CS 3510 Algorithms 10/30/2023

Lecture 15: NP-completeness

Lecturer: Abrahim Ladha *Scribe(s): Himanshu Goyal*

1 Introduction

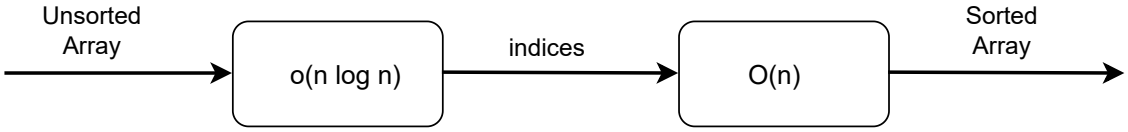
This lecture is like my sales pitch to the field of complexity theory. An algorithm is a constructive solution to a problem. For some problem, an algorithm is an upper bound. A problem has complexity. An algorithm has run time. Given a problem, there may exist any number of upper and lower bounds. Those lower bounds can involve deep mathematical relationships between problems (or class of problems).

Why are we studying complexity in an algorithm course. Sometimes, those deep mathematical relationships are themselves, just algorithms!

Example: Let us consider the following problem, discusses its complexity (upper and lower bounds)

- Input: Unsorted list of integers
- Output: the list of indices as if it was sorted

Here is one potential solution: We have an obvious upper bound on sorting. In $O(n \log n)$ time, sort the array, keeping track of indices and then return indices. Can we do faster? Is there some $o(n \log n)$ algorithm? Suppose to the contrary, there exists an $o(n \log n)$ algorithm, (suppose its $O(n)$ even). Then we given an $o(n \log n)$ sorting algorithm



Using the indices, you can sort the array in linear time. So this gives a $o(n \log n) + O(n) = o(n \log n)$ sorting algorithm, *a contradiction*. Therefore, the complexity of this problem is $\Omega(n \log n)$. This way of solving problems is commonly referred as reduction and is very popular in complexity theory.

Its all about this kind of thinking. Do not think about solving the problem. *Do not solve the problem*. Instead think about how this problem relates to a known problem. All algorithms in this unit will only be reductions, i.e. an algorithm to convert one problem instance into instance of another problem. This is the true moral of this unit.

2 Background

A decision problem is a problem which can be phrased as a *Yes/No* question. For instance, PATH $= \{\langle G, s, t\rangle \mid$ there $\exists$ a path from s to t in $G\}$. Here, it is not asking for the path. It is asking only, if there exits a path. A search search problem is one which would actually output path. Every search problem obviously has a corresponding decision problem. *In our formalisation, we only consider decision problems.*

Define P to be the class of decision problems decidable in polynomial time. If $A \in P$, $\exists$ an algorithm on input x, to correctly say yes if $x \in A$ and no if $x \notin A$, $\exists$ some $k > 0$ such that this algorithm runs in $O(n^k)$ time.

Reason 1: The class P captures our intuitive notion of what it means for an algorithm be be "efficient". If there exists a polynomial time algorithm for a problem, either the problem is ridiculously trivial, or we have some deep mathematical characterization, an understanding of the problem. Think about how all our fast algorithms did things based on properties of the problem they were solving. Most problems have terrible brute force algorithms, most problems seems to be in EXP.

Reason 2: P is closed under operations which do not violate our intuition about efficiency. The sum, product, composition of polynomials are all polynomials. If f, g are polynomial time algorithms they are "efficient". Running f, then running g should also maintain this intuition about "efficiency". This algorithm has run time $f + g$, also a polynomial.

Reason 3: Although by the time hierarchy theorems, there do exist problems of $\Omega(n^{99})$, none of these appear natural or useful. The highest polynomial time algorithm I have personally seen is for the LLL[1] algorithm, what it solves is unimportant to us, but its existence was surprising. It solves a very hard problem in $O(n^8)$ time, thought to be super polynomial. The original (n^8) time algorithm is actually unusable in practice and is purely a theoretical result. However, now since its poly time, it can be handed off to the engineers. There are many hardcore pruning algorithms which have unclear analysis, but make the LLL algorithm practically instant on any input I could test. Recall why big-O hides those addition and multiplication constants. Its not our job, its someone elses. A poly time algorithm for a problem implies non-trivial intuition about the problem statement. If you can be offered this little insight, you can perhaps expand it to take more. This is why its hard to find practical high degree poly time algorithms. If it gets too high, then its not poly time. Many of the algorithms we studied ran in times $O(n), O(n \log n), O(n^2)$ and not much more.

A final reason is that most models of computation can simulate each other up to a polynomial degree. Within P, there should be caution. Our concern for this will be around and just outside of P, so we will continue to only use and care about the word-RAM model.

2.1 Examples within P

- PATH $= \{\langle G, s, t\rangle \mid$ there $\exists$ a path form s to t in G$\}$. BFS/DFS can solve this problem in linear time.

[1] `https://en.wikipedia.org/wiki/LenstraLenstraLovsz_lattice_basis_reduction_algorithm`

- $RELPRIME \in \mathsf{P}$ i.e. $RELPRIME = \{\langle x, y\rangle \mid gcd(x, y) = 1\}$ since GCD takes polynomial time (cubic in fact)

3 Complexity Classes

NP = Non-deterministic polynomial time. With out getting too muddy into the definition of non-determinism. Lets give an equivalent deterministic definition.

A$\in$ NP if solutions to problems in A are verifiable in polynomial time. A$\in$ P $\implies$ $\exists$ a polynomial time algorithm. The algorithm may take on $\{\langle G, s, t\rangle$ and determines $\exists$ a path or not. A $\in$ NP $\implies$ $\exists$ a poly time verifier V. V takes on $\{\langle G, s, t\rangle$ the problem instance, and also a *witness/certificate/solution* say $\langle v_1, v_2....v_k\rangle$ and determines *yes/no*, if $\langle v_1, v_2....v_k\rangle$ is a path in G from $v_1 = s$ to $v_k = t$.

Similarly COMPOSITES $\in$ NP where COMPOSITES $= \{n \mid n$ is not prime$\}$, why? the problem instance would be some $< n >$, the witness would be the factors $p_1, ..., p_k$. Now, we can compute $N \stackrel{?}{=} p_1 \cdot p_2.... \cdot p_k$ in polytime.

Let us prove one direction of a famously hard problem. Let A$\in$ P. We show $\implies$ A$\in$ NP. Since this is $\forall A \in$ P, then we conclude P $\subseteq$ NP.

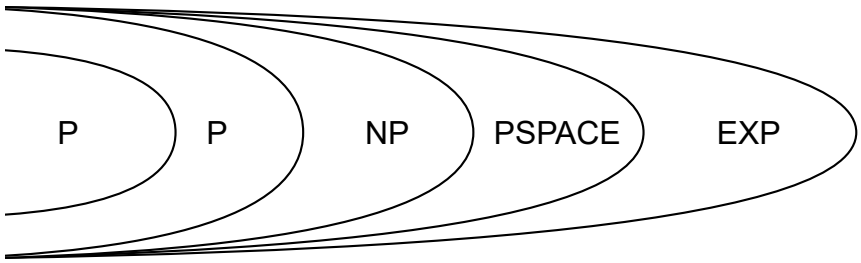
If A $\in$ P, $\exists$ a poly time algorithm f to decide A. We give a poly time verifier V. V on input (Problem p, witness = "") simply returns $f(p)$. Essentially your verifier just solves the problem! Since f is polytime, so is V obviously. Another way to think: a witness can only speed up computation. Having the answers can only help, so P $\subseteq$ NP.

We believe its strict but cannot prove it. We have some, many, thousands of problems in NP, we do not believe to be in P. But we can not prove it. The history of complexity theory is a history of failure.

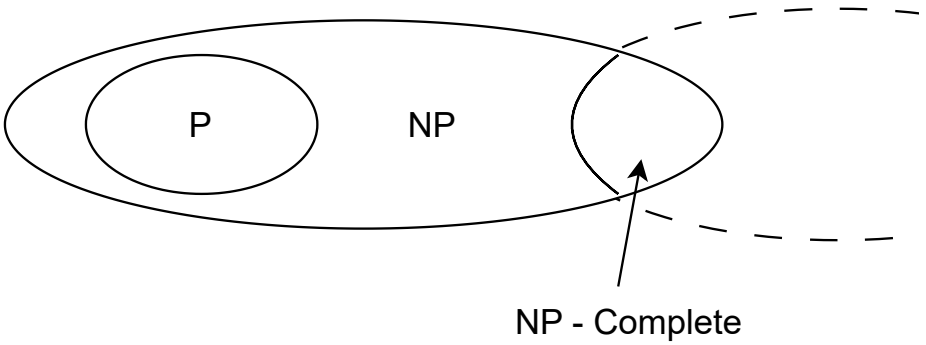
Let EXP = problems solvable in exponential time. We know, and we can prove P $\subsetneq$ EXP. There are problems in EXP provably not in P. We also know we can brute to solve the problems in NP, so we see P $\subseteq$ NP $\subseteq$ EXP. Since P $\subsetneq$ EXP, one of the P $\subsetneq$ NP, NP $\subsetneq$ EXP must be true. So if you could show NP = EXP, this would imply, P $\neq$ NP for free. P $\stackrel{?}{=}$ NP is then equivalent to question " Are there any decision problem requiring exponential time to solve which also require super polynomial time to verify?"

Let L $= SPACE(\log n)$ and PSPACE $= \bigcup_{k=0}^{INF} SPACE(n^k)$ polynomial and logarithmic space. Buy the space hierarchy theorem, we know L $\subsetneq$ PSPACE but we similarly have the chain L $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE. If you could prove L = P and NP = PSPACE then you would get P $\neq$ NP for free! If you could show P = PSPACE , you would get P = NP for free.

If P $\neq$ NP, our world looks like

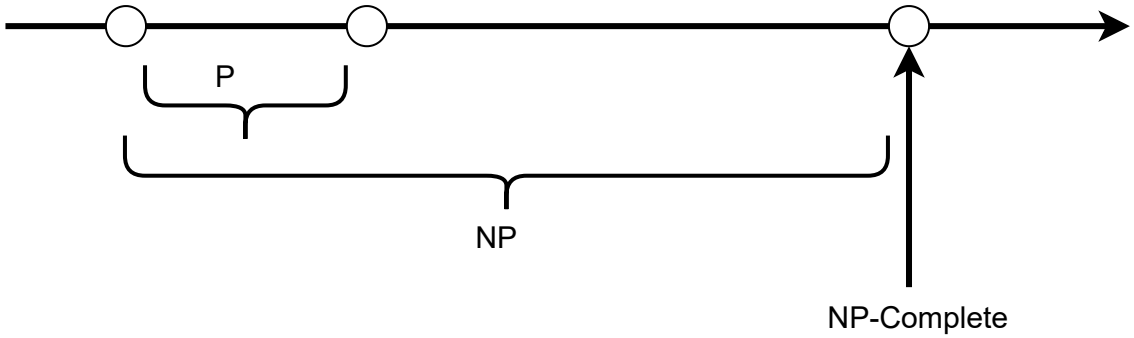


From now on, we won't care about outside NP. Let us zoom in.



4 NP-Completeness

NP-Complete problems are the hardest problems in NP.

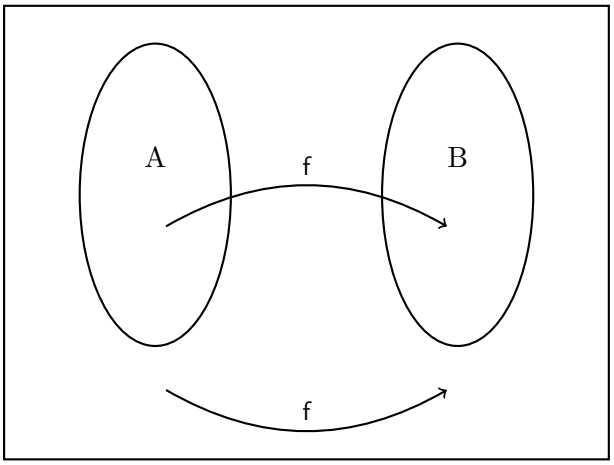


Why do we care about NP-Completeness. It depends who you are. To me, it means you can give up working on a problem, or apply NP-complete specific technqiues, such as SMT solvers.

How do we prove a problem is NP-Complete: First we define a poly time reduction for two problems, A and B, we say $A \leq_p B$. (A is poly time reducible to B) if exists f which is computable in polynomial time such that

$$x \in A \iff f(x) \in B$$

Note this implies $x \notin A \iff f(x) \notin B$. f maps right answers to right answers and wrong answers to wrong answers.



Intuitively, you should think $A \leq_p B$ means *"B is harder than A"*. B is an upper bound for A or A is an lower bound for B.

If $A \leq_p B$ and B $\in$ P $\implies$ $A \in P$. If $A \leq_p B$ then $\exists$ f such that x $\in$ A $\iff$ $f(x) \in B$, with f computable in poly time. IF B $\in$ P , the $\exists$ algorithm for B(x) which runs in poly time, we give a poly time algo for A to prove A $\in$ P.

```
algo for A(x):
compute f(x)
if f(x) in B
    return true
else
    return false
```

Here we accept x if the algorithm for B accepts f(x). We reject x if the algorithm for B rejects f(x). So this decides A by looking at B. Since f, algo for B are computable in poly time, so this decides A in poly time.

To prove B is NP-Complete you show:

- B $\in$ NP
- B is NP-hard: If A is some NP-Complete problem, prove $A \leq_p B$ by giving a poly time reduction form A to B.

Cook and Levin[2] independently proved $\forall A \in \mathsf{NP}$ that $A \leq_p SAT$. We will take about SAT next time, but for us all that matters is that there exists an NP-Complete Problem. To show some B is NP-Complete, pick a candidate NP-Complete problem (like SAT) and show $SAT \leq_p B$. By transitivity, this shows $\forall A \in \mathsf{NP}$, $A \leq_p SAT \leq_p B \implies \forall A \in \mathsf{NP}$, $A \leq_p B$ or that "B is harder than anything in NP". Combining this with $B \in \mathsf{NP}$, you get "B is the hardest problem in NP" truely one of many.

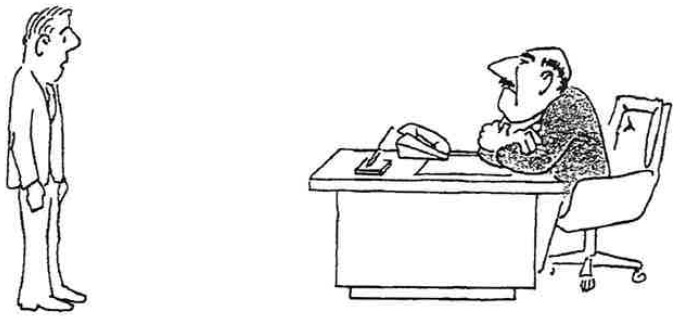
We will have a vast collection of NP-Complete problems by our polynomial time reductions, they all are as hard as each other up to a polynomial factor. A fast algorithm for one is a fast algorithm for all.

Prove: SAT (or any other NP-complete problem) $\in \mathsf{P} \implies \mathsf{NP} = \mathsf{P}$. We already know $\mathsf{P} \subseteq \mathsf{NP}$, so we prove just $\mathsf{NP} \subseteq \mathsf{P}$

Proof: Let A $\in$ NP, we know A$\leq_p$ SAT. We proved A$\leq_p$B and B $\in$ P $\implies$ A $\in$ P. So A $\in$ P $\implies$ NP $\subseteq$ P $\implies$ P = NP.

[2] https://en.wikipedia.org/wiki/Cook–Levin_theorem

:ifications, and the bandersnatch department is already 13 components ind schedule. You certainly don't want to return to his office and re- :



"I can't find an efficient algorithm, I guess I'm just too dumb."

To avoid serious damage to your position within the company, it would much better if you could prove that the bandersnatch problem is *in- ently* intractable, that no algorithm could possibly solve it quickly. You could stride confidently into the boss's office and proclaim:

"I can't find an efficient algorithm, because no such algorithm is possible!"

Unfortunately, proving inherent intractability can be just as hard as ing efficient algorithms. Even the best theoreticians have been stymied heir attempts to obtain such proofs for commonly encountered hard blems. However, having read this book, you have discovered something

almost as good. The theory of NP-completeness provides many straightforward techniques for proving that a given problem is "just as hard" as a large number of other problems that are widely recognized as being difficult and that have been confounding the experts for years. Armed with these techniques, you might be able to prove that the bandersnatch problem is NP-complete and, hence, that it is equivalent to all these other hard problems. Then you could march into your boss's office and announce:

"I can't find an efficient algorithm, but neither can all these famous people."

At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.

Of course, our own bosses would frown upon our writing this book if its sole purpose was to protect the jobs of algorithm designers. Indeed, discovering that a problem is NP-complete is usually just the beginning of work on that problem. The needs of the bandersnatch department won't disappear overnight simply because their problem is known to be NP-complete. However, the knowledge that it is NP-complete does provide valuable information about what lines of approach have the potential of being most productive. Certainly the search for an efficient, exact algorithm should be accorded low priority. It is now more appropriate to concentrate on other, less ambitious, approaches. For example, you might look for efficient algorithms that solve various special cases of the general problem. You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time. Or you might even relax the problem somewhat, looking for a fast algorithm that merely finds designs that