Adam Blank | Lecture 26 | Autumn 2015

CSE 332

Data Abstractions

CSE 332: Data Abstractions

P vs. NP: The Million $ Problem

Complexity Classes

Definition (Complexity Class)
A **complexity class** is a set of problems limited by some resource contraint (time, space, etc.)

Today, we will talk about three: P, NP, and EXP

The Class P

Definition (The Class P)
P is the set of **decision problems** with a polynomial time (in terms of the input) algorithm.

We've spent pretty much this entire course talking about problems in P.

For example:

CONN	
Input(s):	Graph G
Output:	`true` iff G is connected

CONN $\in$ P
`dfs` solves **CONN** and takes $\mathcal{O}(|V|+|E|)$, which is the size of the input string (e.g., the graph).

2-COLOR $\in$ P
We showed this earlier!

And Others?

How About These? Are They in P?

- **3-COLOR**?
- **CIRCUITSAT**?
- **LONG-PATH**?
- **FACTOR**?

We have no idea!

There are a lot of open questions about P...

The Class EXP

But Is There Something NOT in P?
YES: The Halting Problem!
YES: Who wins a game of $n \times n$ chess?

As one might expect, there is another complexity class EXP:

Definition (The Class EXP)
EXP is the set of **decision problems** with an exponential time (in terms of the input) algorithm.

Generalized **CHESS** $\in$ EXP.

Notice that P $\subseteq$ EXP. That is, all problems with polynomial time worst-case solutions also have exponential time worst-case solutions.

Okay, now NP…

But a digression first…

Remember Finite State Machines?

You studied two types:

- DFAs (go through a single path to an end state)
- NFAs (go through **all possible paths** simultaneously)

NFAs "try everything" and if any of them work, it returns true. This idea is called **Non-determinism**. It's what the "N" in NP stands for.

Definition #1 of NP:

Definition (The Class NP)

NP is the set of **decision problems** with a **non-deterministic** polynomial time (in terms of the input) algorithm.

Unfortunately, this isn't particularly helpful to us. So, we'll turn to an equivalent (but more usable) definition.

Certifiers and NP

Definition (Certifier)

A **certifier** for problem **X** is an algorithm that takes as input:

- A `String` s, which is an instance of **X** (e.g., a graph, a number, a graph and a number, etc.)
- A `String` w, which acts as a "certificate" or "witness" that $s \in \mathbf{X}$

And returns:

- `false` (regardless of w) if $s \notin \mathbf{X}$
- `true` for **at least one** `String` w if $s \in \mathbf{X}$

Definition #2 of NP:

Definition (The Class NP)

NP is the set of **decision problems** with a polynomial time **certifier**.

A consequence of the fact that the certifier must run in polynomial time is that the valid "witness" must have **polynomial length** or the certifier wouldn't be able to read it.

Okay, this makes no sense, example plx?

We claim **3-COLOR** ∈ NP. To prove it, we need to find a **certifier**.

Certificate?

We get to choose what the certifier interprets the certificate as. For **3-COLOR**, we choose:

An assignment of colors to vertices (e.g., v_1 = red, v_2 = blue, v_3 = red)

Certifier

```
checkColors(G, assn) {
   if (assn isn't an assignment or G isn't a graph) {
      return false;
   }
   for (v : V) {
      for (w : v.neighbors()) {
         if (assn[v] == assn[w]) {
            return false;
         }
      }
   return true;
}
```

For this to work, we need to check a couple things:

1. Length of the certificate? $\mathcal{O}(|V|)$
2. Runtime of the certifier? $\mathcal{O}(|V|+|E|)$

FACTOR

CONN	
Input(s):	Number n; Number m
Output:	`true` iff n has a factor f, where $f \le m$

We claim **FACTOR** ∈ NP. To prove it, we need to find a **certifier**.

Certificate?

Some factor f with $f \le m$

Certifier

```
checkFactor((n, m), f) {
   if (n, m, or f isn't a number) {
      return false;
   }
   return f <= m && n % f == 0;
}
```

For this to work, we need to check a couple things:

1. Length of the certificate? $\mathcal{O}(\text{bits of } m)$
2. Runtime of the certifier? $\mathcal{O}(\text{bits of } n)$

Proving P ⊆ NP

Let **X** ∈ P. We claim **X** ∈ NP. To prove it, we need to find a **certifier**.

Certificate?

We don't need one!

Certifier

```
runX(s, _) {
   return XAlgorithm(s)
}
```

For this to work, we need to check a couple things:

1. Length of the certificate? $\mathcal{O}(1)$.
2. Runtime of the certifier? Well, **X** ∈ P…

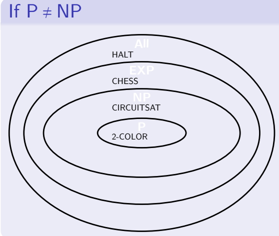
In other words, if **X** ∈ P, then there is a polynomial time algorithm that solves **X**.
So, the "verifier" just runs that program…

P vs. NP

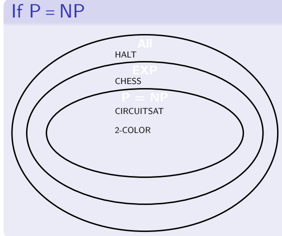
Finally, we can define P vs. NP…

Is finding a solution harder than certification/verification?

If P ≠ NP



If P = NP



Another way of looking at it. If P = NP:

- We can solve **3-COLOR**, **TSP**, **FACTOR**, **SAT**, etc. efficiently
- If we can solve **FACTOR** quickly, there goes RSA…oops

How Could We Even Prove P = NP?

Cook-Levin Theorem

Three Equivalent Statements:

- **CIRCUITSAT** is "harder" than any other problem in NP.
- **CIRCUITSAT** "captures" all other languages in NP.
- **CIRCUITSAT** is **NP-Hard**.

But we already proved that **3-COLOR** is "harder" than **CIRCUITSAT**!
So, **3-COLOR** is **also** **NP-Hard**.

Definition (NP-Complete)

A decision problem is **NP-Complete** if it is a member of NP and it is **NP-Hard**.

Is there an **NP-Hard** problem, **X**, where **X** is **not** **NP-Complete**?

Yes. The halting problem!

And?

Some NP-Complete Problems

CIRCUITSAT, **TSP**, **3-COLOR**, **LONG-PATH**, **HAM-PATH**, **SCHEDULING**, **SUBSET-SUM**, . . .

Interestingly, there are a bunch of problem we don't know the answer for:

Some Problems Not Known To Be NP-Complete

FACTOR, **GRAPH-ISOMORPHISM**, . . .