CSE 332

Data Abstractions

P **vs.** NP: **Efficient Reductions Between Problems**

Let's consider the **longest path** problem on a graph.

Remember, we were able to do **shortest paths** using Dijkstra's.

Take a few minutes to try to solve the **longest path** problem.

Definition (Decision Problem)

A **decision problem** (or **language**) is a set of strings ($L \subseteq \Sigma^*$).
An algorithm ($f : \Sigma^* \to \texttt{boolean}$) solves a decision problem iff it only outputs `true` if the input is in the set.

PRIMES	
Input(s):	Number x
Output:	`true` iff x is prime

An Algorithm that solves **PRIMES**

```
isPrime(x) {
   for (i = 2; i < x; i++) {
      if (x % i == 0) {
         return true;
      }
   }
   return false;
}
```

In this lecture, we'll be talking about **efficient reductions**. So, naturally, we have to answer two questions:

- What is an efficient algorithm?
- What is a reduction?

Efficient Algorithm

We say an algorithm is **efficient** if the worst-case analysis is a **polynomial**. Okay, but...

- $n^{10000000\dots}$ is polynomial
- $3000000000000000n^3$ is polynomial

Are those really efficient?
Well, no, but, in practice...

when a polynomial algorithm is found the constants are actually low

Polynomial runtime is a **very** low bar, if we can't even get that...

This lecture is about exposing **hidden** similarities between problems.

We will show that problems that are **cosmetically different** are **substantially the same**!

Our main tool to do this is called a **reduction**:

Reductions

We have two **decision problems**, **A** and **B**. To show that **A** is "at least as hard as" **B**, we

- Suppose we can solve **A**
- Create an algorithm that calls **A** as a method that solves **B**

To show they're the same, we have to do both directions.

Longest Paths and HAM!

Two New Computational Problems

LONG-PATH

Input(s):	Unweighted Graph G; Number k
Output:	`true` iff G has a path with k edges

HAM-PATH

Input(s):	Unweighted Graph G
Output:	`true` iff G has a path using all vertices

Suppose we could solve **LONG-PATH**...

"Algorithm"

```
HAM−PATH(G) {
   return LONG−PATH(G, |V| − 1)
}
```

Suppose we could solve **HAM-PATH**...

"Algorithm"

```
LONG−PATH(G, k) {
   for (G′ =(v_1,v_2,...,v_k) in G) {
      if (HAM−PATH(G′)) {
         return true;
      }
   }
   return false;
}
```

Definition (k-coloring)

A k-**coloring** of a graph G is an assignment of k colors to vertices such that no two adjacent vertices have the same color.

2-COLOR

Input(s):	Graph G
Output:	`true` iff G has a valid 2-coloring

Can we solve this?

Algorithm For 2-COLOR

Try all 2^n possible colorings of the input graph!

Can we solve this efficiently?

Efficient Algorithm For 2-COLOR

Do a `dfs` on the graph! Every time we hit a vertex, assign it the opposite color from the vertex we just visited. If there's a color conflict, output `false`. If we finish with no color conflict, output `true`.

Definition (k-coloring)

A k-**coloring** of a graph G is an assignment of k colors to vertices such that no two adjacent vertices have the same color.

3-COLOR

Input(s):	Graph G
Output:	`true` iff G has a valid 3-coloring

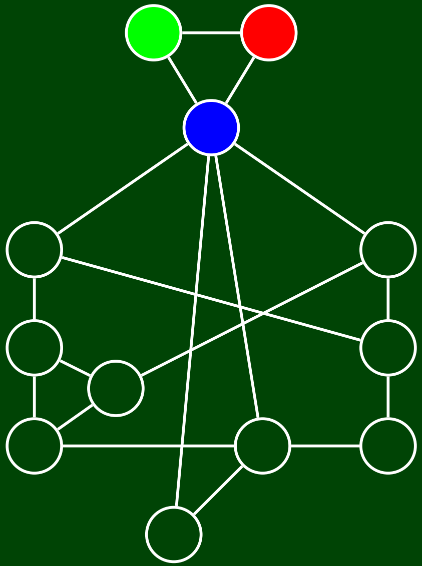
Inefficient Algorithm For **3-COLOR**

Try all 3^n possible colorings of the input graph!

Efficient Algorithm For **3-COLOR**

UNKNOWN

Find a valid 3-coloring of this graph. To orient ourselves, I've started it:



CIRCUITSAT

Input(s): n-Input/1-Output Circuit C
Output: `true` iff C has a satisfying assignment

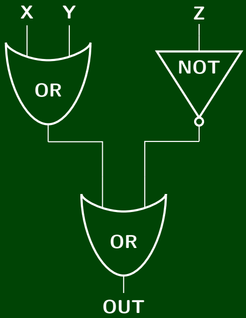
Inefficient Algorithm For CIRCUITSAT

Try all 2^n possible assignments of variables

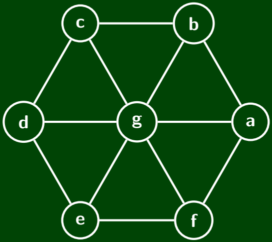
Efficient Algorithm For CIRCUITSAT

UNKNOWN

CIRCUITSAT



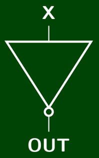
3-COLOR

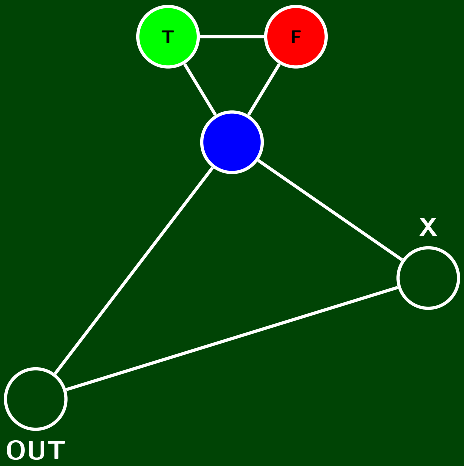


We don't know how to solve either of these problems. . .

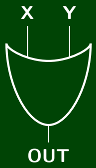
Could they be the same problem in disguise?

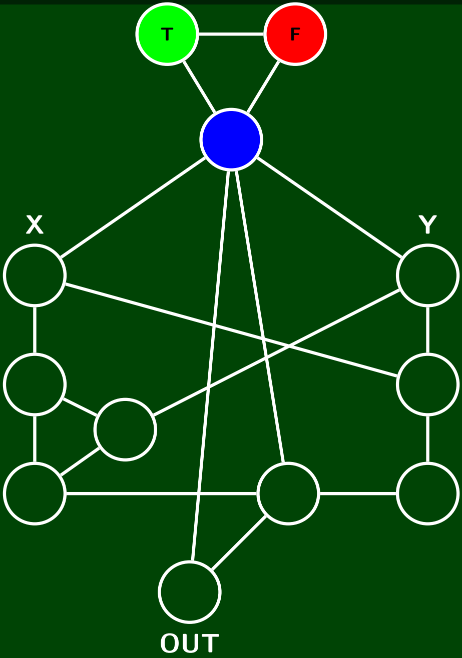
Not Gadget with Labels



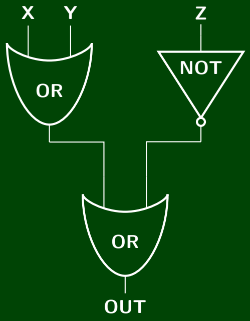


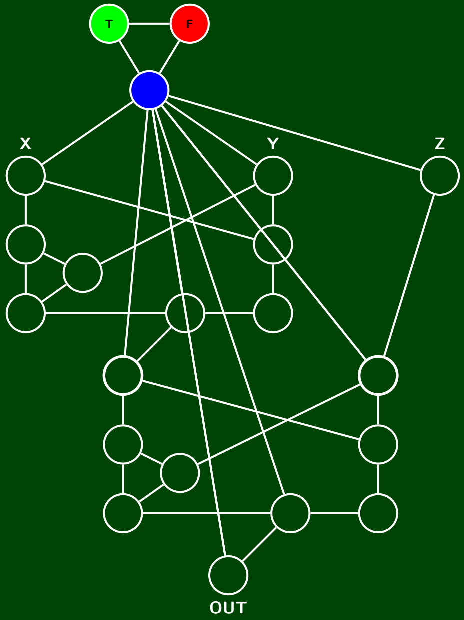
Or Gadget with Labels



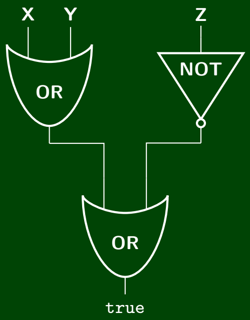


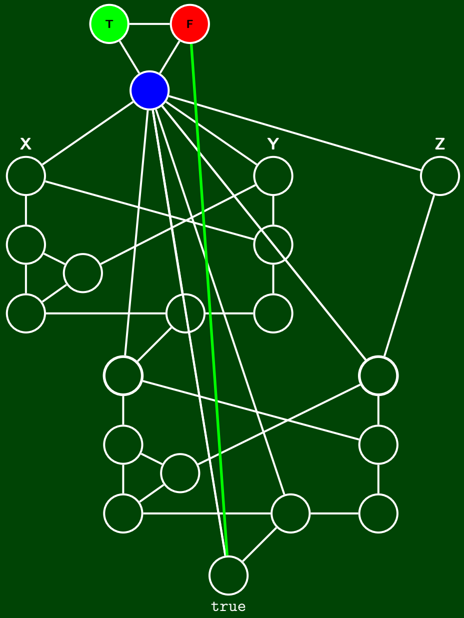
Circuit





SATISFIABLE Circuit





We found a way to "emulate" circuit satisfiability using three coloring!

If we can find a solution to **3-COLOR**, we can solve **CIRCUITSAT** quickly.

These problems are substantially the same

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:

CONNN	
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. . .

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.

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** $\in$ 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., $v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}$)

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[v]) {
            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? $\mathcal{O}(|V| + |E|)$

CONN

Input(s): Number n; Number m

Output: `true` iff n has a factor f, where $f \leq m$

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

Certificate?

Some factor f with $f \leq m$

Certifier

```
checkFactor((n, m), f) {
   if (n, m, or f isn't a number) {
      return false;
   }
   return m % 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 $\mathsf{P} \subseteq \mathsf{NP}$

Let **X** $\in$ P. We claim **X** $\in$ 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:

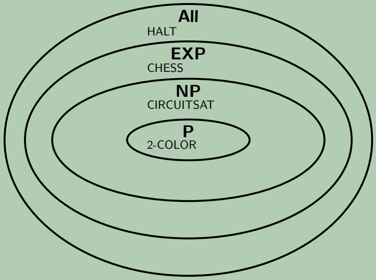
1. Length of the certificate? $\mathcal{O}(1)$.
2. Runtime of the certifier? Well, **X** $\in$ P...

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

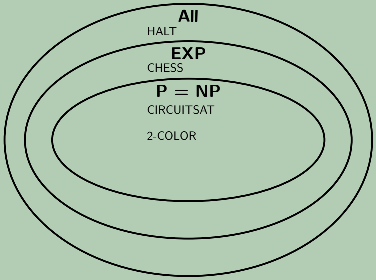
Finally, we can define P vs. NP...

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

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!

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**, . . .