CS 3510 Algorithms Last updated: November 2, 2023

Lecture 19: P vs. NP

Lecturer: Sahil Singla *Scribe(s): Allen Chang, Joseph Gulian*

For most of this class we've discussed problems with polynomial time algorithms, or in other words, the amount of time it takes to perform the algorithm is finitely bounded by some polynomial based on the inputs of the algorithm. At this point in the course, we'll move away from discussing individual problems, and towards discussing classes of problems. Some of these problems don't have known polynomial time algorithms which can be frustrating, so we'll end of the unit by discussing ways of introducing randomness to algorithms to broaden the problems we can solve.

Polynomial	NP-Complete
Shortest Path	Max-Path
Sorting	
MST	
Bipartite Matching	3-D Matching
Min-Cut/Max-Flow	Max-Cut
Independent Set (Trees)	Independent Set (General Graphs)

Table 1: Problems in P and NP

Here are some of the problems we've discussed; as you can see, some of these problems have NP-Complete counterparts that we haven't discussed. Before we rigorously introduce the class NP-Complete, you could think of this as a class with no known Polynomial time algorithm. As a reminder, you might notice that some of the problems we've discussed can solve other problems we've discussed (like Max-Flow solving Max-Matching in a Bipartite Graph), this will lead into the other major part of the unit "reductions", but we'll wait to introduce this.

1 Types of Problems

To understand the different problems we're dealing with better, we will non-rigorously classify the problems we see.

1. **Decision Problems**: Given some constraints, does there exist a solution? (Yes/No)
   - e.g., For some graph G, does there exist a spanning tree of G with cost $\leq k$?
2. **Search Problems**: Given some constraints, either argue that there is no solution, or find a solution.
   - e.g., For some graph G, find a tree of cost $\leq k$.
3. **Optimization Problem**: Given some constraints and an objective, find a solution that minimizes the objective.

- For some graph G, find the minimum spanning tree.

You might think that the decision problem is less expressive than the optimization problem after all, isn't it better to know the minimum spanning tree, rather than checking if there is one below some threshold? It turns out for most of these problems, we can simply solve the optimization problem by searching with the decision problem. Take the minimum spanning tree for instance, if we add all the weights together, we can search from 0 to the weight of all the edges to find the weight of the minimum spanning tree. [1]

We could also solve search problems by giving a potential solution and creating the decision problem "is this the correct solution?" Of course this may not be particularly efficient.

2 Decidability and the Complexity Zoo

Now that we have an understanding of the different problems we're dealing with, let's try to understand how hard these problems are. The first characteristic we'll use is whether a problem is "decidability" or not; for our purposes a problem is decidable if it can be solved by some algorithm in a finite amount of time. Most of the problems we've discussed trivially fit this definition because they run in polynomial time. If we think a bit more abstractly, we can come up with the halting problem which is, given a program and inputs, can we check whether the program halts? This was proven to be undecidable by Alan Turing in 1937. [2]

You might think that a class of problems with one problem in it is rather small. Now that we have one problem, we could build out the class by showing another problem is at least as hard as some problem that is undecidable. [3]

Now that we have introduced the idea of decidability (and the fact that problems can be undecidable), we can draw a diagram depicting the complexity zoo.

We can draw the set P as the "easiest" set of problems; problems which can be solved by an algorithm in polynomial time. Next, we have NP (where P$\subseteq$NP), the set of problems solvable in Nondeterministic Polynomial time. We describe this class in the section below. [4] In another example, we can describe Exp-time (where NP$\subseteq$Exp-time) as the set of problems which are solvable in $2^{\text{polynomial}}$ time (e.g., running time of 2^{n^2} is allowed in Exp-time). We will be proving in Homework that all problems in NP belong to Exp-time. Finally, we have the set of decidable problems (where Exp-time$\subseteq$Decidable). Every problem that is solvable, even if it is in exponential time, must be decidable.

[1] You might be wondering, if the total of all the weights is W, and we're searching from 0 to W, wouldn't that be pseudo-polynomial? It would be, and that is inefficient, but we can binary search over the values instead and by doing this, it takes linear time with respect to the number of bits of W.

[2] You might know Alan Turing for his work on breaking the enigma cipher during World War II, but the most influential work he did for academia was in the field of computability. The two fields cryptography and computability are actually closely intertwined; as you might imagine, we don't want our cryptography easily broken so many cryptosystems rely on hard problems like factoring numbers or finding the shortest vector in a lattice.

[3] We will show this expansion to new problems in the coming lectures, but we won't do more with decidability. If you're more interested in decidability, take CS 4510.

[4] This is of course assuming P $\neq$ NP.

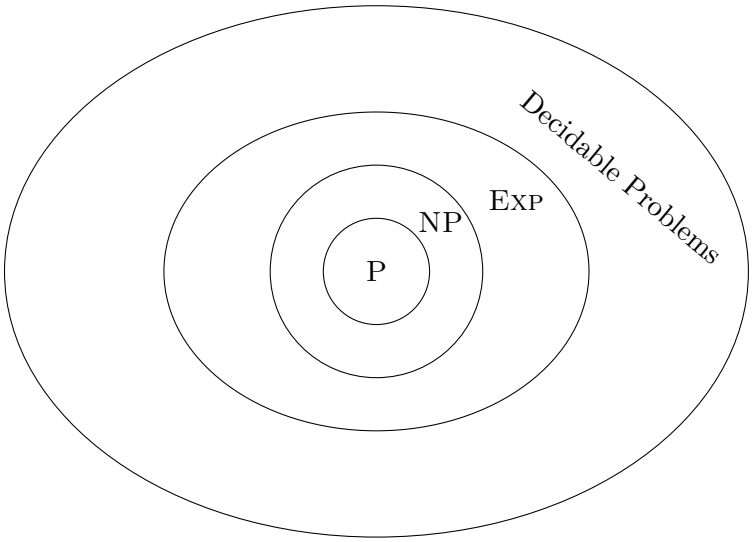


Figure 1: Problems of different complexities assuming P $\neq$ NP

3 Nondeterministic Polynomial

Before we discuss the class of NP, let's first discuss polynomial verifiability. A problem is polynomial verifiable if a solution to that problem can be verified in polynomial time. Consider the maximum matching problem discussed earlier; given some certificate of execution (information that can be used to check the solution like the matching set), we can verify that the set is maximal in polynomial time by checking for an augmenting path, and we can verify that the matching is valid in polynomial time. Because we can check the solution in polynomial time, that means the problem is polynomial verifiable.

Now we come to the main point: NP is the class of problems with polynomial time verifiers. For the purposes of this class, you could think of nondeterministic as being able to do multiple things at the same time. Using this understanding, if a problem has a polynomial time verifier, we could check all solutions to the problem at the same time, and in this way we could fine a solution to the problem. We could also do this without the nondeterminism, but this would make the solution take an exponential amount of time (as there are an exponential number of solutions which can not be checked simultaneously). Notice also that if a problem has a polynomial time algorithm to solve it, then we can trivially verify it in polynomial time as well, so we know that P $\subset$ NP or P $\subseteq$ NP.

A natural question to ask is whether or not $P = NP$; it's a popular question in popular culture and academia, but for decades no one has been able to prove whether $P = NP$ or not. [5] At this point we can understand the figure below: the classes P and NP are related in only two ways either P $\subset$ NP or P = NP. Again, we don't know which one is correct though, so we draw both potential graphs.

[5] This is the scribe talking. At some point in the coming coursework, we will likely ask a question which is really asking whether $P = NP$. If this is the case, the only correct answer we will accept is that we don't know whether $P = NP$ or $P \neq NP$.

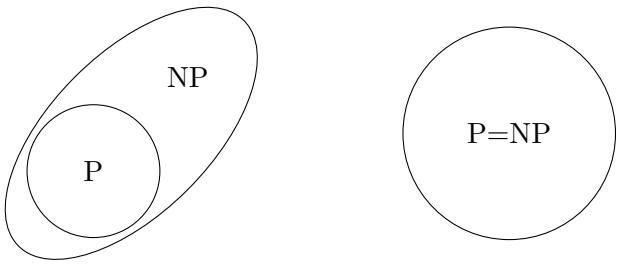


Figure 2: Either P $\subset$ NP or P = NP.

There are several ways to try to show whether $P = NP$, but the main way we'll focus on through this class is through the class NP-Hard. This is a more interesting class than we've previously discussed. Essentially in this class, if a problem has a polynomial time algorithm to solve the problem, this implies that $P = NP$. Proving this implication is beyond this class. [6], the important thing to understand is the implication "if a polynomial time algorithm exists for any problem in the class NP-Hard, all problems in NP would be polynomial time solvable.

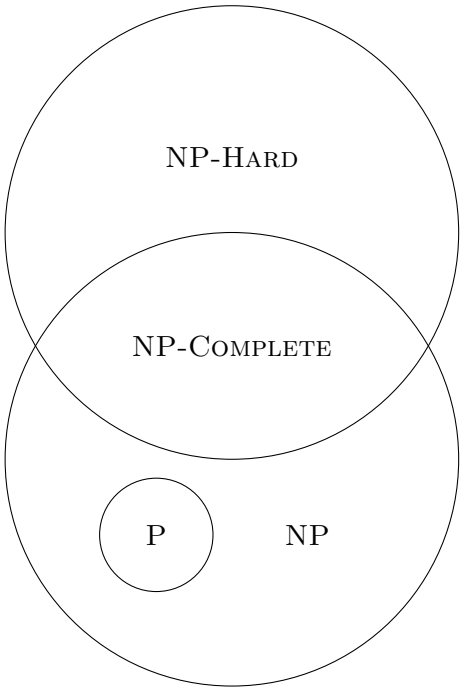
A corollary follows from this, if we believe that P $\neq$ NP, then this implies that if a problem in the class NP-Hard does not have a polynomial time algorithm.

4 NP-Complete

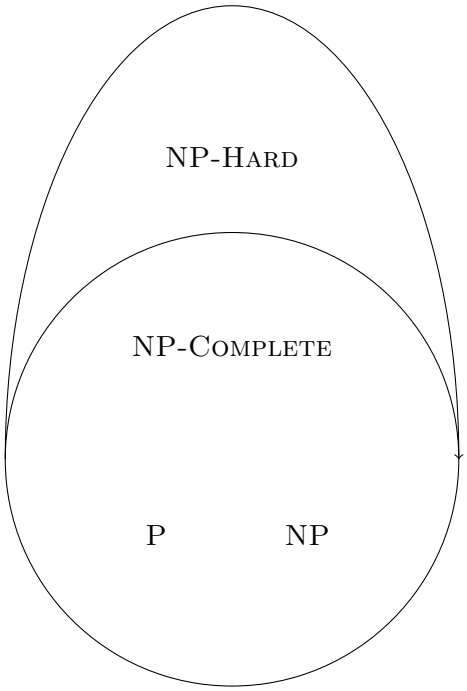
Now that we've discussed the classes NP and NP-Hard, if take the intersection of these two classes, then we have the class NP-Complete. For the remainder of the complexity unit, we will be showing that problems are in NP-Complete. Like with decidability, if we want to expand this class, we need one problem that we know is NP-Complete, and we will show that for a new problem it is harder than a problem that is NP-Complete, making it NP-Hard, and we will need to show that it is NP by giving a polynomial time verifier. Thus, to show that a problem is NP-Complete, it is sufficient to show that it is NP and NP-Hard.

Now that we understand the relationships between different classes, we will give a full graphical representation.

[6]The proof is called Cook-Levin Theorem, and it requires some more rigorous groundwork than we have time to lay out.



if P ≠ NP



if P = NP