Lecture 1

Ishaan Lal

January 17, 2025

1 Lecture 1

What makes a problem hard? Or equivalently, what is a hard problem? You might imagine a situation where you are taking an exam and afterwards your friend turns to you and says *"Yo, Question 2... that was so hard!"*. And while you nod your head politely, in your mind you might be thinking: *"Question 2? That was the easiest question on the exam!"*

This is all to say that whatever definition we come up with for what is a "hard problem" should be one that is objective and has precise meaning.

1.1 CIS 262 TL;DR:

In computer science, there are two main types of problems that we deal with. First are **DECISION** problems, where we are given a setting/prompt, and our job is to simply determine *whether or not a solution exists*. That is, all we need to do is respond with YES or NO, and we must be correct about our answer.

Some example of decision problems include:

- Does this computer program have a bug?
- Is this number composite?
- Can this Sudoku puzzle be solved?

Note that we are only asked to *decide* if a solution exists. We don't necessarily need to provide it. However, more often than not, having a solution is helpful (if it exists). That is, we would really enjoy knowing the location of the bug, or the factorization of the number, or the solution to the Sudoku puzzle.

The other type of problems that we work with are called **OPTIMIZATION** problems. Here, we are given a setting and asked to find/provide the "best" possible solution (where "best" is clearly defined).

A simple example of an optimization problem is in a weighted graph (a graph where edges have numerical weights), we are tasked to find the shortest path between two nodes. This is an optimization problem because we need to find a path that is "best" (lowest cost). Dijkstra's algorithm is something that accomplishes this for us.

But that just tells us what a "problem" is. It doesn't yet tell us what a HARD problem is. For this, we will turn to definitions from computational complexity.

We will say that a problem is **EASY** if we can solve the problem quickly for *any* input. By "quickly", we mean that our solution runs in polynomial time with respect to the input size.

We will say that a problem is **HARD** if we can't solve the problem quickly for every input. That is, we may have a solution or algorithm for a problem, but it may run in exponential time for some inputs.

You might recognize this as a flavor of the distinction of P and NP problems, and that is correct.

When discussing NP-Complete problems, we are referring to a class of problems that are all considered to be equivalent via reductions. Examples include problems in routing shipments, designing circuits, protein folding, Rubik's cubes, etc.

And these problems are difficult! Nobody has been able to figure out how to solve these problems quickly for 50+ years. If you want to be famous instantly, just choose your favorite NP-Complete problem and find a polynomial time solution for it!

1.2 A Brief Aside on Moore's Law

In 1965, Gordon Moore, co-founder of Intel, posited that the number of transistors per chip will double every two years. So why do we even need to bother with solving these hard problems? Can't we just wait until we get faster computers?

Not quite! If a problem takes $\mathcal{O}(2^n)$ time, then even if the computer speed doubles, we would only be able to increase our value of n by 1.

Additionally, Moore's law is slowing down. We aren't quite seeing the same rate of increase of transistors per chip as we once saw before.

1.3 How Do We Solve Hard Problems?

If runtime for problems is exponential – large enough for our computers to be unable to solve them efficiently – what do we do? Just give up? Of course not!

Remember, when we say a problem has time complexity $\mathcal{O}(n^2)$, for example, this is an upper bound. Often, we denote it as the "worst case" bound. Similarly, if a problem is $\mathcal{O}(2^n)$, this could suggest that it is *worst case* time 2^n, but for many instances of the problem, the runtime may be substantially less!

1.3.1 A Bit of History

In the 1950s through 1970s, there was a search for a "general automated reasoning" tool. That is, there was a search for a "Universal Solver" that could solve any problem (yes, this gives flavors of ChatGPT, but oh well). To attempt creating a "Universal Solver", people decided to instead just focus on one single problem, and focus on solving it really really well, and then try to use this "really great solver" to also solve other problems. We'll be doing a similar thing in this class – start by focusing on one problem and trying to solve it really well. And that problem will be 3SAT!

2 The Satisfiability Problem (SAT)

The Satisfiability Problem **(SAT)** is described as follows: Given a formula φ which consists of boolean variables (quantities that take on the value `TRUE` or `FALSE`), does there exist an assigment to the variables such that φ evaluates to `TRUE`? Notice that this is a **decision problem**. We simply need to respond YES/NO in regards to if φ has a satisfiability assignment, and we must be correct about our verdict. We do not *need* to provide the assignment of variables itself.

As an example, we may be presented with:

$$\varphi : (x \vee y) \implies y$$

With this input, an algorithm that solves SAT would output "YES" – indicating that there is an assignement to x and y, namely $x =$ `TRUE` and $y =$ `TRUE` such that when x and y are substituted into φ, we get $\varphi =$ `TRUE`.

For another example, if given:

$$\varphi : (x \vee \overline{x}) \wedge y$$

the SAT solver would output "NO", indicating that there is no possible assignment to the variables such that $\varphi =$ `TRUE`. Equivalently, $\varphi =$ `FALSE` for every assignment of values to the variables.

In 1971, a massive breakthrough was made with the Cook-Levin Theorem which showed that SAT is an NP-Complete problem – the first to be classified. Nowadays, showing that problems are NP-complete involves performing a *reduction* – a method of showing two problems are essentially identical by representing the input of one problem as an equivalent input of another problem.

2.1 SAT Solvers

Modern SAT solvers take in formulas and return YES or NO. SAT solvers have existed in some form for a very long time. The first SAT solver was developed in 1962 by legends Davis, Putnam, and Loveland, and their algorithm was the basis for most other SAT solvers. In the late 1990s, we entered a "SAT revolution" where people developed some very powerful algorithms. Prior to this revolution, solvers were a bit too slow to be practical, but after the 90s, SAT became a competitive solution for many problems.

To see the growth of SAT solvers over time, the timeline is as follows:

- In 1962, the DPLL algorithm was created, which could solve SAT instances with around 10 variables.
- Over 20 years later, in 1986, we got BDDs, which solved SAT problems with 100 variables.
- In 1992, GSAT allowed us to solve for 300 variables.
- Four years later, GRASP was able to handle around 1,000 variables.
- Add another factor of 10, by 2001, Chaff could solve around 10,000 variables.
- And by 2005, we scale up once more, and have MiniSat which can solve around 100,000 variables
- In modern day, we can solve instances of SAT with **millions** of variables.

2.2 SAT Terminology

For simplicity, we will assume that the only logical operators in our equation φ are negations($\neg$), logical ORs ($\vee$) and logical ANDs ($\wedge$).

Pro Tip

Other boolean operators exist, such as $\implies$ and $\iff$, however, these can be decomposed into equivalent expressions using only the above three operators:

$$p \implies q \equiv \neg p \vee q$$

$$p \iff q \equiv (p \implies q) \wedge (q \implies p) \equiv (\neg p \vee q) \wedge (\neg q \vee p)$$

A **LITERAL** is defined to be an instance of a boolean variable or its negation. For example, the equation $\varphi = x_1 \vee \overline{x_2}$ has two literals, those being x_1 and $\overline{x_2}$

A **CLAUSE** is defined to be a disjuction/OR of literals. As an example, $\varphi = \overline{x_1} \vee x_2 \vee x_3$ is a clause, as it is a disjunction of three literals.

Example

QUESTION: Consider the following boolean equation:

$$\varphi = (\overline{x} \vee y) \wedge (x \vee y)$$

How many clauses does φ have? How many variables are there? How many literals are there?

ANSWER:

- φ has **2 clauses** being $(\overline{x} \vee y)$ and $(x \vee y)$
- φ has **2 variables**, being x and y
- φ has **4 literals**, being $\overline{x}, y, x, y$

A boolean formula is said to be in **Conjunctive Normal Form (CNF)** if it is a conjunction/AND of *clauses* (i.e. "an AND of ORs"). As an example, $\varphi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_3) \vee (x_2 \vee x_4)$ is a CNF as it consists of clauses that are ANDed together.

Example

QUESTION: Determine if the following formulas are in CNF:

- $(\overline{x} \vee y \vee z) \wedge (x \implies w)$
- $(\overline{x} \wedge y \wedge z) \vee (\overline{y} \wedge z)$
- $(\overline{x} \vee y \vee z) \wedge (\overline{y} \vee z)$
- $\overline{x} \vee y \vee z$
- $x \wedge \overline{x}$

ANSWER:

- NO, because the "clauses" are not all a disjunction of literals.
- NO, this is an "OR of ANDs", and not an "AND of ORs"
- YES
- YES, this can be thought of as a single clause
- YES, this is a conjunction of two clauses, both having just one literal.

2.3 From SAT to CNF-SAT

Most SAT solvers require the input formula to be in CNF. But of course, not all boolean formulas are born as CNF. Is there a way we can transform an arbitrary boolean formula φ so that it is in CNF form?

Theorem

Every boolean formula φ can be expressed in CNF.

One simple way to apply this transformation is to rewrite your equation in terms of $\vee, \wedge, \neg$, and then apply distributive properties and DeMorgan's laws until the formula is in CNF.

Important Logical Properties

$$\neg(p \vee q) \equiv \neg p \wedge \neg q \qquad \text{(DeMorgan's Law)}$$

$$\neg(p \wedge q) \equiv \neg p \vee \neg q \qquad \text{(DeMorgan's Law)}$$

$$(p \vee q) \vee r \equiv p \vee q \vee r \qquad \text{(Associativity of } \vee\text{)}$$

$$(p \wedge q) \wedge r \equiv p \wedge q \wedge r \qquad \text{(Associativity of } \wedge\text{)}$$

$$(p \wedge q) \vee r \equiv (p \vee r) \wedge (q \vee r) \qquad \text{(Distributive property of } \vee\text{)}$$

$$(p \vee q) \wedge r \equiv (p \wedge r) \vee (q \wedge r) \qquad \text{(Distributive property of } \wedge\text{)}$$

Next week, we will continue to build the foundation of SAT and solve some different, related problems!