Reductions & NP-completeness

Slides by Carl Kingsford

Apr. 16, 2014

Section 8.1

Computational Complexity

- We've seen algorithms for lots of problems, and the goal was always to design an algorithm that ran in polynomial time.

- Sometimes we've claimed a problem is NP-hard as evidence that no such algorithm exists.

- Now, we'll formally say what that means.

Decision Problems

Decision Problems:

- Usually, we've considered optimization problems: given some input instance, output some answer that maximizes or minimizes a particular objective function.
- Most of computational complexity deals with a seemingly simpler type of problem: the decision problem.
- A decision problem just asks for a `yes` or a `no`.
- We phrased Circulation with Demands as a decision problem.

Decision is no harder than Optimization

The decision version of a problem is easier than (or the same as) the optimization version.

Why, for example, is this true of, say, Max Flow: "Is there a flow of value at least C?"

Decision is no harder than Optimization

The decision version of a problem is easier than (or the same as) the optimization version.

Why, for example, is this true of, say, Max Flow: "Is there a flow of value at least C?"

- If you could solve the optimization version and got a solution of value F for the flow, then you could just check to see if $F > C$.
- If you can solve the optimization problem, you can solve the decision problem.
- If the *decision* problem is hard, then so is the optimization version.

Encoding an Instance

We can encode an instance of a decision problem as a string.

Example. The encoding of a NETWORK FLOW might be:

$$u_1, v_1, c_1; u_2, v_2, c_2; u_3, v_3, c_3; ; s, t$$

More explicitly,

```
1,10,5;3,7,20;12,15,1;;10
```

How do we "know" intuitively that all of the problems we've considered so far can be encoded as a single string?

Encoding an Instance

We can encode an instance of a decision problem as a string.

Example. The encoding of a NETWORK FLOW might be:

$$u_1, v_1, c_1; u_2, v_2, c_2; u_3, v_3, c_3; ; s, t$$

More explicitly,

```
1,10,5;3,7,20;12,15,1;;10
```

How do we "know" intuitively that all of the problems we've considered so far can be encoded as a single string?

Because we can represent them in RAM as a string of bits!

Decision Problems and Languages

A decision problem X is really just sets of strings:

String	$\in X$?
`1,10,5;3,7,20;12,15,1;;1;12`	Yes
`1,10,5;3,7,20;12,15,1;;1;200`	No
$\vdots$	$\vdots$

Def. A language is a set of strings.
(Analogy: English is the set of valid English words.)

Hence, any decision problem is equivalent to deciding membership in some language.

We talk about "decision problems" and "languages" pretty much interchangeably.

Recap

Computational complexity primarily deals with decision problems.

A decision problem is no harder than the corresponding optimization problem.

A decision problem can be thought of as a set of the strings that encode "`yes`" instances.

Such sets are called languages.

How can we say a decision problem is hard?

A Model of Computation

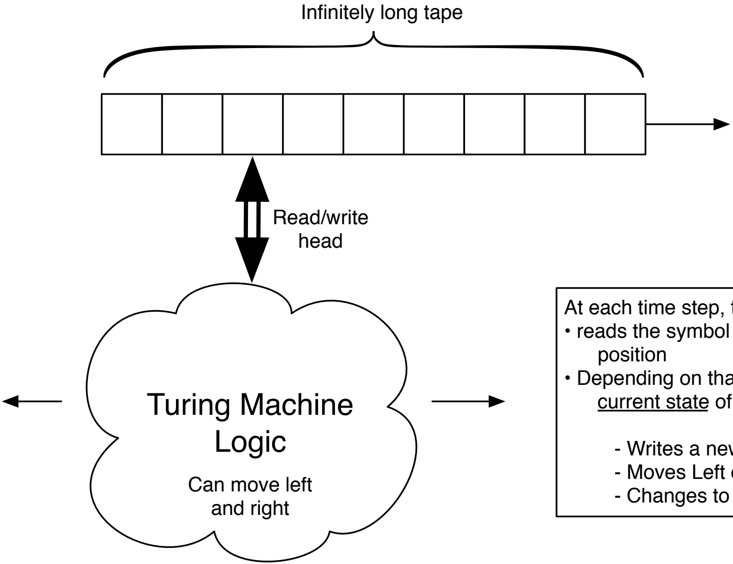
Ultimately, we want to say that "a computer can't recognize some language efficiently."

To do that, we have to decide what we mean by a *computer*.

We will mean a Turing Machine.

Church-Turing Thesis Everything that is efficiently computable is efficiently computable on a Turing Machine.

Turing Machine



At each time step, the TM:

- reads the symbol at the current position
- Depending on that symbol, and the <u>current state</u> of the TM, it:
  - Writes a new symbol *x*
  - Moves Left or Right
  - Changes to a new state *s*

The class **P**

P is the set of languages whose memberships are decidable by a Turing Machine that makes a polynomial number of steps.

By the Church-Turing thesis, this is the “same” as:

P is the set of decision problems that can be decided by a computer in a polynomial time.

From now on, you can just think of your normal computer as a Turing Machine — and we won't worry too much about that formalism.

The Class NP

Now that we have a different (more formal) view of **P**, we will define another class of problems called **NP**.

We need some new ideas.

Certificates

Recall the independent set problem (decision version):

Problem (Independent Set). *Given a graph G, is there set S of size $\geq k$ such that no two nodes in S are connected by an edge?*

Finding the set S is hard (we will see).

But if I give you a set S^*, checking whether S^* is the answer is easy: check that $|S| \geq k$ and no edges go between 2 nodes in S^*.

S^* acts as a certificate that $\langle G, k \rangle$ is a `yes` instance of Independent Set.

Efficient Certification

Def. An algorithm B is an efficient certifier for problem X if:

1. B is a polynomial time algorithm that takes two input strings I (instance of X) and C (a certificate).
2. B outputs either `yes` or `no`.
3. There is a polynomial $p(n)$ such that for every string I:

   $I \in X$ if and only if there exists string C of length $\leq p(|I|)$ such that $B(I, C) = \texttt{yes}$.

B is an algorithm that can decide whether an instance I is a `yes` instance if it is given some "help" in the form of a polynomially long certificate.

The class NP

NP is the set of languages for which there exists an efficient certifier.

The class NP

NP is the set of languages for which there exists an efficient certifier.

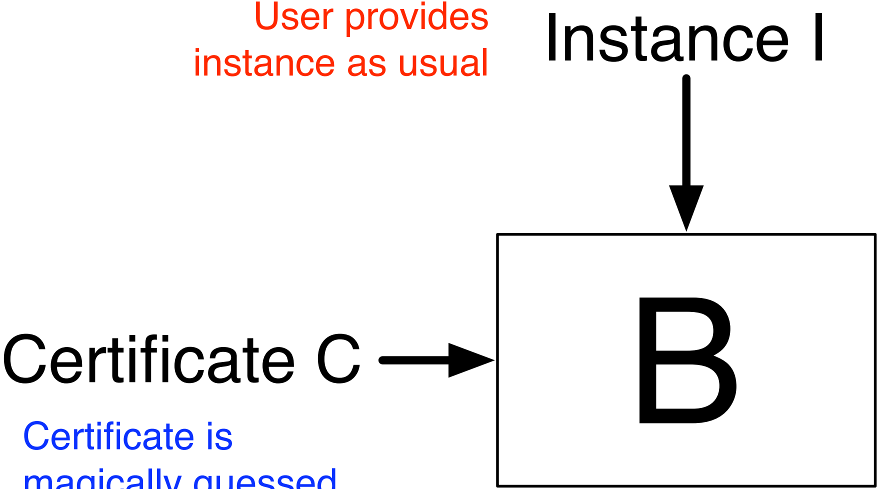
P is the set of languages for which there exists an efficient certifier that ignores the certificate.

That's the difference:

A problem is in **P** if we can decided it in polynomial time. It is in **NP** if we can decide them in polynomial time, if we are given the right certificate.

Do we have to find the certificates?

User provides
instance as usual



Certificate is
magically guessed

$\mathbf{P} \subseteq \mathbf{NP}$

Theorem. $\mathbf{P} \subseteq \mathbf{NP}$

Proof. Suppose $X \in \mathbf{P}$. Then there is a polynomial-time algorithm A for X.

To show that $X \in \mathbf{NP}$, we need to design an efficient certifier $B(I, C)$.

Just take $B(I, C) = A(I)$. $\square$

Every problem with a polynomial time algorithm is in **NP**.

$\mathbf{P} \neq \mathbf{NP}$?

The big question:

$$\mathbf{P} = \mathbf{NP}?$$

We know $\mathbf{P} \subseteq \mathbf{NP}$. So the question is:

Is there some problem in **NP** that is not in **P**?

Seems like the power of the certificate would help a lot.
But no one knows. . . .

How do we prove a problem is probably hard?

Reductions as tool for hardness

We want prove some problems are computationally difficult.

As a first step, we settle for relative judgements:

Problem X is at least as hard as problem Y

To prove such a statement, we reduce problem Y to problem X:

If you had a black box that can solve instances of problem X, how can you solve any instance of Y using polynomial number of steps, plus a polynomial number of calls to the black box that solves X?

We've Seen Reductions Before

Examples of Reductions:

- Max Bipartite Matching $\leq_P$ Max Network Flow.
- Image Segmentation $\leq_P$ Min-Cut.
- Airplane Scheduling $\leq_P$ Max Network Flow.
- Disjoint Paths $\leq_P$ Circulation with Demands & Lower Bounds.
- Circulation with Demands & Lower Bounds $\leq_P$ Circulation with Demands.
- Circulation with Demands $\leq_P$ Max Network Flow.

Polynomial Reductions

- If problem Y can be reduced to problem X, we denote this by $Y \leq_P X$.

- This means "Y is polynomal-time reducible to X."

- It also means that X is at least as hard as Y because if you can solve X, you can solve Y.

- Note: We reduce *to* the problem we want to show is the harder problem.

Polynomial Problems

Suppose:

- $Y \leq_P X$, and

- there is an polynomial time algorithm for X.

Then, there is a polynomial time algorithm for Y.

Why?

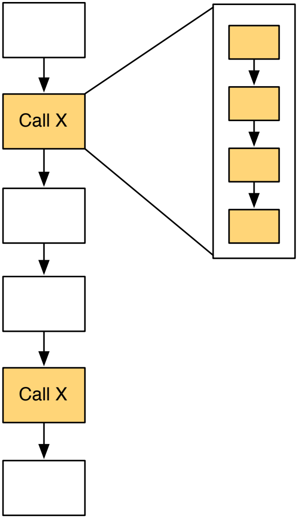
Polynomial Problems

Suppose:

- $Y \leq_P X$, and
- there is an polynomial time algorithm for X.

Then, there is a polynomial time algorithm for Y.

Why? Because polynomials compose.



Reductions for Hardness

Theorem. *If $Y \leq_P X$ and Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.*

Why? If we *could* solve X in polynomial time, then we'd be able to solve Y in polynomial time using the reduction, contradicting the assumption.

So: If we could find one hard problem Y, we could prove that another problem X is hard by reducing Y to X.

Vertex Cover

Def. A vertex cover of a graph is a set S of nodes such that every edge has at least one endpoint in S.

In other words, we try to "cover" each of the edges by choosing at least one of its vertices.

Problem (Vertex Cover). *Given a graph G and a number k, does G contain a vertex cover of size at most k.*

Independent Set to Vertex Cover

Problem (Independent Set). *Given graph G and a number k, does G contain a set of at least k independent vertices?*

Can we reduce independent set to vertex cover?

Problem (Vertex Cover). *Given a graph G and a number k, does G contain a vertex cover of size at most k.*

Relation btwn Vertex Cover and Indep. Set

Theorem. *If $G = (V, E)$ is a graph, then S is an independent set $\iff V - S$ is a vertex cover.*

Proof. $\implies$ Suppose S is an independent set, and let $e = (u, v)$ be some edge. Only one of u, v can be in S. Hence, at least one of u, v is in $V - S$. So, $V - S$ is a vertex cover.

$\impliedby$ Suppose $V - S$ is a vertex cover, and let $u, v \in S$. There can't be an edge between u and v (otherwise, that edge wouldn't be covered in $V - S$). So, S is an independent set. $\square$

Independent Set $\leq_P$ Vertex Cover

Independent Set $\leq_P$ Vertex Cover

To show this, we change any instance of Independent Set into an instance of Vertex Cover:

- Given an instance of Independent Set $\langle G, k \rangle$,
- We ask our Vertex Cover black box if there is a vertex cover $V - S$ of size $\leq |V| - k$.

By our previous theorem, S is an independent set iff $V - S$ is a vertex cover. If the Vertex Cover black box said:

yes: then S must be an independent set of size $\geq k$.
no: then there is no vertex cover $V - S$ of size $\leq |V| - k$, hence there is no independent set of size $\geq k$.

Vertex Cover $\leq_P$ Independent Set

Actually, we also have:

Vertex Cover $\leq_P$ Independent Set

Proof. To decide if G has an vertex cover of size k, we ask if it has an independent set of size $n - k$. $\square$

So: VERTEX COVER and INDEPENDENT SET are equivalently difficult.

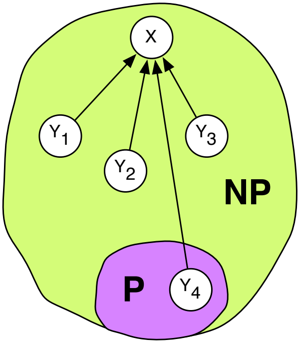
NP-completeness

Def. We say X is NP-complete if:

- $X \in \mathbf{NP}$
- for all $Y \in \mathbf{NP}$, $Y \leq_P X$.

If these hold, then X can be used to solve every problem in **NP**.

Therefore, X is definitely at least as hard as every problem in **NP**.



NP-completeness and P=NP

Theorem. *If X is NP-complete, then X is solvable in polynomial time if and only if $\mathbf{P} = \mathbf{NP}$.*

Proof. If $\mathbf{P} = \mathbf{NP}$, then X can be solved in polytime.

Suppose X is solvable in polytime, and let Y be any problem in **NP**. We can solve Y in polynomial time: reduce it to X.

Therefore, every problem in **NP** has a polytime algorithm and $\mathbf{P} = \mathbf{NP}$.

Reductions and NP-completeness

Theorem. *If Y is NP-complete, and*

1. *X is in NP*
2. *$Y \leq_P X$*

then X is NP-complete.

In other words, we can prove a new problem is NP-complete by reducing some other NP-complete problem to it.

Proof. Let Z be any problem in **NP**. Since Y is NP-complete, $Z \leq_P Y$. By assumption, $Y \leq_P X$. Therefore: $Z \leq_P Y \leq_P X$. $\square$

Some First NP-complete problem

We need to find some first NP-complete problem.

Finding the first NP-complete problem was the result of the Cook-Levin theorem.

We'll deal with this later. For now, trust me that:

- Independent Set is a *packing problem* and is NP-complete.
- Vertex Cover is a *covering problem* and is NP-complete.

Set Cover

Another very general and useful covering problem:

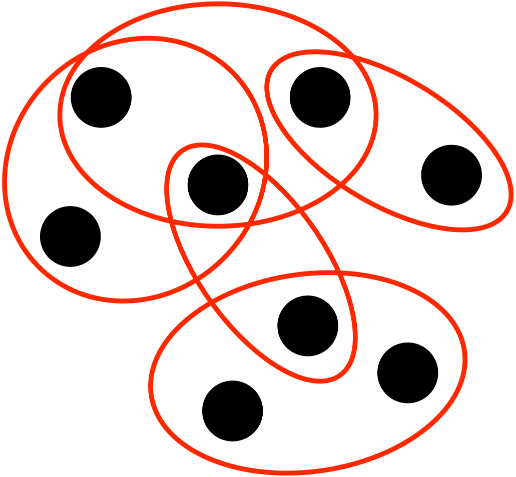
Problem (Set Cover). *Given a set U of elements and a collection $S_1, \ldots, S_m$ of subsets of U, is there a collection of at most k of these sets whose union equals U?*

We will show that

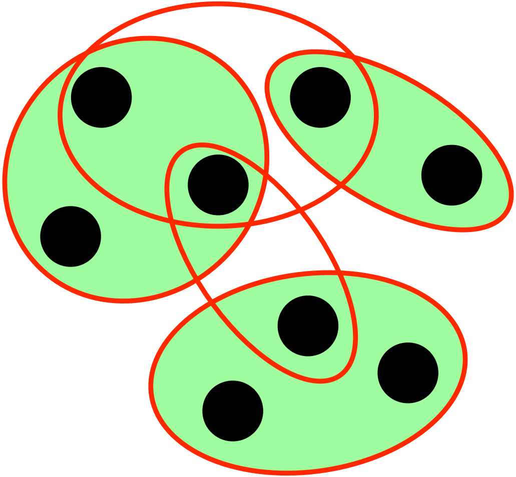
$$\textsc{Set Cover} \in NP$$
$$\textsc{Vertex Cover} \leq_P \textsc{Set Cover}$$

And therefore that Set Cover is NP-complete.

Set Cover, Figure



Set Cover, Figure



Vertex Cover $\leq_P$ Set Cover

Thm. Vertex Cover $\leq_P$ Set Cover

Proof. Let $G = (V, E)$ and k be an instance of VERTEX COVER. Create an instance of SET COVER:

- $U = E$
- Create a S_u for for each $u \in V$, where S_u contains the edges adjacent to u.

U can be covered by $\leq k$ sets iff G has a vertex cover of size $\leq k$.

Why? If k sets $S_{u_1}, \dots, S_{u_k}$ cover U then every edge is adjacent to at least one of the vertices $u_1, \dots, u_k$, yielding a vertex cover of size k.

If $u_1, \dots, u_k$ is a vertex cover, then sets $S_{u_1}, \dots, S_{u_k}$ cover U. $\square$

Last Step:

We still have to show that Set Cover is in **NP**!

The certificate is a list of k sets from the given collection.

We can check in polytime whether they cover all of U.

Since we have a certificate that can be checked in polynomial time, Set Cover is in **NP**.

Summary

You can prove a problem is NP-complete by reducing a known NP-complete problem to it.

We know the following problems are NP-complete:

- Vertex Cover
- Independent Set
- Set Cover

Warning: You should reduce the *known* NP-complete problem to the problem you are interested in. (You *will* mistakenly do this backwards sometimes.)