CSE 332

Data Abstractions

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

The crayons are important!

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

BFS for unweighted

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

Idea? Explore all paths in the graph $O(|V|^{|V|})$

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

What is the # of edges in the longest (unweighted) path in a graph G?

Decision Problems

Definition (Decision Problem)

A **decision problem** (or **language**) is a set of strings ($L \subseteq \Sigma^*$).
An algorithm (from Σ^* to `boolean`) solves a decision problem when it outputs `true` iff the input is in the set.

EVENS = { "2", "0", "4", ... }

```
isEven(n){
    if (n is not a number){
        return false;
    }
    return n%2 == 0;
}
```

Definition (Decision Problem)

A **decision problem** (or **language**) is a set of strings ($L \subseteq \Sigma^*$).
An algorithm (from Σ^* to `boolean`) solves a decision problem when it outputs `true` iff 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 false;
      }
   }
   return true;
}
```

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**

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, which calls **A** as a method, to solve **B**

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

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

Input: G

LONG-PATH(G, |V| − 1)

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′ = (v1, v2, ..., vk) in G) {
      if (HAM−PATH(G′)) {
         return true;
      }
   }
   return false;
}
```

$\binom{n}{k} = \frac{n!}{k!(n-k)!} < n^k$

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

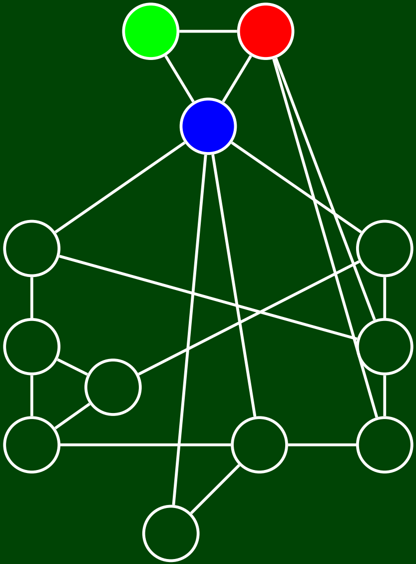
Inefficient Algorithm For **3-COLOR**

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

Efficient Algorithm For **3-COLOR**

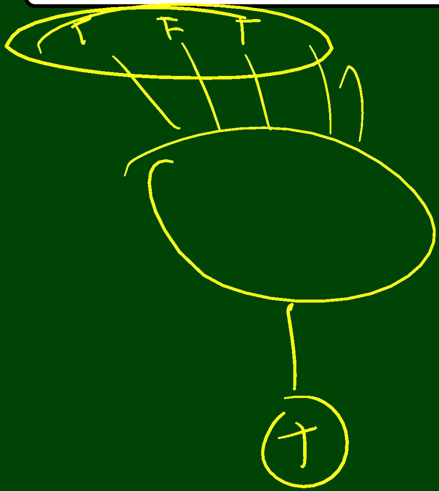
UNKNOWN

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



CIRCUITSAT

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

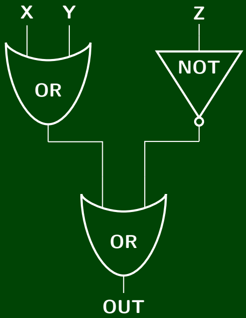
Inefficient Algorithm For **CIRCUITSAT**

Try all 2^n possible assignments of variables

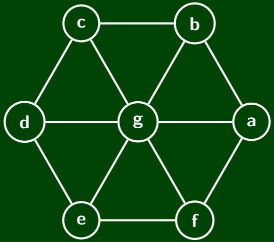
Efficient Algorithm For **CIRCUITSAT**

UNKNOWN

CIRCUITSAT



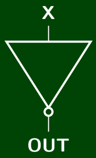
3-COLOR

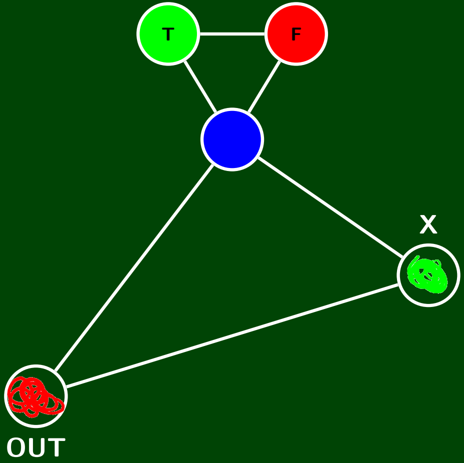


We don't know how to solve either of these problems. . .

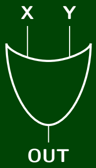
Could they be the same problem in disguise?

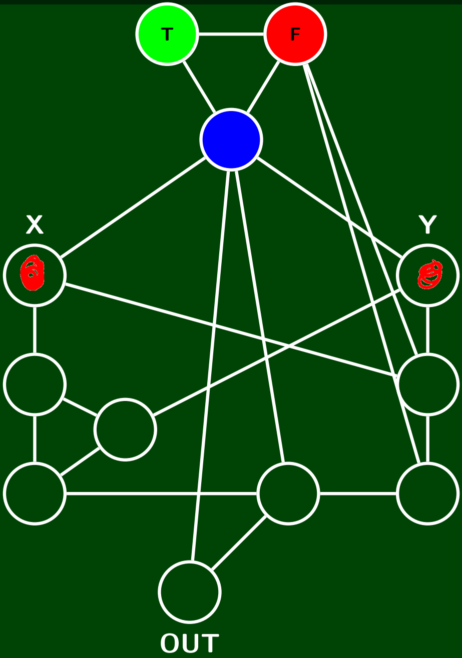
Not Gadget with Labels

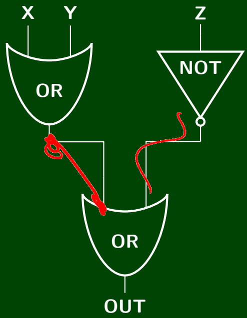


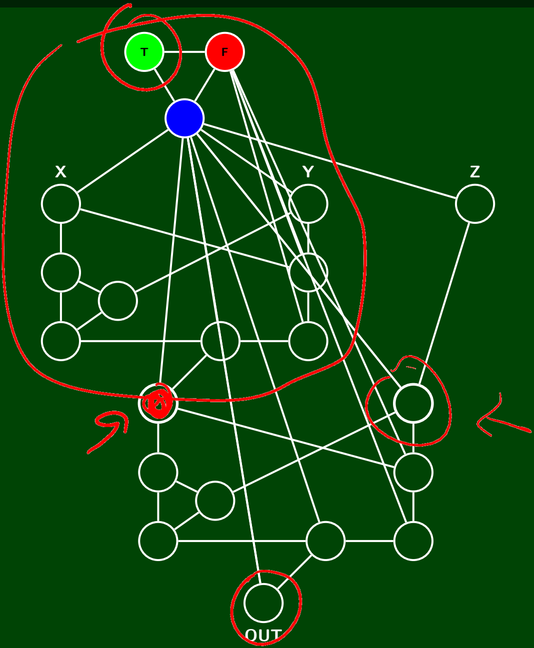


Or Gadget with Labels

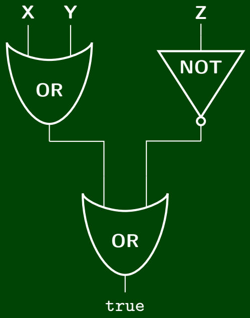


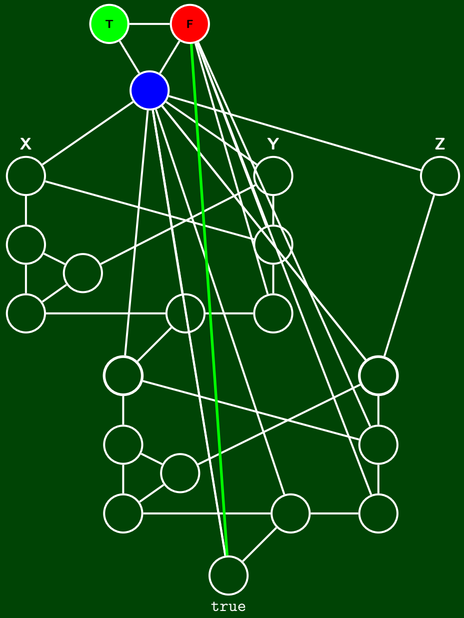



X
Y
Z
OR
NOT
OR
OUT


T
F
X
Y
Z
OUT

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