterministic machine can guess cleverly to solve a problem (but we still have to check the solution).
  - Slightly more formally, a non-deterministic machine, on a given input, when presented with more than one available transition, will follow the path that leads to an accepting configuration.
  - In other words, if there's a way it can say YES, it will - this doesn't guarantee it's correct though!.

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 $\textsc{Yes}$ 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 $c \in \mathbb{N}$.

That is, given a solution we can check if it's right quickly. It turns out that this is the same as **NP**
Why are these two definitions equivalent?

A *third* definition: **NP** is the class characterised by existential second-order logic (Fagin's Theorem).

Putting Things in **NP**

So, for example,

- SORT is in **NP** - check the final array in linear time.
- MST is in **NP** - add up the weights of the solution and check (note that we're always talking about the decision version).

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

NP Seems Bigger

It seems like **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 the basis of a way to show that a problem has no polynomial-time algorithm unless **P** = **NP**.

Before we can say how hard things are (when we don't have an algorithm), we need to be able to compare them. We do this via *polynomial-time many-one mapping reductions*, AKA ptime reductions, or Karp reductions:

Definition

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

- So if we could solve B in **P**-time, we could solve A in **P**-time (start with I_A, map to I_B, solve I_B).
- If B is in **P**, and $A \leq_m B$, then A must also be in **P**.
- Note that as the running time is polynomial, $|I_B| \leq |I_A|^c$.
- So then B is "at least as hard as" A, up to some polynomial factor.

- So reductions give us an ordering on decision problems.
- So what happens when start looking for reductions between problems in **NP**?
- It turns out that there's a group of *really hard* problems that cluster together at the 'end' of **NP**.
- These are the **NP**-complete problems.

NP-hardness and **NP**-completeness

1. A problem Π is **NP**-hard if for every $\Pi' \in$ **NP** there exists a Karp reduction such that $\Pi' \leq_m \Pi$.
2. If Π is also in **NP** (note we did not make this assumption above), then Π is **NP**-complete.

If a problem A is **NP**-hard and $A \leq_m B$, then B is also $\mathbf{NP} - \mathit{hard}$. (Why?)
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*?*

Theorem (Cook-Levin Theorem)

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

And from there...

- Then from SAT, we can reduce to thousands of other problems.
- Some you may see or hear of:
  - 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.
  - An efficient algorithm for *any* **NP**-complete problem would give an algorithm for all of them, and no-one has found one yet.
- Even worse, it turns out that lots of problems are **NP**-complete.
  - Sometimes it seems like there's only about 6 problems in **P**.
- So we have to develop methods for coping with intractability.

Methods for Coping with Intractability

- Approximate - maybe we're happy with not-quite-the-best solutions.
- Randomize - maybe we can get lucky most of the time.
- Develop heuristics and metaheuristics - use what seems to work in practice.
- Relax what it means to be tractable - sometimes exponential is better than polynomial.
- Build a different type of computer - will at least get you research grants.