Data Abstractions
P vs. NP: Efficient Reductions Between Problems

The crayons are important!
Let's consider the **longest path** problem on a graph.

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

**Idea?** Explore all paths in the graph \( O(n \cdot m) \).

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

What is the **length** 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 $\Sigma^*$ to boolean) solves a decision problem when it outputs `true` iff the input is in the set.

\[
\text{Evens} = \{ 2, 4, 6, 8, \ldots \}
\]

\[
is\text{Even}(n) \exists \\
\text{if } (n \text{ is not a number}) \exists \\
\text{return } \text{false} \text{ else,} \\
\}
\]

\[
\text{return } m \mod 2 = 0
\]
Definition (Decision Problem)

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

PRIMES

<table>
<thead>
<tr>
<th>Input(s):</th>
<th>Number $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>true iff $x$ is prime</td>
</tr>
</tbody>
</table>

An Algorithm that solves PRIMES

```python
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…

- \( n^{10000000} \) 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:

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**:

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONG-PATH</td>
<td>Unweighted Graph $G$; Number $k$</td>
<td>true iff $G$ has a path with $k$ edges</td>
</tr>
<tr>
<td>HAM-PATH</td>
<td>Unweighted Graph $G$</td>
<td>true iff $G$ has a path using all vertices</td>
</tr>
</tbody>
</table>

**Input:** $G$

**Algorithm**

1. **LONG-PATH**($G$, $|V| - 1$)
2. for ($G' = (v_1, v_2, \ldots, v_k)$) in $G$
   1. if **HAM-PATH**($G'$)
      1. return true
   2. return false
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"
1 HAM-PATH(G) {
2     return LONG-PATH(G, |V| - 1)
3 }
```

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

```
"Algorithm"
1 LONG-PATH(G, k) {
2     for (G'=(v_1,v_2,...,v_k) in G) {
3         if (HAM-PATH(G')) {
4             return true;
5         }
6     }
7     return false;
8 }
```
Definition \((k\text{-coloring})\)

A \textbf{\textit{k-coloring}} of a graph \(G\) is an assignment of \(k\) colors to vertices such that no two adjacent vertices have the same color.

\[2\text{-COLOR}\]

<table>
<thead>
<tr>
<th>Input(s):</th>
<th>Graph (G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>true iff (G) has a valid 2-coloring</td>
</tr>
</tbody>
</table>

Can we solve this?

\textbf{Algorithm For 2-COLOR}

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

Can we solve this efficiently?

\textbf{Efficient Algorithm For 2-COLOR}

Do a \(\text{dfs}\) 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 \text{false}. If we finish with no color conflict, output \text{true}.
A 3-CRAYOLA Question

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.

<table>
<thead>
<tr>
<th>3-COLOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input(s):</strong> Graph $G$</td>
</tr>
<tr>
<td><strong>Output:</strong> true iff $G$ has a valid 3-coloring</td>
</tr>
</tbody>
</table>

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:
Another Decision Problem!

<table>
<thead>
<tr>
<th>CIRCUITSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input(s):</strong></td>
</tr>
<tr>
<td><strong>Output:</strong></td>
</tr>
</tbody>
</table>
Another Decision Problem!

<table>
<thead>
<tr>
<th>CIRCUITSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input(s):</strong></td>
</tr>
<tr>
<td><strong>Output:</strong></td>
</tr>
</tbody>
</table>

Inefficient Algorithm For CIRCUITSAT

Try all $2^n$ possible assignments of variables

Efficient Algorithm For CIRCUITSAT

UNKNOWN
Suspicious...

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
SATISFIABLE Circuit

X OR Y
Z
OR true

X Y
NOT
Z

T F
X Y Z
true
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