CS 3510 Algorithms 3/14/2024

Complexity Classes

Lecturer: Jan van den Brand *Scribe(s): Ankith Thalanki*

Complexity Theory, or Computational Complexity Theory, is the study of problems that can be solved by computers, and the study of the rough time complexity used by the corresponding solution/algorithm. Complexity Classes are groups of problems where problems in one class are roughly equally difficult, or related (like via reductions). In our lecture today we covered 4 main complexity classes, P, NP, NP-Hard and NP-Complete.

1 P

The first complexity class we covered was **P**, a complexity class that contains all problems that we know can be solved in polynomial time with respect to the size of an input. For example, a problem that can be solved in $O(n^c)$ would be polynomial, while a problem that can only be solved in $O(2^n)$ would have an exponential runtime, and therefore wouldn't be in **P**. Some examples of problems in **P** would be sorting, minimum, Bellman-Ford, BFS, etc. Most of the algorithms we have covered before in this class fall into **P**. Keep in mind that for a problem to be in **P** it must be polynomial with respect to the size of the input, not input itself. for example, factoring an integer x can easily be done in $O(x)$, however the size x is actually $log_x(n)$, so the actual runtime is $O(2^x)$. This explains why current factoring algorithms are not in **P**.

2 NP

The **NP** complexity class contains problems that can be solved in a non-deterministic polynomial time. This simply means that a solution to a problem in **NP** can be verified in polynomial time. It doesn't however mean that all problems in **NP** cannot be solved in polynomial time. In fact, $\mathbf{P} \subset \mathbf{NP}$, meaning that every problem that can be solved in polynomial time can have such a solution verified in polynomial time, which makes sense. One of the biggest questions in complexity theory and math in general is if $\mathbf{P} = \mathbf{NP}$, which if proven true would mean that every problem whose solution we can verify in polynomial time we can solve in polynomial time. But how would we go about proving every of the infinite problems in **NP** are polynomial? The key lies in **NP-Complete** problems which are discussed in deeper measure further below. Some examples of **NP** problems include sorting (which is also in **P**, finding the Hamiltionian path in a graph, and the traveling salesman problem.

3 NP-Hard

NP-Hard problems is the complexity class that contains all problems that are "harder" than any problem in **NP**. What do we mean by some problem A being harder than another

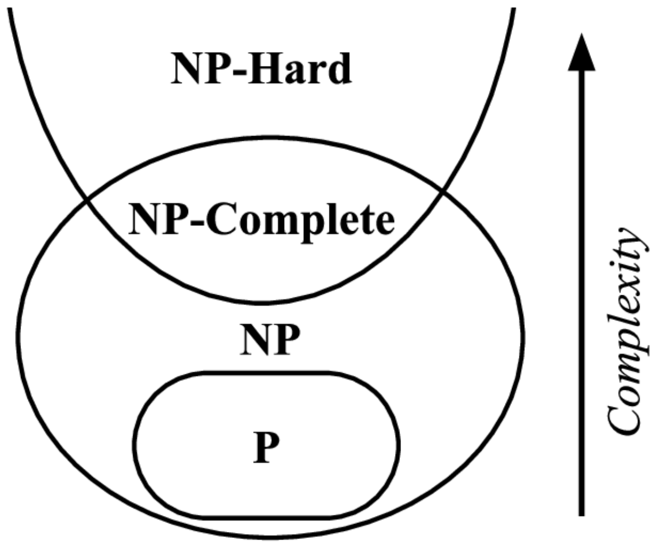

Figure 1: Diagram showcasing the relations between all the complexity classes covered. As you can see, $\mathbf{P} \subset \mathbf{NP}$
, and **NP-Complete** is the intersection of the **NP-Hard** and **NP** complexity classes. You can also see that the **NP-Hard** complexity class is harder than the **NP** complexity class.

problem B? We essentially are saying if A is harder than B then given an algorithm that can solve A, we can easily make/derive a new one for B based on the previous algorithm for A that can solve B with a minor amount of extra time (only polynomial overhead). In other words, for a problem A to be harder than B, we must be able to reduce problem B into A in polynomial time. We represent problem A being harder than B via $A \geq_p B$ .Notice that a problem is considered harder than itself, as it technically can be reduced to itself in polynomial time. Also notice than the transitive property applies to hardness, or namely if $A \leq_p B$, and $B \leq_p C$, then $A \leq_p C$. Going back to **NP-Hard** problems, how would you go about showing that a problem is harder than every problem in **NP**? Well one way to prove a problem A is **NP-Hard**, assuming that we already know some other problem B is **NP-Hard** is to show that A is harder than B, or $A \geq_p B$. By definition of **NP-Hard**, $B \geq_p X \ \forall X \in \mathbf{NP}$. But if we've proved A is harder than B, than we also know by the transitive property of hardness that $A \geq_p X \ \forall X \in \mathbf{NP}$. This would make A **NP-Hard**. How would we go about proving that the first **NP-hard** problem was hard however? That proof is contained in the Cook-Levin theorem, which shows that all problems in **NP** can be reduced to the SAT problem. It only need the sole property that **NP** problems can be verified in polynomial time to do so. This is covered in the next lecture. Some examples of NP-Hard problems include the halting problem, finding Hamiltonian paths, or 3-SAT (a specific case of the SAT problem).

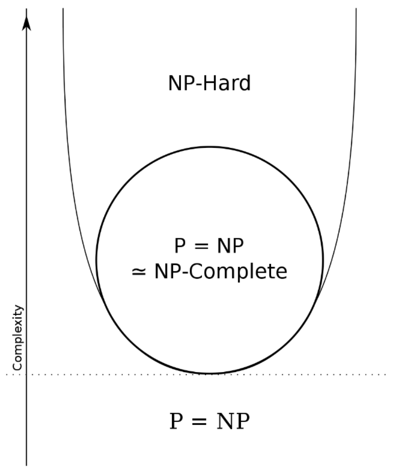

Figure 2: Diagram showcasing the relations between all the complexity classes covered, if **P** = **NP**.

4 NP-Complete

The **NP-Complete** complexity class contains all the problems which are in both **NP-Hard** and **NP**. Therefore, problems in **NP-Complete** are both harder than any other problem that can be verified in polynomial time and also verifiable in polynomial time themselves. A common example of this problem is the SAT (Satisfiability) problem, which basically asks whether or not a boolean expression can be evaluated to true or not. To verify a solution for this problem, we simply need to plug in boolean values for the variables in simplify to the final answer, which can be done so in polynomial time. **NP-Complete** problems have the interesting property that if we can show that one is actually in **P**, then **NP** = **P**. This is because every problem in **NP** can be reduced to our **NP-Complete** problem in polynomial time, and since we have shown that we can solve our **NP-Complete** problem in polynomial time, the time taken to solve every **NP** problem is simply polynomial. That makes **NP** = **P**.