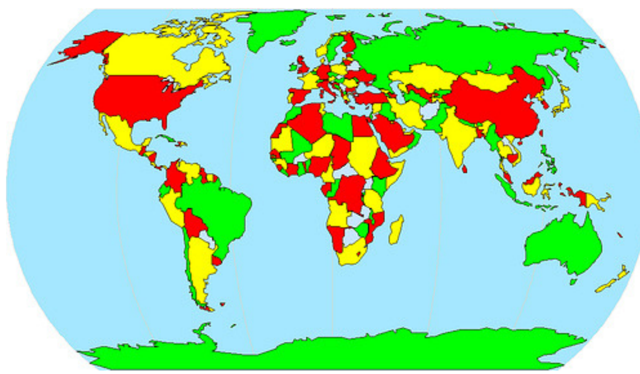
Limits of efficiency

- What is the fastest way to solve a problem?
  - E.g. sorting n numbers takes O(n log n) operations using mergesort.
  - Is there a different algorithm that sorts n numbers faster, say in O(n) time?
  - No. Sorting n numbers takes Ω(n log n) time if the algorithm can only compare numbers.

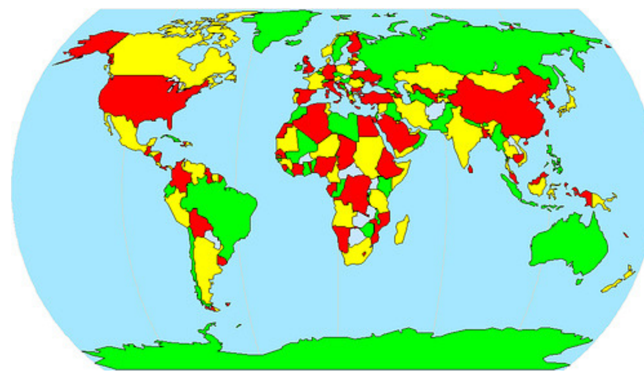
Limits of efficiency

- How much time does it take to solve a more difficult problem, like coloring?
- Four Color Theorem says every map can be colored with 4 colors, s.t. adjacent regions have different colors.
- So given a map, we can always efficiently answer the question whether the map can be 4-colored. Namely, yes.
- But sometimes 3 colors are enough to color the map.
- Can we efficiently determine whether a map can be 3-colored?
- If there are n regions, we can find a 3-coloring by trying all possible colorings.
  - There are 3^n possible colorings.
  - There are 195 countries in the world, and 1.1×10^{93} possible colorings!



Limits of efficiency

- Is there a much more efficient algorithm?
- Nobody knows of one. And almost everybody thinks no such algorithm exists.
- But no one can prove it doesn't exist either.
- The theory of NP-completeness is a mathematical attempt to prove some problems have no efficient solutions.
  - So far, it's led to more questions than answers...
- We'll define P, NP, and NP-completeness.



The class P

- A polynomial time (polytime) algorithm is one that runs in $O(n^k)$ time, for some constant k, when input has size n.
- P is the set of all problems that can be solved by a polytime algorithm.
  - These problems are called “efficiently computable”, because a polytime algorithm is considered efficient.
  - In practice though, an e.g. $O(n^3)$ algorithm is quite slow, even for moderate sized n.
- If a problem takes $\omega(n^k)$ time, for any constant k, it’s considered not efficiently solvable.
  - Ex An $\Omega(2^n)$ time or $\Omega(n!)$ time algorithm isn’t efficient.
  - We only know how to 3-color a map in $\Omega(3^n)$ time (more or less), so 3-coloring (currently) can’t be solved efficiently.
  - An $\Omega(3^n)$ time algorithm is much slower than an $O(n^3)$ algorithm.
    - Ex If n=10000, then $n^3 = 10^{12}$, but $3^n = 1.6 \times 10^{4771}$.

The class NP

- NP = Nondeterministic polynomial time.
- Def An instance of a problem consists of an input for the problem.
  - Ex An instance of the sorting problem is a set {3,1,2,4} that we want to sort.
  - Ex An instance of the SSSP problem is a weighted graph along with a source node.
- P is the class of problems for which all instances can be solved in polynomial time by some algorithm.
- NP is the class of problems for which the solvability of an instance can be verified in polynomial time.
  - The verification is done by a "verifier" algorithm.
  - The verifier needs an additional "hint" to work correctly.
    - The hint is also called a "witness" or "certificate".
  - The verifier doesn't find a solution to a problem instance, but only checks that the instance has been solved.

The class NP

- The verifier has the following properties.
  - The verifier's input is a problem instance x, and a certificate y.
  - The verifier's output is either "accept" or "reject".
  - If x has a solution, then if y is a "good" certificate, the verifier will output accept.
    - If y is not a "good" certificate, the verifier can either accept or reject.
  - If x has no solution, the verifier rejects no matter what y is.
  - Intuitively, the certificate y indicates x is solvable.
    - For example, y can be a solution to x.
    - But y can also be an indirect representation of a solution.
  - The verifier is efficient, i.e. runs in polynomial time.

The class NP, formally

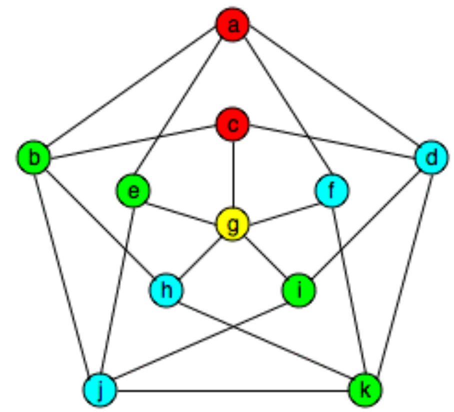
- Def A decision problem is a problem with a yes / no answer.
  - Ex Given a graph, is there a path from node s to t?
  - Ex Given a map, is there a way to 3-color it?
  - Ex Given a number, is it prime?
- Def Given a decision problem, the set of yes (resp. no) instances are the instances of the problem for which the answer is yes (resp. no).
  - Ex 11 is a yes instance to the prime problem, 10 is a no instance.
- Def Given a decision problem A, a polynomial time verifier V for A is an algorithm that does the following
  - V's input is an instance x of A, and a certificate string y.
  - $V(x, y) \in \{0,1\}$, representing "reject" and "accept", resp.
  - If x is a yes instance, there exists a y for which V outputs 1, i.e. $\exists y: V(x, y) = 1$.
  - If x is a no instance, every y makes V output 0, i.e. $\forall y: V(x, y) = 0$.
  - V runs in polynomial time.
- NP is the set of all decision problems with polytime verifiers.

Showing a problem is in P or NP

- To show a problem is in P, give an algorithm solving the problem that runs in polynomial time.
- To show a decision problem is in NP, give a polynomial time verifier for the problem satisfying the properties on the previous slide.
  - This requires specifying what the cerficates are, and how the verifier operates, given an instance of the problem and a certificate.

4-coloring is in NP

- Given a graph, can we assign each vertex one of 4 colors, such that adjacent vertices have different colors?
- Verifier
  - Certificate y is an assignment of colors to the vertices of graph x.
  - Check y uses at most 4 colors. If not, output no.
  - Go through all edges of x, and checks endpoints of each edge have different colors.
  - If true for all edges, output 1. Else output 0.
- If x has solution
  - Then x is 4-colorable.
  - So there's way to assign each vertex one of 4 colors s.t. endpoints of each edge have different colors.
  - Let y be this assignment, and give y to V.
  - Clearly V outputs 1.
- x has no solution
  - Then x is not 4-colorable.
  - So no matter how we assign 4 colors to vertices of x, some edge has endpoints with the same order.
  - So V outputs 0, for any input y.
- V runs in polytime.
  - If x has n vertices, then it has $O(n^2)$ edges, so V runs in $O(n^2)$ time.

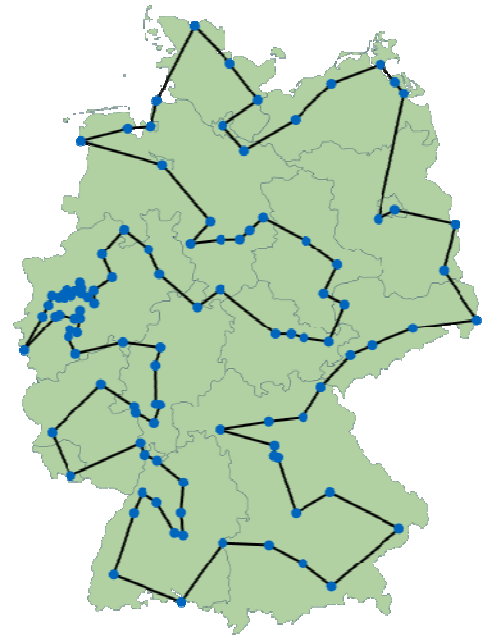


Factoring is in NP

- Given an integer x, does it have a factor y ≠ 1,x.
- Verifier
  - Certificate y is a number.
  - Check y divides x, and y ≠ 1,x.
  - If so, output 1, else output 0.
- If x has a solution.
  - Then x has a nontrivial factor y.
  - Give y to V, and V outputs 1.
- If x has no solution.
  - Then every factor of x is either 1 or x.
  - So for any y ≠ 1,x given to V, V outputs 0.
- V runs in polytime.
  - Dividing x by y takes polynomial time.
- However, factoring does not seem to be in P.
  - Given an n digit number, there's no known way determine if it has a nontrivial factor in $O(n^k)$ time, for any constant k.

999999866000004473 =
999999929 x 999999937

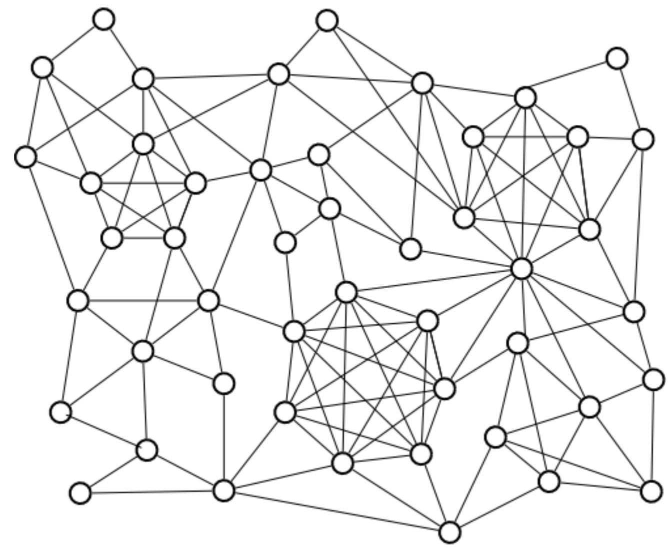
Traveling salesman is in NP



- Given a set of n cities, and distances between each pair of cities, is there a path visit each city exactly once, and has distance at most D, for a given D?
- Verifier
  - Certificate y is a path through the graph.
  - Check y goes through every vertex once, and total length of y is $\leq D$. If so, output 1, else output 0.
- If x has a solution.
  - Then there is a path going through each vertex once with total length $\leq D$.
  - Call the path y and give it to V.
  - Clearly V outputs 1.
- If x has no solution.
  - Then no matter what path y you use, either y doesn't go through each city once, or y has length > D.
  - So V outputs 0, no matter what y it gets.
- V runs in polytime.
  - If the graph has n vertices, then all of V's checks can be done in O(n) time.

k-Clique is in NP

- Given a graph with n nodes and a number k, are there k nodes that form a clique, i.e. that are all connected to each other?
- Verifier
  - Certificate y is a set of k nodes in x.
  - Check each pair of the k nodes is connected by an edge. If so, output 1. Otherwise output 0.
- If x has a solution.
  - Then there are k nodes that are mutually connected.
  - Call this set y and give it to V.
  - Clearly V outputs 1.
- If x has no solution.
  - Then in any set of k nodes, some 2 nodes aren't connected.
  - So V outputs 0, no matter what set of k nodes it gets.
- V runs in polytime.
  - Checking k nodes are mutually connected takes $O(k^2)$ time.



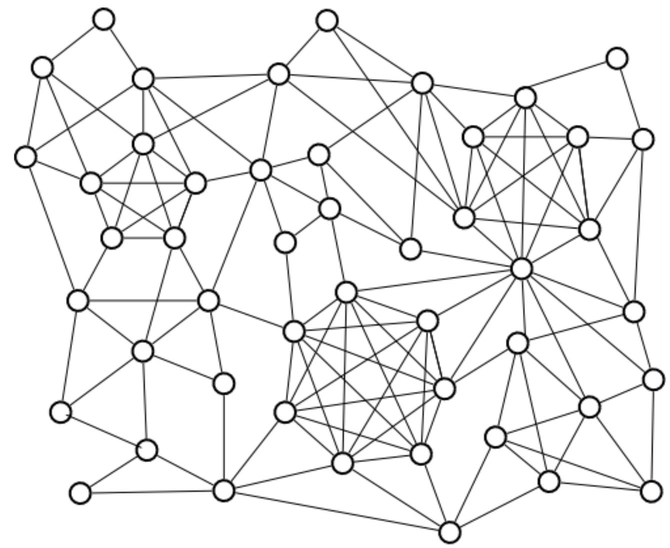
All problems in P are in NP

- Let A be a problem in P. I.e. there's a polytime algorithm S s.t. on every instance x of A
  - If x has a solution, S returns a solution.
  - If x has no solution, S returns fail.
- Verifier
  - V runs S. If S finds a solution, V outputs 1. Otherwise V outputs 0.
- If x has a solution.
  - S finds a solution, so V outputs 1.
- If x has no solution.
  - S returns fail, so V outputs 0.
- V runs in polytime.
  - Because V just runs S, which runs in polytime.
- Notice that for problems in P, V doesn't need a certificate y.
  - For problems in P, it's easy to determine if they're solvable or not.
- But for hard problems (not in P), V isn't powerful enough to determine solvability by itself.
  - So it needs a hint / witness / certificate.
- Ex In factoring, a polytime verifier isn't powerful enough to find a nontrivial factor of an input.
  - But if it's given a nontrivial factor, it can check the factor works in polytime, and therefore verify the input is composite.

Primes is in NP

- Proving a problem is in NP isn't always so easy...
- Given a number x, is x prime?
- Verifier
  - What should the certificate y be?
  - If y is a single number s.t. $y \nmid x$, then y doesn't certify that x is prime.
  - Suppose y a vector giving $\frac{x}{y}$ for every $y \leq \sqrt{x}$.
  - V returns 1 if all these aren't integers, and 0 otherwise.
- If x has a solution.
  - I.e., x is prime. Then all the quotients are non-integer, so V returns 1.
- If x has no solution.
  - Then x is composite, so x has a factor $y \leq \sqrt{x}$, so $\frac{x}{y}$ is integer, and V outputs 0.
- V runs in polytime.
  - No it doesn't!
  - Say x has n digits. Then there are ~ $2^{n/2}$numbers $\leq \sqrt{x}$, so y has size $O(n2^{n/2})$.
  - Since V has to check all values in y, it doesn't run in poly(n) time.
- So, this verifier is incorrect. This verifier does not show Primes is in NP.
- That doesn't mean Primes $\notin$ NP, it just means our verifier doesn't work.
- We can show Primes is in NP using another verifier and some number theory. This is called Pratt's Theorem, and is beyond our scope.

Incorrect verifiers



- We showed k-Clique is in NP by giving a correct verifier.
- Let's see some incorrect verifiers.
  - None of these verifiers can be used to prove k-Clique is in NP.
- Verifier 1 Always outputs 1, regardless of y.
  - Wrong, because when graph doesn't contain a k-clique, V is supposed to output 0.
- Verifier 2 Always output 0, regardless of y.
  - Wrong, because when the graph does contain a k-clique, V is supposed to output 1, for some y.
- Verifier 3 Check all subsets of k nodes. If any form a clique, output 1, else output 0.
  - Seems OK. When x has a k-clique, V outputs 1, and when x doesn't, it outputs 0.
  - But V is still wrong, because it doesn't run in polytime.
    - There are $O(n^k)$ subsets of k nodes, and V checks all of them.

P vs NP

- Does P=NP?
  - I.e. suppose there's a problem for which we can verify solvability in polynomial time. Does that mean we can actually find a solution in polynomial time?
- This is the arguably the most important question in computer science.
  - The other would be to produce general AI.
- Many real-world problems are in NP. If P=NP, we can solve them efficiently. If P≠NP, then we can't.
- Every P problem is in NP, as we saw. So $P \subseteq NP$.
- Is every NP problem in P, i.e. $NP \subseteq P$?
- After 50 years, nobody knows.
  - Most, but not all researchers think not all NP problems are in P.
  - There are probably problems we can efficiently verify but not efficiently solve.
  - Ex Factoring is something we can efficiently verify, but not solve.
- If you can prove $P \neq NP$, or even better, $P = NP$, then
  - you ≥ Newton ≥ Einstein ≥ …
  - You also get $1M from the Clay Math Institute.
- Answering this question has vast and profound implications for CS, AI, math, physics, etc.

NP-completeness

- Out of all the NP problems, there’s a subset of NP problems called NP-complete (NPC) problems that are the “hardest” NP problems.
- To determine whether P=NP, it suffices to know whether P=NPC.
  - If the hardest problems can be solved in polytime, then all NP problems can be solved in polytime. I.e. P=NP.
- So the study of P vs NP focuses on NPC problems.

Hardness and reductions

- What does it mean to say problem B is harder than problem A?
- It means if you can solve B, you can also solve A.
  - Ex Algebra is harder than arithmetic, because if you can do algebra, you can also do arithmetic.
  - So if I have an algorithm for solving B, I can use it to solve A.
- We say A reduces to B.
  - Write $A \leq_R B$.
  - Read this as "A is equally or less difficult than B".

Example

- FACTOR-ALL(n) finds all the factors of a number n.
- FACTOR-1(n) finds one factor.
- Of course, FACTOR-1 $\leq_R$ FACTOR-ALL.
  - If we can find all the factors, we can certainly find one.
- FACTOR-ALL $\leq_R$ FACTOR-1.
  - We use FACTOR-1(n) to find one factor m of n.
  - Then divide n by m, and run FACTOR-1 on the result, to find another factor of n.
  - Keep repeating the previous steps until we get all the factors.
- The hard part about factoring a number, is just to find one factor.
  - Since FACTOR-1 $\leq_R$ FACTOR-ALL, FACTOR-ALL $\leq_R$ FACTOR-1, these problems have the same hardness.

Reductions, formally

- Let A and B be two decision problems.
- Let X and Y be the set of yes instances for A and B, resp.
- Ex Say A = PRIME and B = k-CLIQUE.
  - X is the set of prime numbers.
  - Y is the set of graphs containing a k-clique.
- Let f be a function that maps instances of A to instances of B.
- Def A reduces to B if there exists $f: A \rightarrow B$ s.t. for all instances x of A, $x \in X \Leftrightarrow f(x) \in Y$.
  - We write $A \leq_R B$.
- To show $A \leq_R B$, just give the mapping f.
- If $A \leq_R B$, then we can use an algorithm for B to solve A.
  - To solve an instance of A, first map it to an instance of B using f.
  - Then run the B algorithm.
  - Return the same answer for A as the B algorithm gives.
  - By definition, A is true ⇔ f(A) is true.

Example

- Suppose we want to show PRIME $\leq_R$ k-CLIQUE.
- This means there's some mapping f such that.
  - Given an instance of PRIME, i.e. a number n.
  - f(n) is an instance of k-CLIQUE, i.e. f(n) is a graph G.
  - n is prime if and only if f(n) contains a k-clique.
- If we have an algorithm to solve k-CLIQUE, we can use it solve PRIME.
  - To tell if n is prime, map n to a graph G and run the k-CLIQUE algorithm on G.
  - If it returns true, n is prime. Otherwise n isn't.

Polynomial time reductions

- If the mapping function from A to B runs in polynomial time, then it's a polynomial time reduction, and we write $A \leq_P B$.
  - Ex If we're reducing PRIME to k-CLIQUE, then the function to generate a graph from a number must run in polytime.
- Thm 1 Let A, B and C be three problems, and suppose $A \leq_P B$ and $B \leq_P C$. Then $A \leq_P C$.
- Proof Since $A \leq_P B$, there's a polytime mapping f from instances of A to instances of B.
  - Since $B \leq_P C$, there's a polytime mapping g from instances of B to instances of C.
  - Given an instance X of A, let $Y = f(X)$, and $Z = g(Y) = g(f(X))$.
  - Then X is a yes instance of A ⇔ Y is a yes instance of B ⇔ Z is a yes instance of C.
  - So $g \circ f$ is a valid mapping of A to C.
  - Since f and g are both polytime, $g \circ f$ is also polytime.

NP-completeness

- Def A problem A is NP-complete (NPC) if the following are true.
  - $A \in NP$.
  - Given any other problem $B \in NP$, $B \leq_P A$.
- Thus, a NP-complete problem is an NP problem that can be used to solve any other NP problem.
  - It's a "hardest" NP problem.

NP-completeness and SAT

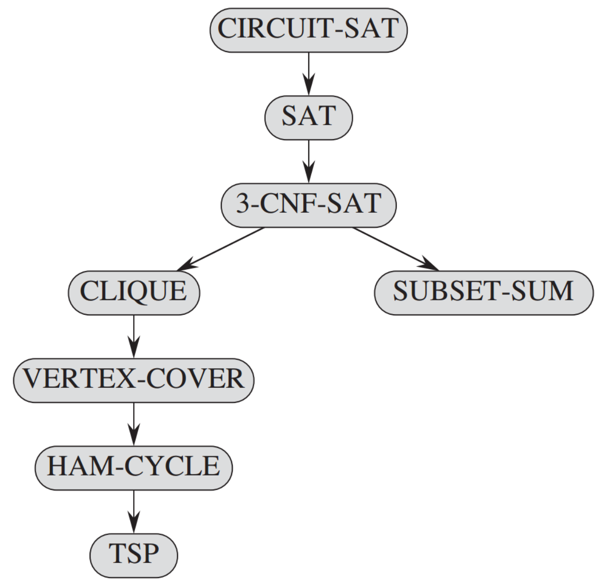
- Do NP-complete problems really exist?
  - Can we really find an NP problem that can be used to solve every other NP problem?
  - One problem to rule them all?
- Yes! Steve Cook and Leonid Levin proved around 1970 that SAT is NP-complete.
- SAT = satisfiable Boolean formulas.
  - Given a Boolean formula, is there any setting for the variables which makes the formula true?
  - Ex $(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee \neg B \vee \neg D) \in SAT$.
    - Setting A=B=C=true, D=false makes the formula true.
  - Ex $A \wedge \neg A \notin SAT$.
    - The formula's false for all settings of A.

NP-completeness and SAT

- Cook-Levin theorem says 2 things.
  - SAT $\in$ NP.
    - Prove this yourself.
  - Every NP problem reduces to SAT. I.e. every problem A in NP can be mapped to an SAT formula ϕ in polytime, such that
    - If A is true, then ϕ is satisfiable.
    - If A is false, then ϕ is not satisfiable.
- Basic idea of the theorem is to use the logical operations in a SAT formula to emulate the logical operations in any algorithm.
  - Any NP problem X has a polytime verifier V. The Cook-Levin theorem uses a SAT formula ϕ to emulate the verifier's operations.
  - For a yes instance of X, there's some certificate making V return 1.
    - The certificate can be transformed to a satisfying truth setting for ϕ.
  - Any certificate making V return 0 corresponds to a non-satisfying truth setting for ϕ.
  - So $\phi \in SAT$ if and only if X is a yes instance, and $X \leq_P SAT$.

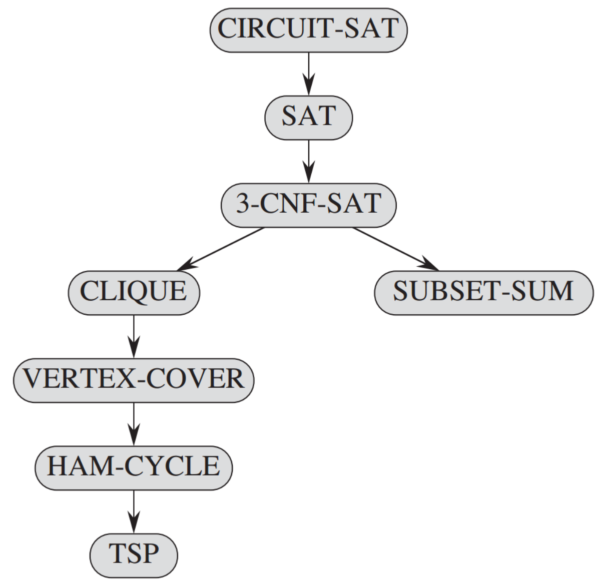
The web of NP-completeness

- For every problem in the picture, if A points to B, it means $A \leq_P B$.
  - So A can be solved using B.
- CIRCUIT-SAT was the original problem that Cook-Levin proved was NP-complete.
- So every problem in NP can be solved using CIRCUIT-SAT.
- But CIRCUIT-SAT can be solved using SAT, because CIRCUIT-SAT $\leq_P$SAT.
  - So every problem in NP can be solved using SAT.
  - So SAT is also NP-complete!
- SAT can be solved using 3-CNF-SAT.
  - So every NP problem can be solved using 3-CNF-SAT.
  - So 3-CNF-SAT is also NP-complete.
- All problems in the diagram are NP-complete.
- Of course, each of the reductions requires a proof, which is sometimes tricky.
  - We'll see some reduction proofs next lecture.
- There are thousands of other NPC problems.



The web of NP-completeness

- Thm Given two NP problems A and B, suppose A is NP-complete, and A $\leq_P$ B. Then B is also NP-complete.
- Proof Let C be any NP problem. Then C $\leq_P$ A, since A is NP-complete.
  - Since $A \leq_P B$, then by Theorem 1, we have $C \leq_P A \leq_P B$.
  - Since also $B \in NP$, then B is NPC.
- To prove a problem B is NP-complete
  - Take a problem A you know is NPC, and prove $A \leq_P B$.
  - E.g., A can be any problem in the previous diagram.
  - To prove $A \leq_P B$, you need to give a polytime reduction from A to B.
    - This can sometimes be quite challenging.
  - You also have to prove $B \in NP$, but that's usually not hard.



NP-completeness and P vs NP

- Thm 2 Suppose a problem A is NP-complete, and $A \in P$. Then P=NP.
- Proof Consider any other NP problem B. We'll show $B \in P$.
  - Since A is NPC, there's a polytime mapping f from B to A.
  - Given an instance X of B, run f on X to get an instance Y of A.
  - Since $A \in P$, there's a polytime algorithm g to solve A.
  - Run $g(Y)$, and return the same answer for X.
  - By the definition of $\leq_P$, $g(Y)$ is true $\Leftrightarrow$ X is true.
  - Running f and g both take polytime. So we can solve B in polytime.
- Cor Suppose a problem A is NP-complete, and $A \notin P$. Then for any NP-complete problem B, $B \notin P$.
  - If $B \in P$, then since B is NPC, we have P = NP by Theorem 2. So since $A \in NP$, we have $A \in P$, a contradiction.
- To prove $P \neq NP$ (which is what most people think), it's enough to show one NPC problem is not solvable in polytime, by the corollary.
  - But after 50 years, no one has any such proof.
  - Nor has anyone shown a polytime algorithm for any NPC problem.