15-251

Great Theoretical Ideas in Computer Science

NP and NP-completeness I

February 24th, 2015

Toolbox of a computer scientist

1. Basic algorithmic techniques

   e.g. greedy algorithms, divide and conquer, dynamic programming, linear programming, semi-definite programming, etc…

2. Basic data structures

   e.g. queues, stacks, hash tables, binary search trees, etc…

3. Identifying and dealing with intractable problems

Toolbox of a computer scientist

1. Basic algorithmic techniques

   e.g. greedy algorithms, divide and conquer, dynamic programming, linear programming, semi-definite programming, etc…

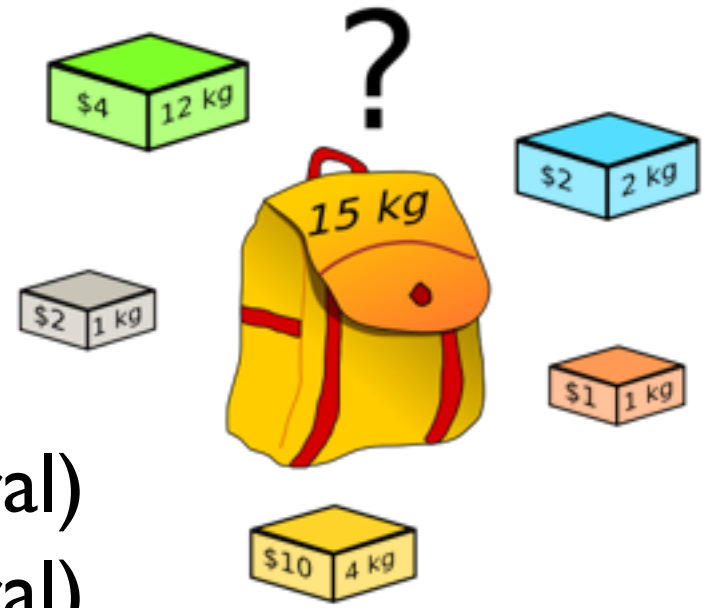
2. Basic data structures

   e.g. queues, stacks, hash tables, binary search trees, etc…

3. Identifying and dealing with intractable problems

Some examples

The Knapsack Problem



Input:

n items:

- value v_i (non-negative)
- weight w_i (non-negative, integral)

capacity W (non-negative, integral)

target value t (non-negative)

Output:

Yes, if there is a subset $S \subseteq \{1, 2, \ldots, n\}$

such that $\sum_{i \in S} v_i \geq t$ and $\sum_{i \in S} w_i \leq W$

Some examples

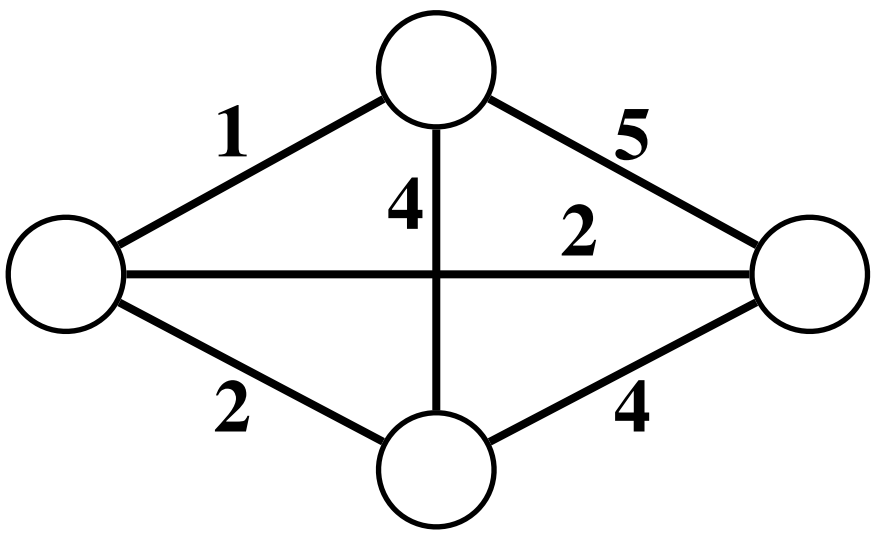
The Traveling Salesman Problem (TSP)

Input:

A graph $G = (V, E)$, edge weights w_e (non-negative, integral) and target t.

Output:

Yes, if there is a cycle of cost at most t that visits every vertex exactly once.



Some examples

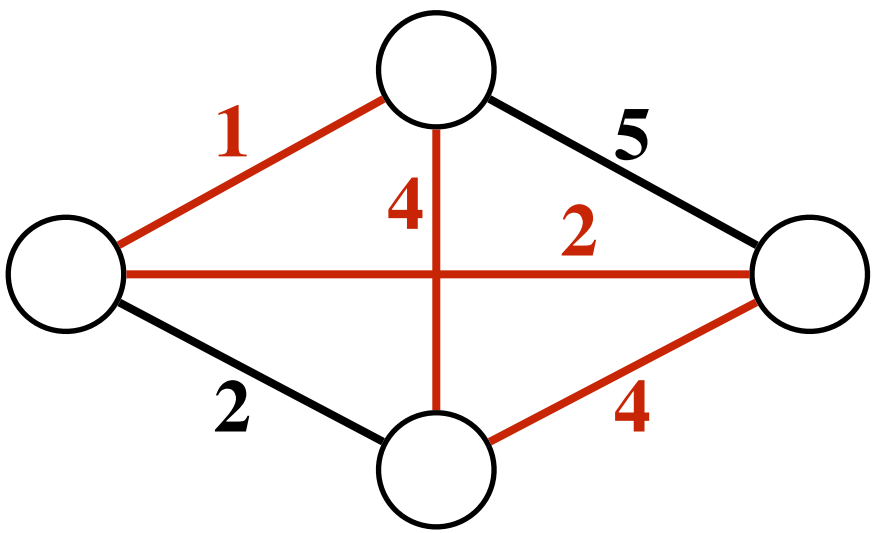
The Traveling Salesman Problem (TSP)

Input:

A graph $G = (V, E)$, edge weights w_e (non-negative, integral) and target t.

Output:

Yes, if there is a cycle of cost at most t that visits every vertex exactly once.



Some examples

The Traveling Salesman Problem (TSP)

Input:

A graph $G = (V, E)$, edge weights w_e (non-negative, integral) and target t.

Output:

Yes, if there is a cycle of cost at most t that visits every vertex exactly once.

In which order should you visit the cities so the total cost is less than $20,000?

Some examples

Satisfiability Problem (SAT)

Input:

A Boolean propositional formula.

e.g. $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee (\neg x_2 \wedge \neg x_3)$

Output:

Yes if there is an assignment to the variables that makes the formula True.

Some examples

Theorem Proving Problem

Input:

A FOL formula. A target length t.

Output:

Yes if there is a proof of the formula in
FOL deductive calculus of length at most t.

Toolbox of a computer scientist

3. Identifying and dealing with intractable problems

After decades of research, no one has been able to come up with an efficient solution for these problems.

It would be fantastic if we could directly prove that these problems cannot be solved in poly-time.

P

Toolbox of a computer scientist

3. Identifying and dealing with intractable problems

After decades of research, no one has been able to come up with an efficient solution for these problems.

It would be fantastic if we could directly prove that these problems cannot be solved in poly-time.

But we are far from doing this.

And who knows, maybe these problems are in P.

So what can we do???

Toolbox of a computer scientist

3. Identifying and dealing with intractable problems

So what can we do???

We can try to gather evidence that these problems are hard.

And in fact, we will be able to do this!

These problems are described as NP-hard or NP-complete. (synonyms for "computationally intractable")

You have to know what these mean!

In fact, every scientist and engineer should know what these mean!

Goal:

Find evidence that, say TSP, is computationally hard.

Revisiting reductions

A central concept used to compare the "difficulty" of problems.

will differ based on context

Now we are interested in polynomial time decidability vs **not** polynomial time decidability

Want to define: $A \leq B$

B is at least as hard as A
(with respect to polynomial time decidability).

B poly-time decidable $\implies$ A poly-time decidable

$B \in \mathsf{P} \implies A \in \mathsf{P}$

A not poly-time decidable $\implies$ B not poly-time decidable

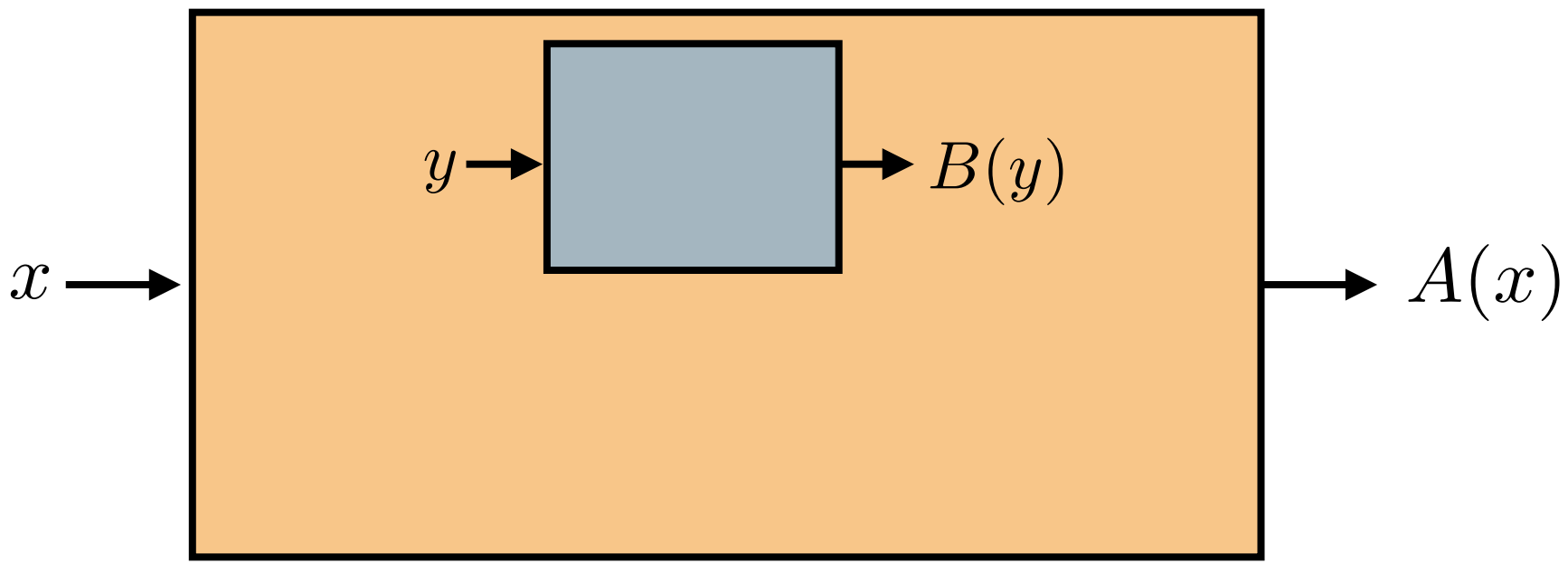
$A \notin \mathsf{P} \implies B \notin \mathsf{P}$

Revisiting reductions

Definition:

$A \leq_T^P B$ (A polynomial time reduces to B):

if it is possible to decide A in polynomial time
using an algorithm that decides B in polynomial time.



Revisiting reductions

```
def b(...):
    # some code that solves the problem B

def a(...):
    # some code that solves the problem A
    # that makes calls to function b when needed
```

If b efficient (poly time) implies a efficient, then we write

$$A \leq_T^P B$$

When you want to show $A \leq_T^P B$,
you need to come up with an efficient a.

Revisiting reductions

A:
Given a graph and an integer k, does there exist at least k pairs of vertices connected to each other?

B:
Given a graph and a pair of vertices (s,t),
is s and t connected?

A poly-time reduces to B

Revisiting reductions

A:

Given a sequence of integers, and a number k,
is there an increasing subsequence of length at least k?

3, 1, 5, 2, 3, 6, 4, 8

B:

Given two sequences of integers, and a number k,
is there a common subsequence of length at least k?

3, 1, 5, 2, 3, 6, 4, 8

1, 5, 7, 9, 2, 4, 1, 0, 2, 0, 3, 0, 4, 0, 8

A poly-time reduces to B

The two sides of reductions

1. Expand the landscape of tractable problems.

If $A \leq_T^P B$ and B is tractable, then A is tractable.

$$B \in \mathsf{P} \implies A \in \mathsf{P}$$

Whenever you are given a new problem to solve:

- check if it is already a problem you know how to solve in disguise.
- check if it can be reduced to a problem you know how to solve.

The two sides of reductions

2. Expand the landscape of intractable problems.

If $A \leq_T^P B$ and A is intractable, then B is intractable.

$$A \notin \mathsf{P} \implies B \notin \mathsf{P}$$

But we are pretty lousy at showing a problem is intractable.

Maybe we can still make good use of this…

Gathering evidence for intractability

Suppose we want to gather evidence that $A \notin \mathsf{P}$.

If we can show $L \leq_T^P A$ for **many** L

(including some L that we really think should not be in P)

then that would be good evidence that $A \notin \mathsf{P}$.

Definitions of C-hard and C-complete

Definition:

Let $\mathcal{C}$ be some set of decision problems.

We say that decision problem A is $\mathcal{C}$-hard if

$$C \leq_T^P A \qquad \text{for all } C \in \mathcal{C}$$

A is harder than every problem in $\mathcal{C}$.

Definition:

We say that decision problem A is $\mathcal{C}$-complete if

- A is $\mathcal{C}$-hard
- $A \in \mathcal{C}$

A is the hardest problem in $\mathcal{C}$.

Definitions of C-hard and C-complete

Definition:

Let $\mathcal{C}$ be some set of decision problems.

We say that decision problem A is $\mathcal{C}$-hard if

$$C \leq_T^P A \qquad \text{for all } C \in \mathcal{C}$$

Observations:

Suppose A is $\mathcal{C}$-hard.

- If there is a problem in $\mathcal{C} - \mathrm{P}$, then $A \notin \mathrm{P}$.
- If $A \in \mathrm{P}$, every problem in $\mathcal{C}$ is in P.
  (In a sense, A encodes all problems in $\mathcal{C}$.)

Recall the goal

Goal: Find evidence that, say TSP, is computationally hard.

If A is $\mathcal{C}$-hard for a really big set $\mathcal{C}$, that is some evidence that A is computationally hard.

The bigger the $\mathcal{C}$, the better!

So what is a good choice for $\mathcal{C}$
if we want to show TSP is $\mathcal{C}$-hard?

Recall the goal

Goal: Find evidence that, say TSP, is computationally hard.

What if we let $\mathcal{C}$ be the set of all languages?

Can it be true that TSP is $\mathcal{C}$-hard?

What if we let $\mathcal{C}$ be the set of all languages decidable using Brute Force Search (BFS)?

Can it be true that TSP is $\mathcal{C}$-hard?

A complexity class for BFS

How can we identify the problems solvable using BFS?

What would be a reasonable definition?

What is common about the Knapsack Problem, TSP, SAT, and Theorem Proving Problem?

Seems hard to find a solution.

BUT, quite easy to verify a given solution.

A complexity class for BFS

Seems hard to find a solution.

BUT, quite easy to verify a given solution.

- Given a satisfying assignment to a SAT formula,
  can easily verify that it indeed satisfies the formula.
- Given a cycle that visits every vertex in a graph,
  can easily verify it is indeed a cycle that visits every vertex.
- Given a proof for a FOL sentence,
  can easily verify it is indeed a valid proof.

We often call the solution a "proof" for the fact that the instance is in the corresponding language.

A complexity class for BFS

Seems hard to find a solution.

BUT, quite easy to verify a given solution.

BFS goes through the solution space one by one to find a solution.

Easy to distinguish a needle from hay.

But the haystack is exponentially large.

The complexity class NP

Informally:

A decision problem is in NP if:

whenever we have a Yes instance,
there is a *simple* proof for this fact.

The complexity class NP

Informally:

A decision problem is in NP if:

whenever we have a Yes instance,
there is a *simple* proof for this fact.

1. The length of the proof is polynomial in the input size.
2. The proof can be verified/checked in polynomial time.

The complexity class NP

Formally:

Definition:

A language A is in NP if

- there is a polynomial time TM V
- a polynomial p

such that for all x:

$$x \in A \iff \exists u \text{ with } |u| \leq p(|x|) \text{ s.t. } V(x,u) = 1$$

"$x \in A$ iff there is a polynomial length proof u that is verifiable by a poly-time algorithm."

The complexity class NP

Formally:

Definition:

A language A is in NP if

- there is a polynomial time TM V
- a polynomial p

such that for all x:

$$x \in A \iff \exists u \text{ with } |u| \leq p(|x|) \text{ s.t. } V(x,u) = 1$$

If $x \in A$, there is some proof that leads V to accept.

If $x \notin A$, every "proof" leads V to reject.

The complexity class NP

"$x \in A$ iff there is a polynomial length proof u that is verifiable by a poly-time algorithm."

If $x \in A$, there is some proof that leads V to accept.
If $x \notin A$, every "proof" leads V to reject.

SAT $\in$ NP

x : a Boolean formula
u : an assignment to the variables that makes x True.

The complexity class NP

"$x \in A$ iff there is a polynomial length proof u that is verifiable by a poly-time algorithm."

If $x \in A$, there is some proof that leads V to accept.

If $x \notin A$, every "proof" leads V to reject.

TSP $\in$ NP

x: a graph with edge weights, and a target t

u: a cycle of cost at most t that visits each vertex exactly once.

proof = certificate = solution

The complexity class NP

2 Observations:

1. Every decision problem in NP can be solved using BFS.

   - Go through all possible proofs u, and run $V(x, u)$

2. This is a pretty BIG class!

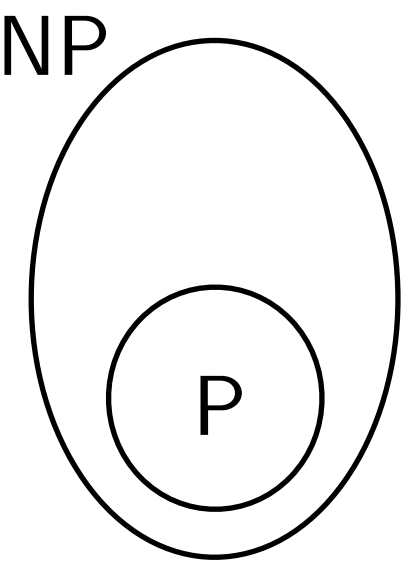
   Contains everything in P.

   If $A \in \mathrm{P}$, it has a poly-time decider M.

   V takes as input:

   real input x, and empty string as proof u

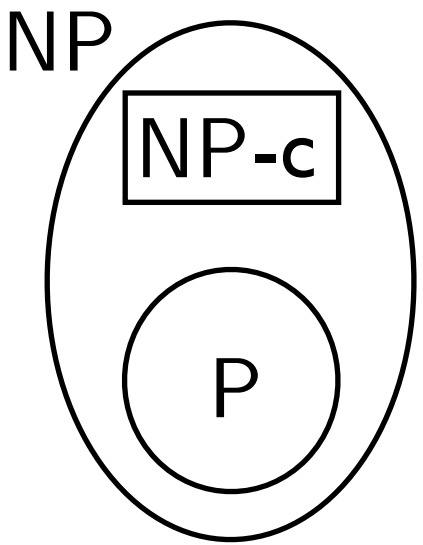
   V just runs $M(x)$ and returns its output.



People expect that NP contains much more than P.

Coming back to our goal

A language in NP-c represents the hardest language in NP.



In fact, if we believe $P \neq NP$,

$$A \in \text{NP-c} \implies A \notin P$$

We wanted to find evidence that the Knapsack problem, TSP, SAT, Theorem Proving problem are not in P.

Could it be true that one of them is NP-complete?

Is there **any** decision problem that is NP-complete?

Coming back to our goal

Is there **any** decision problem that is NP-complete?

If $A \in$ NP-c, it is like A encodes all problems whose solutions can be efficiently verifiable.

(If we could solve *A* in poly-time, we could solve every problem in NP in poly-time.)

Is NP-completeness a useful definition?

The Cook-Levin Theorem

Theorem (Cook 1971 - Levin 1973):

SAT is NP-complete.

That is, every problem in NP polynomial time reduces to SAT.

Karp's 21 NP-complete problems

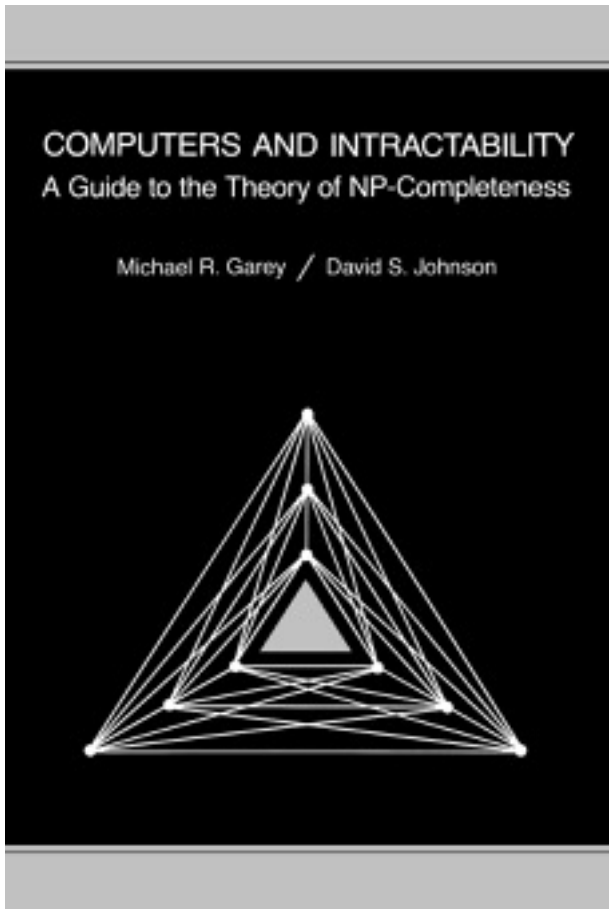
1972: "Reducibility Among Combinatorial Problems"

0-1 Integer Programming
Clique
Set Packing
Vertex Cover
Set Covering
Feedback Node Set
Feedback Arc Set
Directed Hamiltonian Cycle
Undirected Hamiltonian Cycle
3SAT
SAT-CNF
Partition
Clique Cover
Exact Cover
Hitting Set
Knapsack
Steiner Tree
3-Dimensional Matching
Job Sequencing
Max Cut
Chromatic Number

Today

Thousands of problems are known to be NP-complete.

(including SAT, TSP, Knapsack, Theorem Proving)



1979

Some other examples

Longest Common Subsequence

Given a set of sequences, and a number k,
is there a subsequence of length at least k
that is common to all the given sequences?

Subset Sum

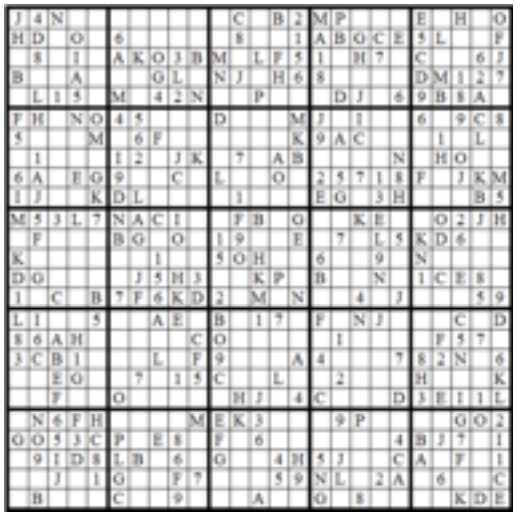
Given a set of integers, and a separate integer k,
is there a subset of the integers that sum to k?

Longest Path

Given a graph and an integer k,
is there a path of length at least k?

Some other examples

Sudoku (arbitrary dimension)



Super Mario Bros

Given a Super Mario Bros level, is it completable?

Tetris

Given a sequence of Tetris pieces, and a number k, can you clear more than k lines?

And many more…

Amazingly, all of these problems are "equivalent" to each other.

If A and B are NP-complete, then

$$A \leq_T^P B \quad \text{and} \quad B \leq_T^P A$$

How do you prove a problem is NP-complete?

You want to show A is NP-complete.

i.e., A is in NP, and for every $B \in$ NP, $B \leq_T^P A$.

How can you do it?

Observation:

Suppose you know L is NP-hard,

i.e., $B \leq_T^P L$ for every $B \in$ NP.

If you can show $L \leq_T^P A$

then A is NP-hard too!

How do you prove a problem is NP-complete?

You want to show A is NP-complete.

3 Steps:

1. Show that A is in NP.

2. Pick your favorite NP-complete problem L.

3. Show that $L \leq_T^P A$.

   i.e., given a poly-time algorithm for A,
   come up with a poly-time algorithm for L.

How do you prove a problem is NP-complete?

Cool!

How did Cook-Levin show SAT is NP-complete???

Will have to wait for this…

Good evidence for intractability?

If A is NP-hard,
that seems to be good evidence that $A \notin \mathrm{P}$.

(if you believe $\mathrm{P} \neq \mathrm{NP}$)

But is $\mathrm{P} \neq \mathrm{NP}$??

The P vs NP Question

What does NP stand for anyway?

Not Polynomial?

None Polynomial?

No Polynomial?

Nurse Practitioner?

It stands for Nondeterministic Polynomial time.

Languages in NP are the languages solvable in polynomial time by a nondeterministic TM.

DFA $\longleftrightarrow$ SFA (actually called NFA)

DTM $\longleftrightarrow$ NTM

What does NP stand for anyway?

Other contenders for the name of the class:

Herculean

Formidable

Hard-boiled

PET "possibly exponential time"

"provably exponential time"

"previously exponential time"

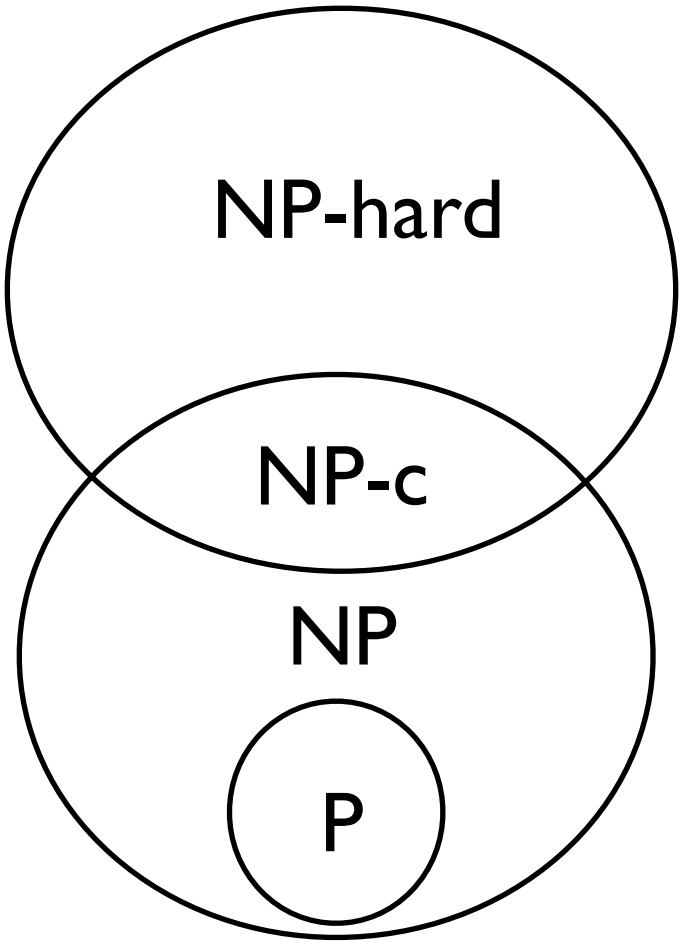
The P vs NP question

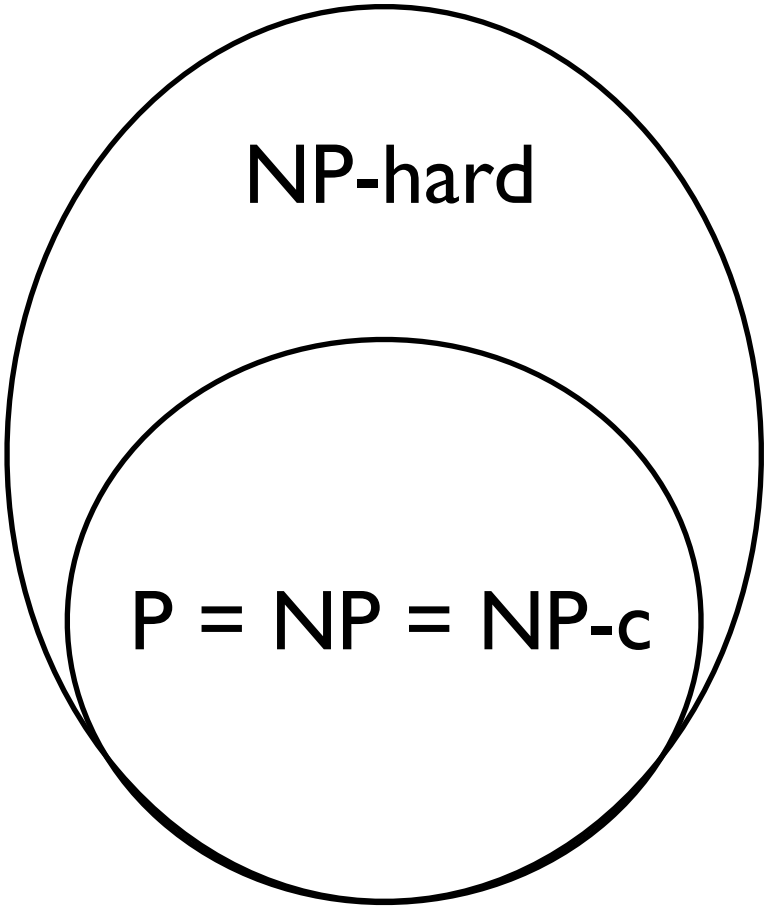
Yes, if you solve it, you get 1 million dollars.

But this understates the importance of the problem!

We are now pretty confident that this is one of the deepest questions we have ever asked.

The two possible worlds





The P vs NP question

To show P != NP:

pick you favorite NP-complete problem,
show that it cannot be solved in polynomial time.

To show P = NP:

pick you favorite NP-complete problem,
show that it can be solved in polynomial time.

P = NP is equivalent to SUDOKU $\in$ P

What do experts think?

Two polls from 2002 and 2012

Number of respondents in 2002: 100

Number of respondents in 2012: 152

	$P \neq NP$	$P = NP$	Ind	DC	DK
2002	61(61%)	9(9%)	4(4%)	1(1%)	22(22%)
2012	126 (83%)	12 (9%)	5 (3%)	5 (3%)	1(0.6%)

What do experts think?

Common arguments you'll hear:

If P = NP, someone would have found an efficient algorithm for an NP-complete problem.

Verifying a given solution is quite easy.

Coming up with a solution seems to require "creativity".

Can creativity be automated?

What do experts think?

Common arguments you'll hear:

Recall: $L \subseteq P \subseteq PSPACE \subseteq EXP$

$L \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP$

We know $L \neq PSPACE$

We also know $P \neq EXP$

How to deal with NP-complete problems?

1. Focus on tractable special cases

   e.g. Given some graph problem.
   Maybe it is easy for bipartite graphs, or trees.

2. Heuristics

   Algorithms that are not guaranteed to be always correct.

3. Try to do better than BFS

4. Approximation algorithms

Summary

Summary

- How do you identify intractable problems? (problems not in P) e.g. SAT, TSP, …
- Can't prove they are intractable. Can we gather some sort of evidence?
- Poly-time reductions. $A \leq_T^P B$
- If we can show $L \leq_T^P A$, for **many** L, that can be good evidence that $A \notin$ P.
- Definitions of $\mathcal{C}$-hard, $\mathcal{C}$-complete.
- What is a good choice for $\mathcal{C}$, if we want to show, say, SAT is $\mathcal{C}$-hard?

Summary

- The complexity class NP
- NP-hard, NP-complete
- Cook-Levin Theorem: SAT is NP-complete
- **Many** natural problems are NP-complete
- So is that good evidence that a problem is intractable?
- The P vs NP question

Next Time

How did Cook-Levin show SAT is NP-complete?

And examples of reductions that show other problems are NP-complete.