# Proving NP-Completeness

Most NP-complete problems are proven NP-complete by a reduction to a known NP-complete problem.

If problem A has a polynomial-time reduction to problem B, which of the following can you conclude?

- **if A is NP-complete, there can't be a polynomial-time algorithm for B**
- if B is NP-complete, there can't be a polynomial-time algorithm for A
- none of the above



# Reduction Example

**Course scheduling** –

Given a set of courses requested by each student, a set of time slots, and an integer *k*, determine if there is an assignment of courses to time slots with at most *k* conflicts amongst the students' schedules.

Is course scheduling in NP?

- yes
  - it is a decision problem (yes/no question)
  - a 'yes' answer is verifiable in polynomial time – given an assignment of courses to time slots, one simply needs to go through each student's schedule and count conflicts

Is course scheduling NP-complete?



# Reduction Example

**Course scheduling** –
Given a set of courses requested by each student, a set of time slots, and an integer k, determine if there is an assignment of courses to time slots with at most k conflicts amongst the students' schedules.

Is course scheduling NP-complete?

- we need a known NP complete problem to reduce to this problem

**3-coloring** –
Determine if you can color a graph with three colors so that no two adjacent nodes have the same color.

- known to be NP-complete



# Reduction Example

remember that reducing A to B means that B is at least as hard as A
- B can't be easier than A, or else we'd have a better algorithm for A
- A could be easier than B, because there may be another algorithm for A

Reduce an instance of 3-coloring to an instance of course scheduling:

- vertices → courses
- edges → students
  - each student will have two requested courses, corresponding to the two vertices connected by the edge
- colors → timeslots
  - there will be three timeslots, corresponding to the three colors
- legal coloring → k = 0
  - an illegal 3-coloring means two adjacent nodes with the same color – this corresponds to a student whose courses are scheduled in the same timeslot
  - a schedule with no conflicts thus means a legal 3-coloring

This construction only takes polynomial time.

- traverse the graph – O(n) to create courses for n vertices, O(m) to create students for m edges, O(1) to create three timeslots

## Reduction Example

Thus, course scheduling is also NP-complete.

- if course scheduling could be solved in polynomial time, so could 3-coloring
- if 3-coloring can be solved in polynomial time, so can everything else in NP



## Direction of Reductions Matters!

A sudoku puzzle is an instance of an exact cover problem, meaning that sudoku can be reduced to exact cover.

- exact cover: given a collection of subsets of X, is there a group of those subsets such that every element of X is contained in exactly one subset?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | | | 4 | | 6 | | | 7 |
| | | | | | | 4 | | |
| | 1 | | | | | 6 | 5 | |
| 5 | | 9 | | 3 | | 7 | 8 | |
| | | | | 7 | | | | |
| | 4 | 8 | | 2 | | 1 | | 3 |
| | 5 | 2 | | | | | 9 | |
| | | 1 | | | | | | |
| 3 | | | 9 | | 2 | | | 5 |

Exact cover is NP complete.

Does this mean sudoku is NP complete too?

- not necessarily – the hard problem (exact cover) can be used to solve sudoku, but maybe there's some other way to solve sudoku more efficiently



## Proving NP-Completeness

But how do you get a known NP-complete problem in the first place?

- need a direct proof!

[1971] Cook-Levin theorem proved SAT is NP-complete.

SAT: is there an assignment of values to variables in a boolean expression so that the expression evaluates to TRUE?

Idea:

- start with a nondeterministic Turing machine which solves some problem in NP
- for each possible input, build a boolean expression which is satisfiable if and only if the machine accepts the input (a "yes" response)

Solving satisfiability tells whether or not the machine accepts the input, so satisfiability cannot be easier than the NP problem. (And since this construction can be done for any NP problem, satisfiability cannot be easier than any NP problem.)



## Cook-Levin Details

- SAT is in NP because
  - it is a decision problem
  - whether a given assignment of values actually satisfies the expression can be checked in polynomial time

## Cook-Levin Details

Ingredients – building a boolean expression to represent the Turing machine's task

- boolean variables represent the state of the Turing machine as it executes its program

| Variables | Intended interpretation | How many? |
|---|---|---|
| $T_{ijk}$ | True if tape cell $i$ contains symbol $j$ at step $k$ of the computation. | $O(p(n)^2)$ |
| $H_{ik}$ | True if the $M$'s read/write head is at tape cell $i$ at step $k$ of the computation. | $O(p(n)^2)$ |
| $Q_{qk}$ | True if $M$ is in state $q$ at step $k$ of the computation. | $O(p(n))$ |

http://en.wikipedia.org/wiki/Cook-Levin_theorem

  - p(n) = number of steps required for the Turing machine to accept or reject an input
    - polynomial function because it solves a problem in NP



## Cook-Levin Details

- expressions involving those variables represent the machine's input, goal, program, and rules of operation

| Expression | Conditions | Interpretation | How many? |
|---|---|---|---|
| $T_{ij0}$ | Tape cell $i$ initially contains symbol $j$ | Initial contents of the tape. For $i > n$-1 and $i < 0$, outside of the actual input $I$, the initial symbol is the special default/blank symbol. | $O(p(n))$ |
| $Q_{s0}$ | | Initial state of $M$. | 1 |
| $H_{00}$ | | Initial position of read/write head. | 1 |
| $T_{ijk} \rightarrow \neg T_{ij'k}$ | $j \neq j'$ | One symbol per tape cell. | $O(p(n)^2)$ |
| $T_{ijk} \wedge T_{ij'(k+1)} \rightarrow H_{ik}$ | $j \neq j'$ | Tape remains unchanged unless written. | $O(p(n)^2)$ |
| $Q_{qk} \rightarrow \neg Q_{q'k}$ | $q \neq q'$ | Only one state at a time. | $O(p(n))$ |
| $H_{ik} \rightarrow \neg H_{i'k}$ | $i \neq i'$ | Only one head position at a time. | $O(p(n)^3)$ |
| $(H_{ik} \wedge Q_{qk} \wedge T_{i\sigma k}) \rightarrow \bigvee_{(q, \sigma, q', \sigma', d) \in \delta}(H_{(i+d)(k+1)} \wedge Q_{q'(k+1)} \wedge T_{i\sigma'(k+1)})$ | $k<p(n)$ | Possible transitions at computation step $k$ when head is at position $i$. | $O(p(n)^2)$ |
| $\bigvee_{f \in F} Q_{fp(n)}$ | | Must finish in an accepting state. | 1 |

http://en.wikipedia.org/wiki/Cook-Levin_theorem



## Cook-Levin Details

The transformation from Turing machine to instance of SAT is a polynomial-time reduction.

  - $O(p(n)^2)$ variables
  - $O(p(n)^3)$ clauses

Transforming the SAT solution to a solution for the NP problem is O(1).

Thus, if SAT could be solved in polynomial time, so could any problem in NP.



## Hardness

Hard problems are at least as hard as the hardest problems in the class but may not be in the class itself.

**NP-hard** – set of problems that NP-complete problems can be reduced to

- all NP-complete problems are also NP-hard
- FNP versions of NP-complete problems are NP-hard
- there appear to be problems that are NP-hard but not NP-complete
  - e.g. determining whether or not a given chess board configuration is checkmate

## Common Myths

NP problems require exponential time.

Probably **yes**.

But technically **unknown**, because P != NP hasn't been proven.

- though it is generally thought that there are problems in NP that are not in P (and thus some problems in NP do require exponential time)

Also **no**, because there may be specific instances or classes of instances which can be solved in polynomial (or at least subexponential) time.

- e.g. maximum independent set – NP-hard/complete in general, O(n) for trees
- e.g. longest path – NP-hard/complete in general, O(n+m) for DAGs



## Common Myths

NP-complete problems are difficult because the search space is so large.

framed as a series of choices, the running time depends on branching factor, longest path

**No**, it's not the size of the search space as such, but that you have to look at so much of it.

- compare to greedy – and even many dynamic programming – algorithms

these are also framed as a series of choices, but either only one option is needed for each choice (a branching factor of 1) or there are many repeated subproblems so most branches can be pruned



## Common Myths

P is good and NP is bad.

**Not necessarily** –

- a polynomial-time algorithm may have large constant factors or exponents
- the exponential-time worst-case behavior may be rare
  - e.g. simplex algorithm for linear programming
- you might not need to solve large input sizes



## Harder than NP?

Are there any problems harder than NP?

**Yes.**

- e.g. Presburger arithmetic

## Presburger Arithmetic

Definition –

- contains constants 0 and 1 and the operator +
- axioms
  1. $\neg(0 = x + 1)$
  2. $x + 1 = y + 1 \rightarrow x = y$
  3. $x + 0 = x$
  4. $x + (y + 1) = (x + y) + 1$
  5. Let $P(x)$ be a first-order formula in the language of Presburger arithmetic with a free variable $x$ (and possibly other free variables). Then the following formula is an axiom: $(P(0) \wedge \forall x(P(x) \rightarrow P(x + 1))) \rightarrow \forall y\, P(y)$.

http://en.wikipedia.org/wiki/Presburger_arithmetic

Why is Presburger arithmetic interesting?

- it is *decidable* – an algorithm exists to determine if any given statement is derivable from the axioms
- the decision problem has worst-case runtime $2^{2^{cn}}$ for c > 0
- there are theorems of length *n* whose proofs have doubly exponential length
  - implies that there are computational limits on what can be proven by computer programs



## Unknown Problems

Are there problems whose hardness is unknown?

**Yes.**

Two examples –

- graph isomorphism
  - given two graphs G and H, determine if there is mapping *f* from vertices of G to vertices of H such that if (x,y) is an edge of G, (f(x),f(y)) is an edge of H
  - thought to be between P and NP-complete – can often be solved quickly in practice
- integer factorization
  - given integers n and m, does *n* have a factor at most *m*?
  - known to be in both NP and co-NP, thought to not be in P or NP-complete

    co-NP = can verify “no” answer in polynomial time
  - primality testing is in P