St. Francis Xavier University
Department of Computer Science
CSCI 550: Approximation Algorithms
Lecture 2: Complexity Theory Basics
Fall 2021

1 P, NP, and All That

You may have noticed that we made mention of some well-known complexity classes in our introductory lecture, such as P and NP. In preparation for some of the later results we will discuss in this course, in this lecture we will briefly review the basics of complexity theory.

To begin, let us define the kinds of problems we will commonly see. A *decision problem* is a problem for which the answer is either "yes" or "no"; that is, an algorithm solving a decision problem decides whether or not its input meets some specified condition. By contrast, an *optimization problem* aims to find not just the answer, but the best answer for a given input. An optimization problem may take some auxiliary value N in addition to its input, and it may try to answer questions such as "does this input have a solution better than N?" In this sense, answering an optimization problem is similar to answering a series of decision problems where each decision problem fixes a different value N.

When we think of an "efficient" algorithm, we typically think of an algorithm that runs in *polynomial time*.

Definition 1 (Polynomial-time algorithm)**.** An algorithm for a problem is said to be polynomial-time if the running time of the algorithm is upper-bounded by some polynomial expression in the size of the input to the algorithm.

If the input to an algorithm is of size n, then a polynomial-time algorithm might perform n, or $n^2 + 25$, or $50n^{100} + n^{10}$ steps, but the crucial observation is that all of these values are polynomial in the size of the input.

To make things more precise, if x is an instance of a problem, then in order for an algorithm to operate on this instance we must encode it in some way. If $|x|$ is the number of bits in the encoding of the instance x, then we say that $|x|$ is the *size* of the input to the algorithm. Then, the algorithm is polynomial-time if it runs in time $p(|x|)$, where p is some polynomial expression.

If there exists some polynomial-time algorithm solving a decision problem, we say that the decision problem belongs to the complexity class P (or "is in P").

Definition 2 (The class P)**.** A decision problem belongs to the complexity class P if there exists a polynomial-time algorithm solving that decision problem.

Unfortunately, the world is not nice, and not all decision problems come with a polynomial-time algorithm. We cannot say that these problems belong to the class P, so we must have another class in which to place these problems. While we can define such a class in a few ways, we will use the following definition in this course.

Definition 3 (The class NP)**.** A decision problem belongs to the complexity class NP if there exists a verification algorithm A and two polynomial expressions p_1 and p_2 such that

1. For all inputs x and y, $A(x, y)$ runs in time $p_1(|x|)$;
2. For all instances x of the decision problem for which the answer is "yes", there exists a string y with $|y| \leq p_2(|x|)$ such that $A(x, y)$ outputs "yes"; and

3. For all instances x of the decision problem for which the answer is "no", and for all strings y with $|y| \leq p_2(|x|)$, $A(x, y)$ outputs "no".

The definition of the class NP we use here is sometimes referred to as the "verifier-based" definition. Here, the algorithm A takes as input the problem instance x and a string y, and then verifies that y is a "proof" for x; that is, if x corresponds to a "yes" answer, then y encodes the answer itself, and if x corresponds to a "no" answer, then no valid proof string y exists. The first condition of our definition ensures that A is able to verify its input in polynomial time. In addition, the statement $|y| \leq p_2(|x|)$ in conditions 2 and 3 ensures that y is a short proof for x, or a proof whose length is upper-bounded by some polynomial in the size of the instance.

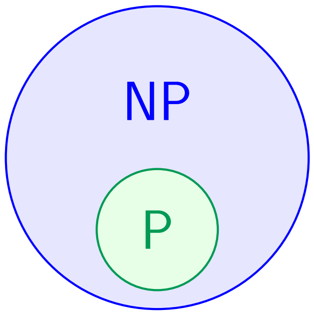
If you take away one thing from this lecture, let it be this:

NP does not stand for "non-polynomial"!

In a machine-centric context, the N in NP stands for "nondeterministic". This stems from the fact that a problem in NP can be solved by a nondeterministic Turing machine in polynomial time. By contrast, our "verifier" algorithm is entirely deterministic. Even though it cannot actually solve the problem, it can at least check whether a claimed solution is correct.

There are some known relationships between the classes P and NP. For example, clearly $\mathsf{P} \subseteq \mathsf{NP}$, since any decision problem that can be solved in polynomial time can also have its solution verified in polynomial time. On the other hand, proving (or disproving) the other direction of the inclusion is a notoriously difficult problem.[1]

All in all, here's how we generally think the complexity world looks at the moment (assuming $\mathsf{P} \neq \mathsf{NP}$):



2 Reductions

Occasionally, we may have a deep understanding of the complexity or difficulty of one decision problem, and we may come across another decision problem that resembles the first problem in some way. For example, recall that we discussed the (unweighted) set cover problem in our previous lecture. As it turns out, there exists another problem that is very similar to the set cover problem:

Vertex-Cover Given: an undirected graph $G = (V, E)$ and an integer k Determine: a subset of vertices $S \subseteq V$ such that $\|S\| \leq k$ and, if $(u, v) \in E$, then either $u \in S$, $v \in S$, or both

If, in the set cover problem, we want to know whether we can find a collection of k subsets that cover all elements of a ground set, then in the vertex cover problem we want to know whether we can find a subset of k vertices that covers at least one endpoint of every edge in the graph.

Indeed, the vertex cover problem and the set cover problem appear to be so similar that one might reasonably think that we can model an instance of the vertex cover problem in terms of an instance of the set cover

[1] If you have a correct solution to this problem, please let me know.

problem. This is indeed true: if we have a “black box” decider that solves instances of the set cover problem, then we can solve the vertex cover problem in the following way:

- Take an instance $\{G = (V, E), k\}$ of the vertex cover problem.
- Create an instance of the set cover problem as follows:
  - Take the ground set of elements, E, to be the set of edges E;
  - Take each subset S_v to contain all edges $e \in E$ incident to vertex $v \in V$; and
  - Take the integer k to be unchanged.
- Give the instance of the set cover problem to the “black box” decider.

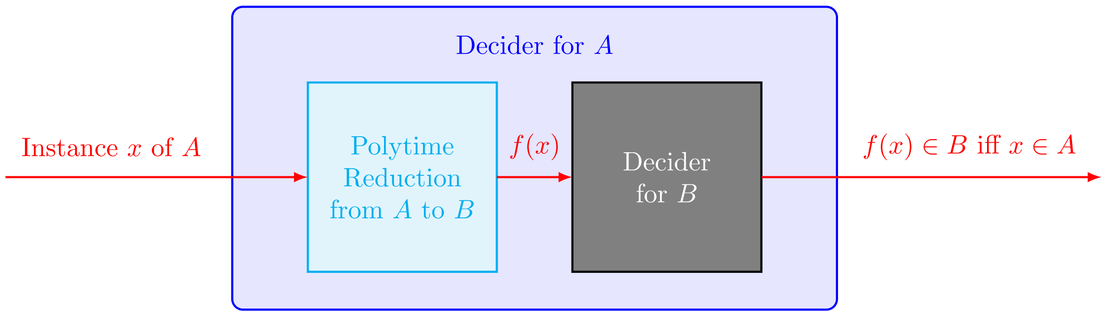
Thus, there exists a set cover of size at most k if and only if there exists a vertex cover of size at most k.

We can generalize this idea of modelling instances of one problem as instances of another problem using the notion of a *reduction*. Specifically, in this course, we would like our reductions to be efficient; that is, to run in polynomial time.

Definition 4 (Polynomial-time reduction)**.** Given two decision problems A and B, we say that there exists a polynomial-time reduction from A to B (or “A reduces to B”) if there exists some polynomial-time algorithm that takes an instance of A as input and produces an instance of B as output, such that the transformed instance has the same output as the original instance.

In other terms, if A reduces to B, then we can transform every instance x of A to an instance $f(x)$ of B. Moreover, since the transformed instance has the same output as the original instance, we can conclude that $f(x) \in B$ if and only if $x \in A$. Thus, we can use our transformation algorithm together with our “black box” decider for B to get an answer for our instance of A.

Diagrammatically, we can visualize a polynomial-time reduction from A to B in the following way:


Decider for A
Instance x of A
Polytime Reduction from A to B
$f(x)$
Decider for B
$f(x) \in B$ iff $x \in A$

We denote a polynomial-time reduction from A to B by the notation $A \leq_m^P B$. Thus, for example, we showed in this section that Vertex-Cover $\leq_m^P$ Set-Cover.

3 Hardness and Completeness

Having established the notion of a reduction, we can now talk about decision problems having certain complexity-theoretic properties.

For starters, if we know that there exists a reduction from a decision problem A to a decision problem B, then we can conclude that B is at least as difficult to solve as A. This is because we need to use the “black box” decider for B as part of our decider for A. With this observation, we can prove two important results:

Theorem 5. *If $A \leq_m^P B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$.*

Proof Sketch. If we are able to both apply the reduction from A to B in polynomial time and run the decider for B in polynomial time, then the overall decider for A also runs in polynomial time. □

Theorem 6. *If $A \leq_m^P B$ and $B \in \mathsf{NP}$, then $A \in \mathsf{NP}$.*

Proof Sketch. After applying the reduction from A to B, we are able to use a verifier for instances of B to verify instances of A. □

These two theorems tell us that, if we know to which complexity class a decision problem B belongs, and if we have a polynomial-time reduction from another decision problem A to B, then we can conclude that A belongs to the same complexity class.

The class NP is central to the study of complexity theory. As such, it has been investigated in depth, and a number of "NP-like" classes have been defined. The first such class we will introduce is the class NP-*hard*, which contains decision problems whose difficulty is on par with those problems in NP itself.

Definition 7 (NP-hard)**.** A decision problem A is NP-hard if, for every decision problem $B \in \mathsf{NP}$, there exists a polynomial-time reduction $B \leq_m^P A$.

Since we can reduce any decision problem in NP to an NP-hard decision problem A, we can informally characterize A as being *at least as difficult* as the most difficult decision problem in NP.

We can equivalently view an NP-hard decision problem A as a problem for which there exists an *oracle*[2] to solve A, and we can use this oracle for A to solve any problem $B \in \mathsf{NP}$ in polynomial time.

Note from our earlier definition that an NP-hard decision problem does not necessarily need to be in NP itself! If we know additionally that A belongs to the class NP, then it falls into a particular subclass at the intersection of NP and NP-hard, which we call NP-*complete*.

Definition 8 (NP-complete)**.** A decision problem A is NP-complete if $A \in \mathsf{NP}$ and A is NP-hard.

The class NP-complete contains all decision problems whose verifiers we can use to verify solutions to *any* other problem in NP via reductions; in that sense, therefore, NP-complete decision problems are the most difficult problems of any in NP. The class of NP-complete problems is occasionally denoted by NPC.

Just like with Theorems 5 and 6, we have a result pertaining to reductions from NP-complete problems:

Theorem 9. *If $A \in \mathsf{NP}$, B is NP-complete, and $B \leq_m^P A$, then A is NP-complete.*

Proof. We require the fact that polynomial-time reductions are transitive: that is, for three decision problems X, Y, and Z, if $X \leq_m^P Y$ and $Y \leq_m^P Z$, then $X \leq_m^P Z$.

Since B is NP-complete, we know that for all decision problems $C \in \mathsf{NP}$, there exists a polynomial-time reduction $C \leq_m^P B$. Additionally, we know by our hypothesis that $B \leq_m^P A$. By the transitivity of reductions, we therefore have that $C \leq_m^P A$ for all decision problems $C \in \mathsf{NP}$, meaning that A is NP-hard. Since A is additionally in NP, we conclude that A is NP-complete. □

If we know an example of an NP-complete decision problem B, then using this result, we can easily show that another decision problem A is NP-complete: all we need to do is show that $A \in \mathsf{NP}$, and then find a reduction $B \leq_m^P A$.

[2]An oracle is a special machine that is capable of solving its problem in only one computational step.

Let's now establish the hardness of our two decision problems, VERTEX-COVER and SET-COVER.

Theorem 10. VERTEX-COVER *is* NP-*complete.*

Proof Sketch. We first show that VERTEX-COVER is in NP. If we are given a subset of vertices S, a verifier can check that the subset is of size at most k and that the subset contains at least one endpoint of every edge in the graph.

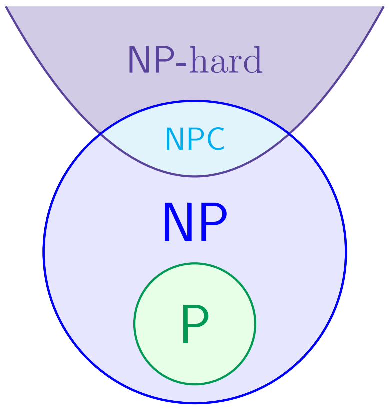
We can show that VERTEX-COVER is NP-complete by constructing a reduction from the well-known NP-complete problem 3-SATISFIABILITY to VERTEX-COVER. However, we omit that here. □

Theorem 11. SET-COVER *is* NP-*complete.*

Proof. We first show that SET-COVER is in NP. If we are given a collection of subsets S_i, a verifier can check that the collection contains at most k subsets and that the union of all subsets is equal to the ground set E.

To show that SET-COVER is NP-complete, we use our earlier reduction from VERTEX-COVER to SET-COVER. Since VERTEX-COVER is NP-complete and there exists a reduction from it to SET-COVER, we can conclude that SET-COVER is also NP-complete. □

Lastly, taking these two new classes into account, here's our expanded view of the complexity world:


NP-hard
NPC
NP
P