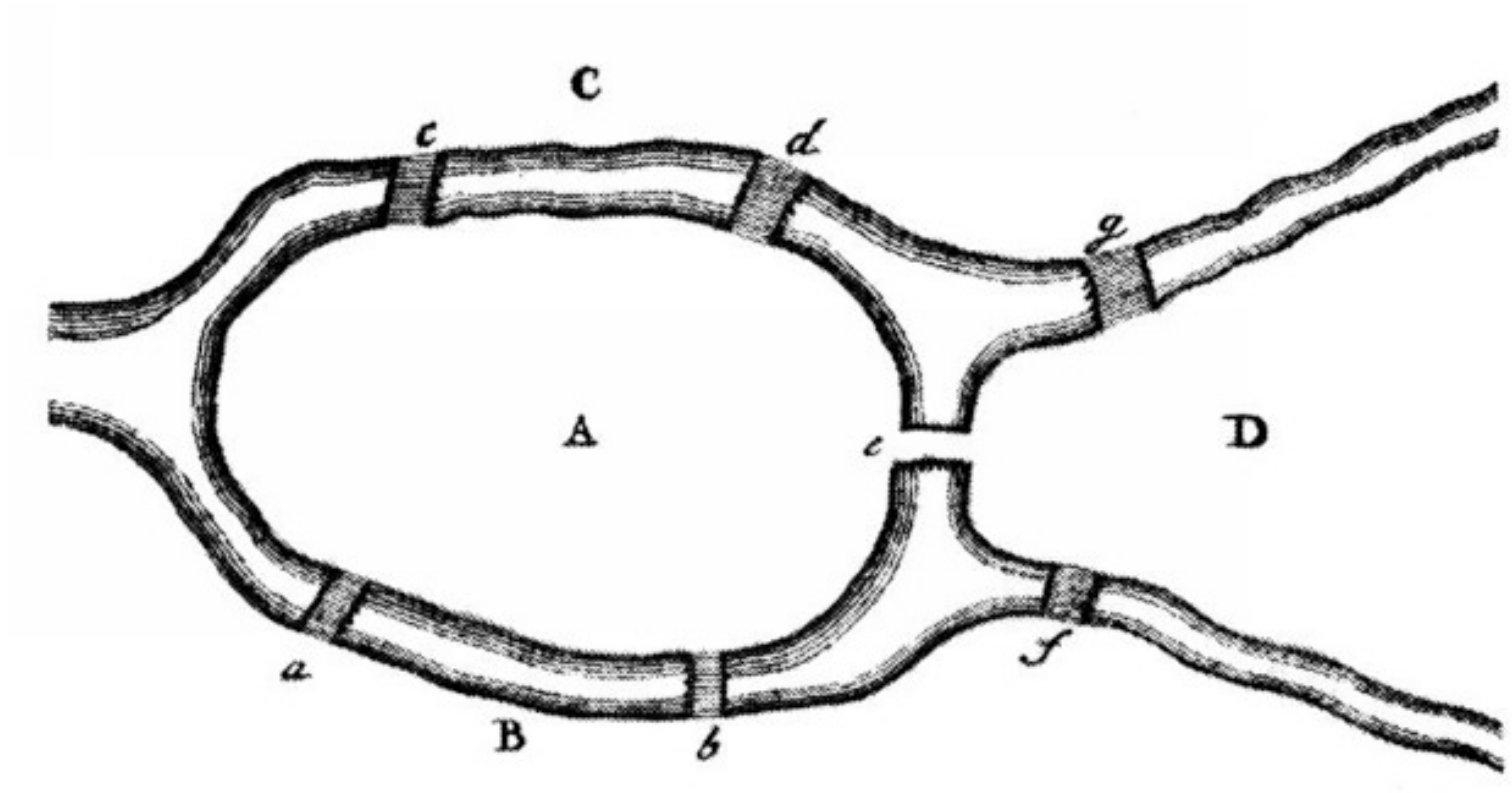
Computational Complexity

Cristopher Moore
Santa Fe Institute
Computer Science / Physics and Astronomy,
University of New Mexico

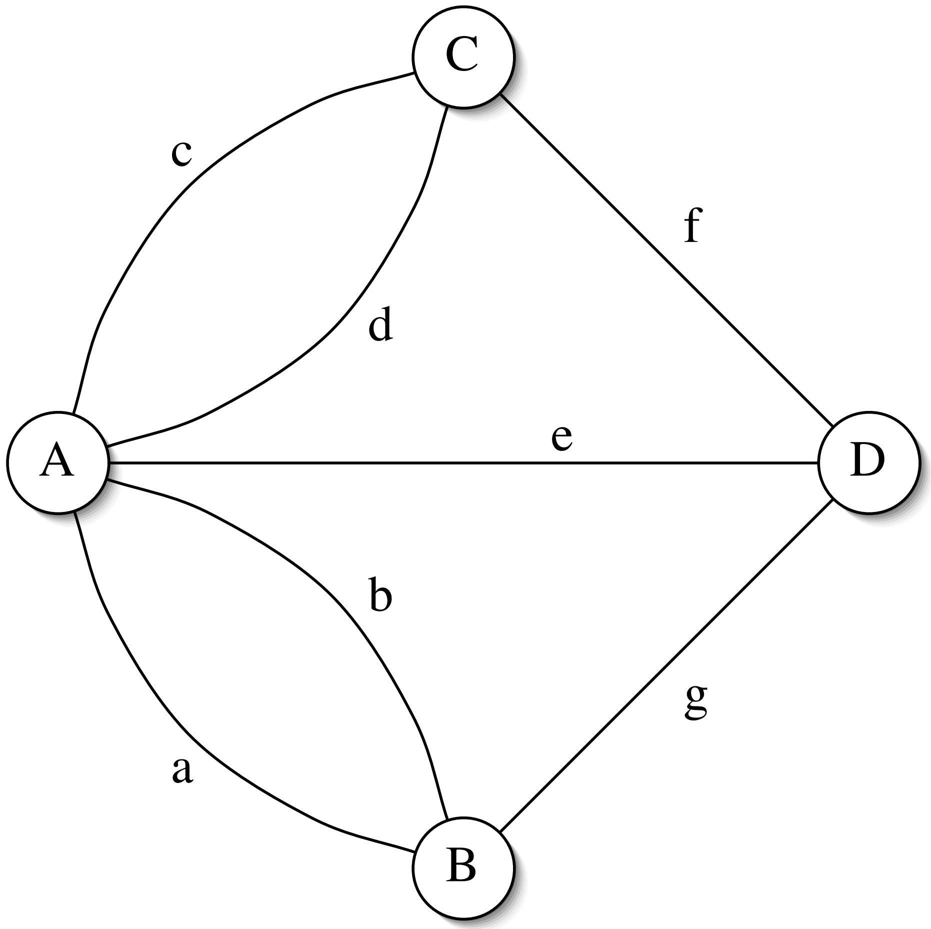
Computational complexity

Why are some problems qualitatively harder than others?



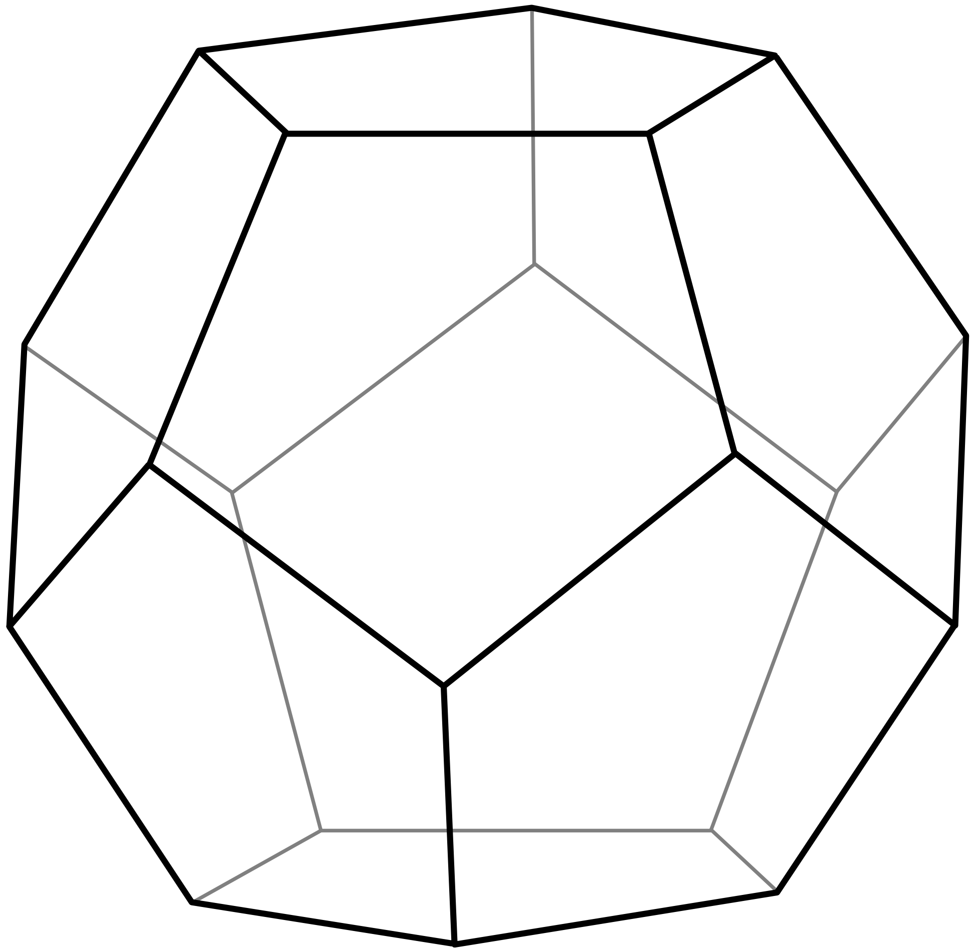
Computational complexity

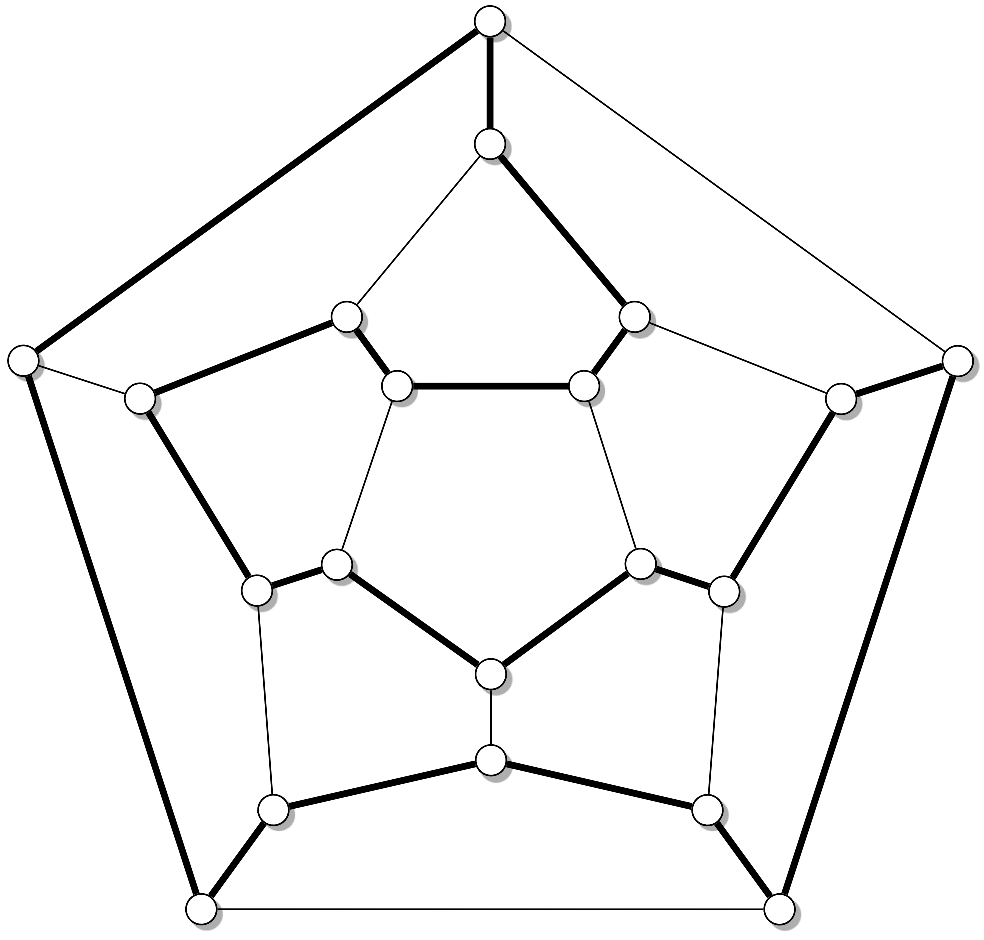
Euler: at most two nodes can have an odd number of bridges, so no tour is possible!



Computational complexity

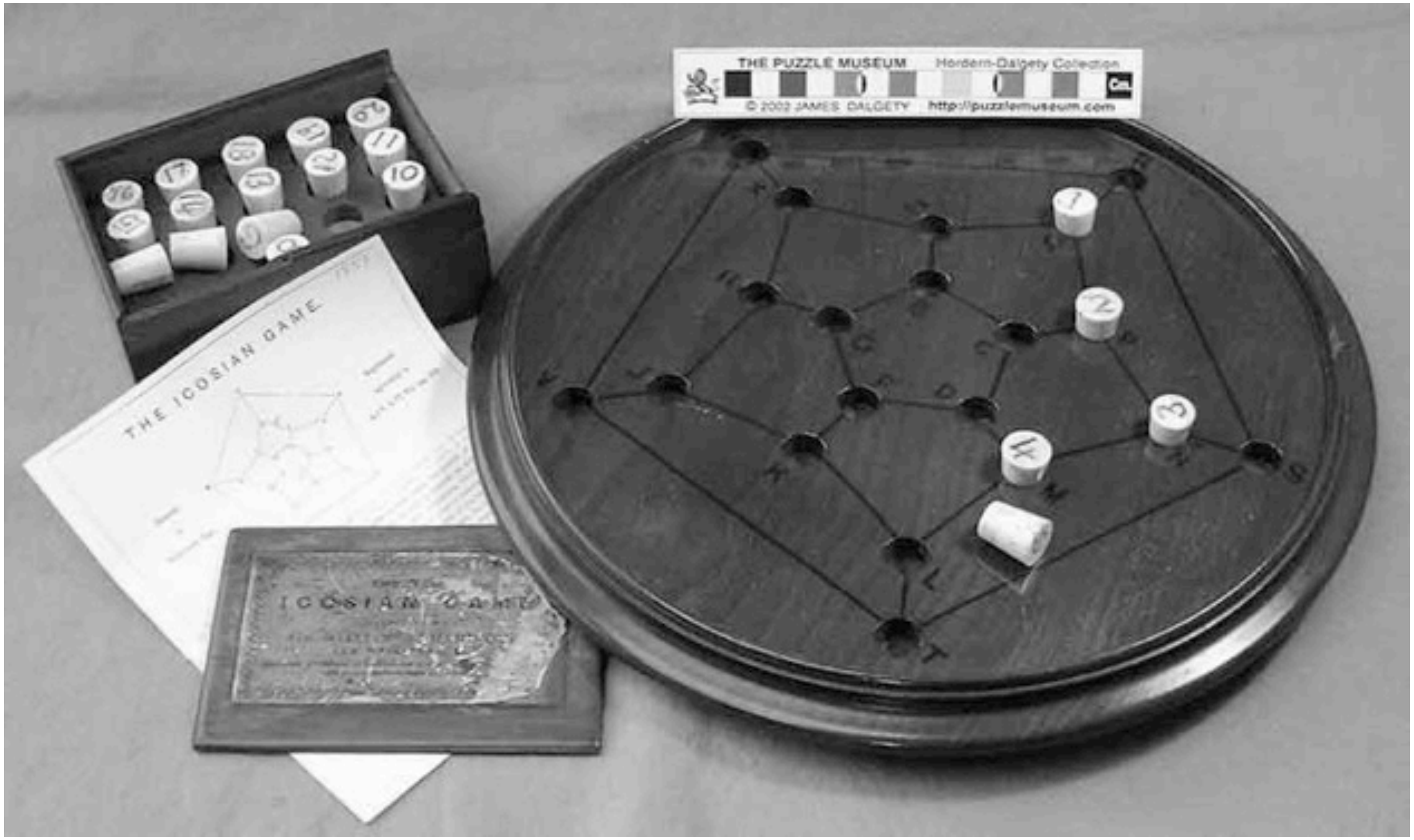
What if we want to visit every vertex, instead of every edge?





Computational complexity

As far as we know, the only way to solve this problem is (essentially) exhaustive search!

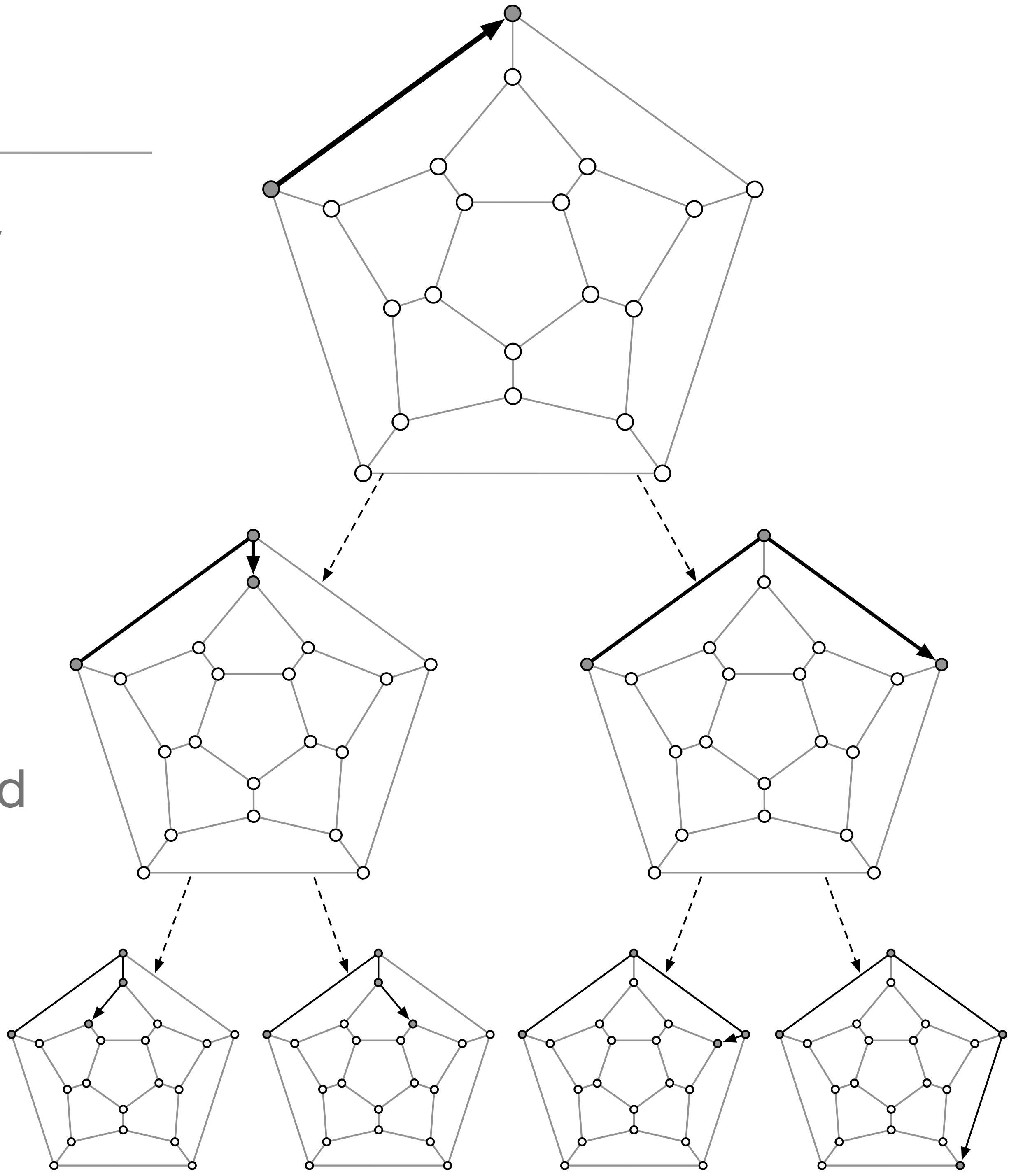


An exponential tree

Backtracking search: follow a path until you get stuck, then backtrack to your last choice

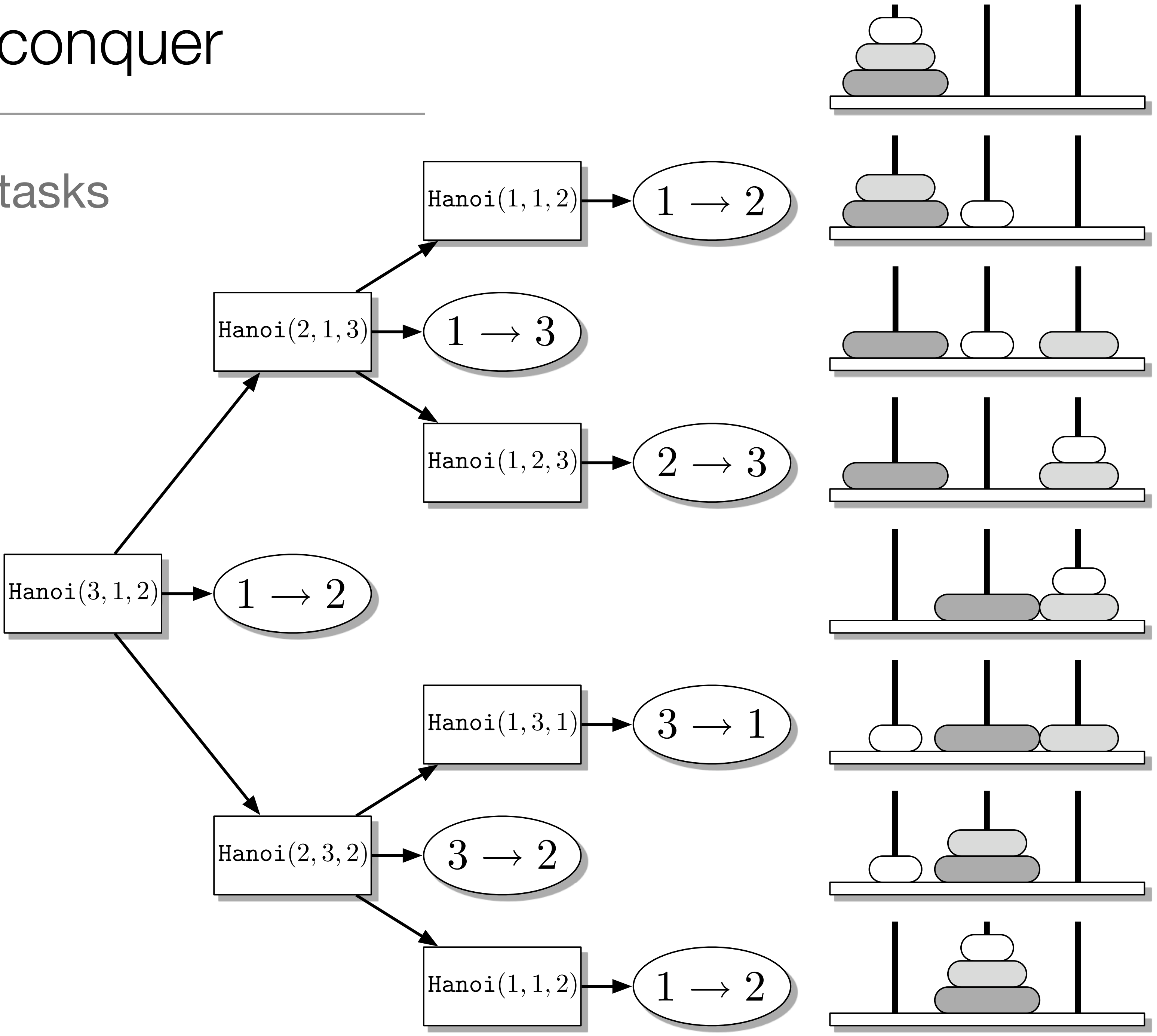
If there are *n* nodes, this could take 2^n time

When can we avoid this kind of search?



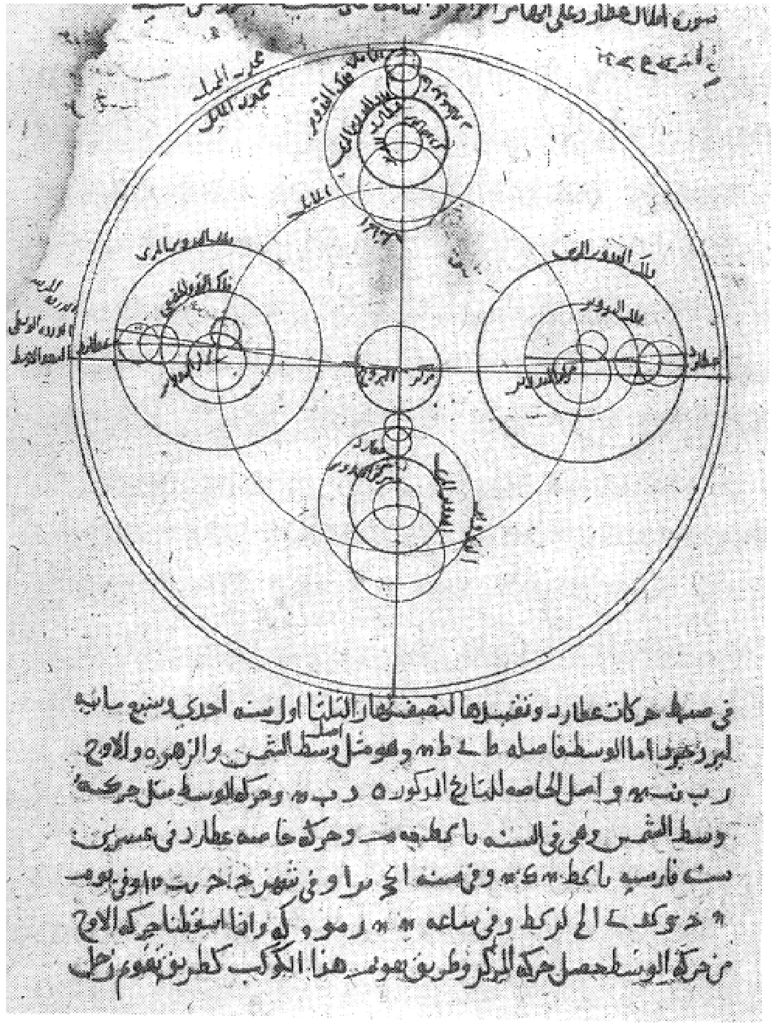
Divide and conquer

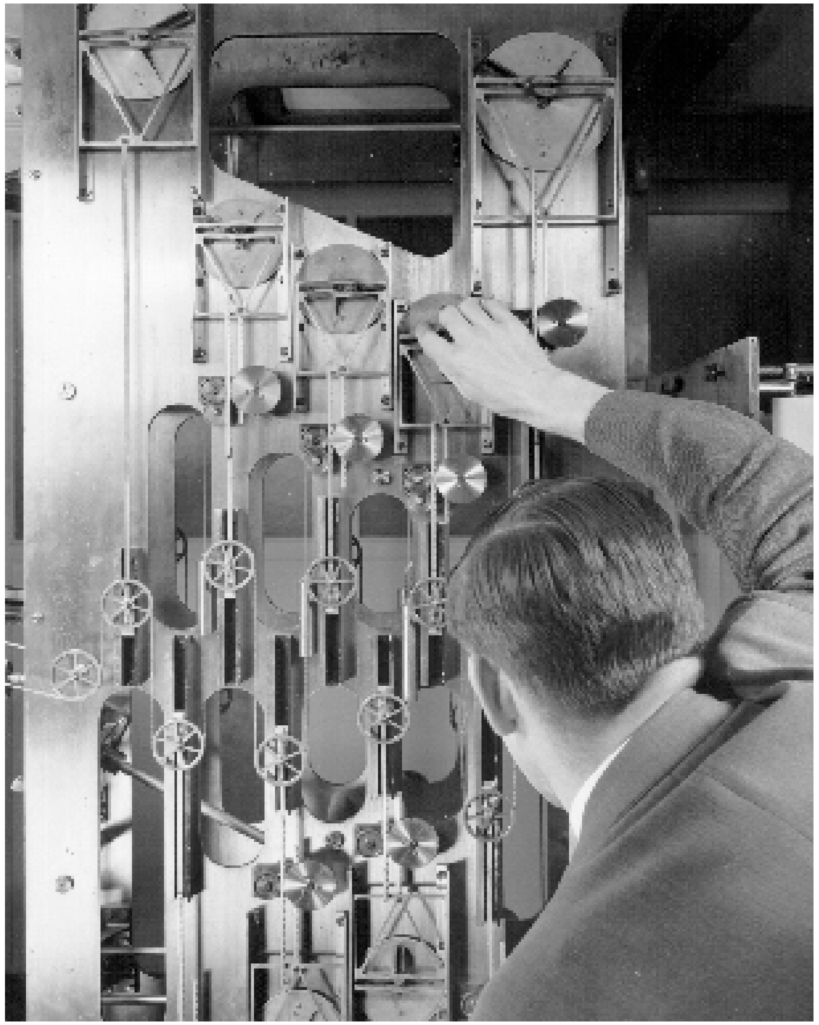
Tasks and subtasks



Divide and conquer

The Fast Fourier Transform

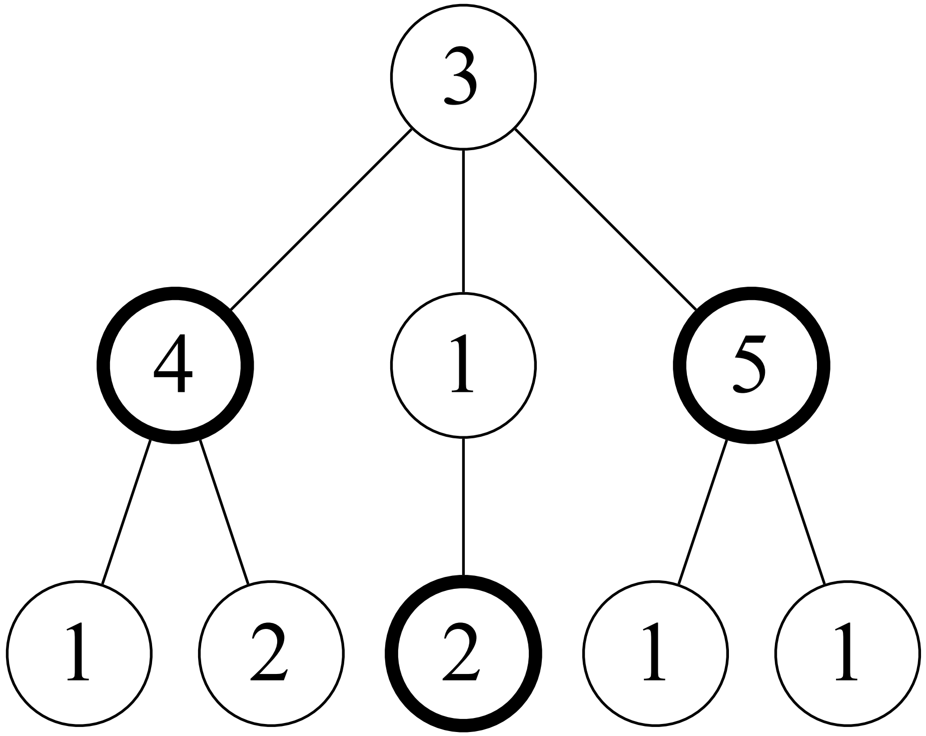


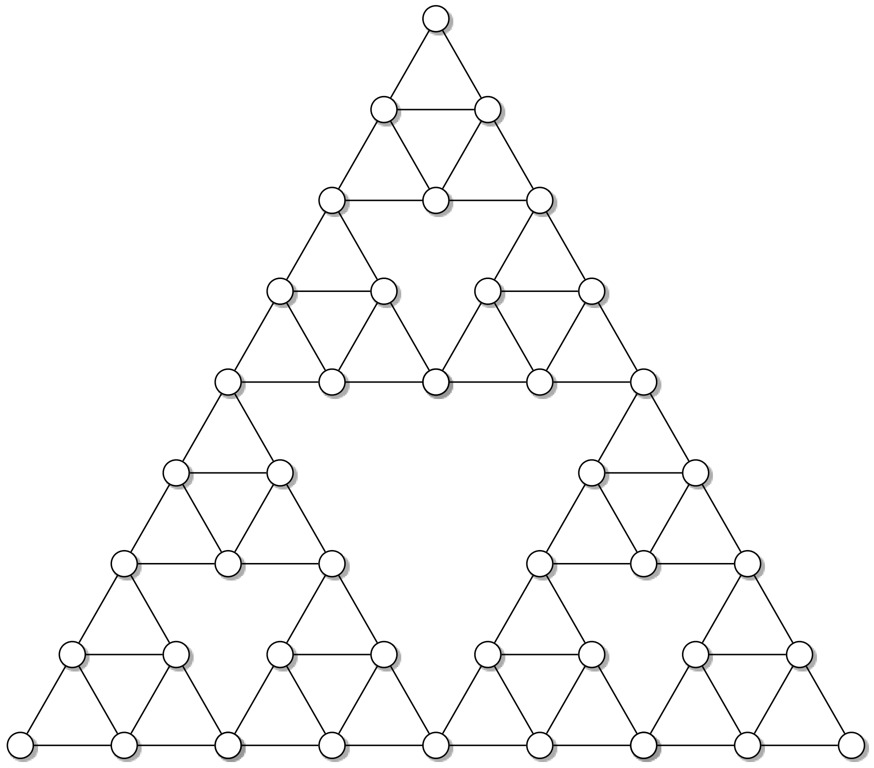


Divide and conquer, again

Dynamic programming: subproblems become independent after we make some choices

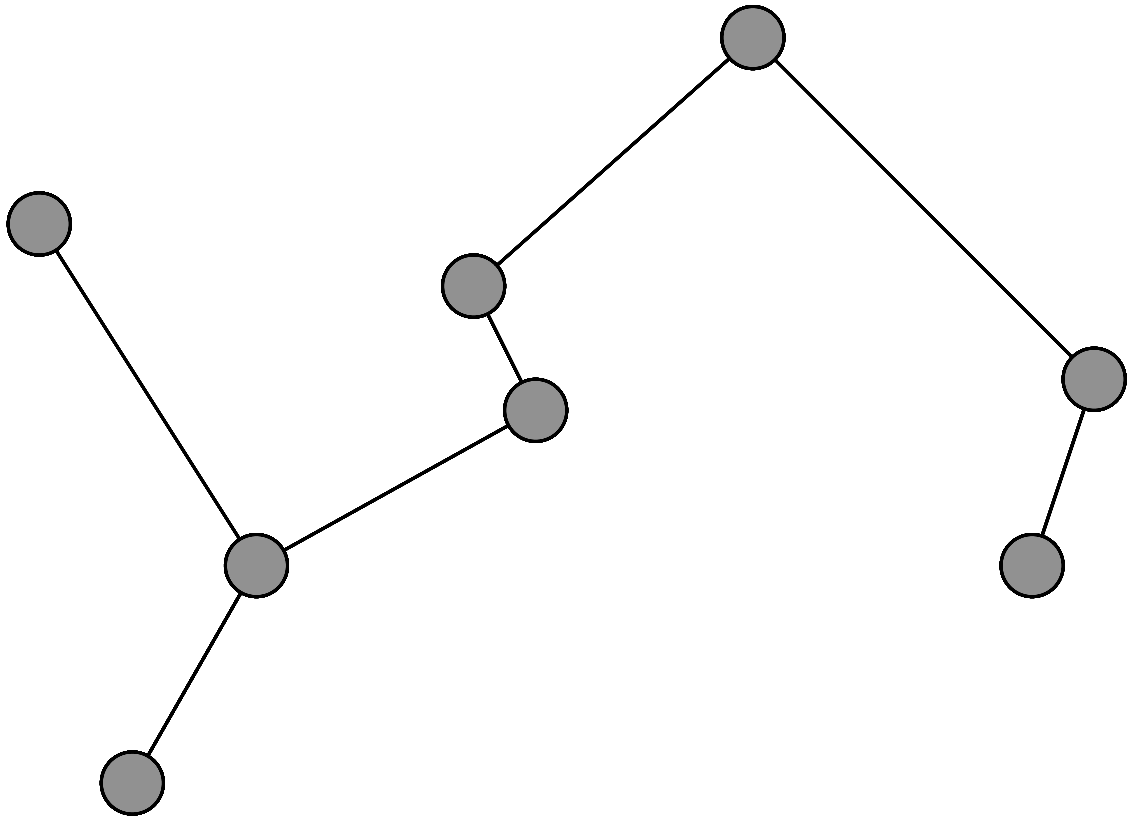
The “boundaries” between subproblems are small





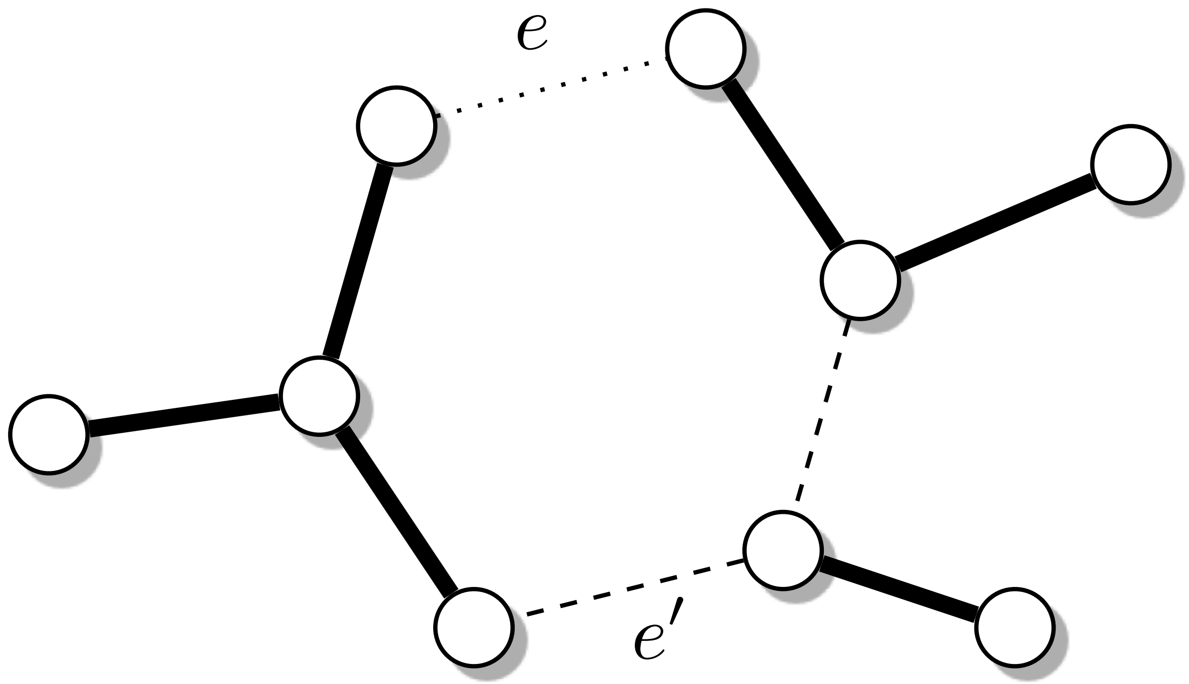
When greed is good

Minimum Spanning Tree (Boruvka, 1920): add the shortest edge



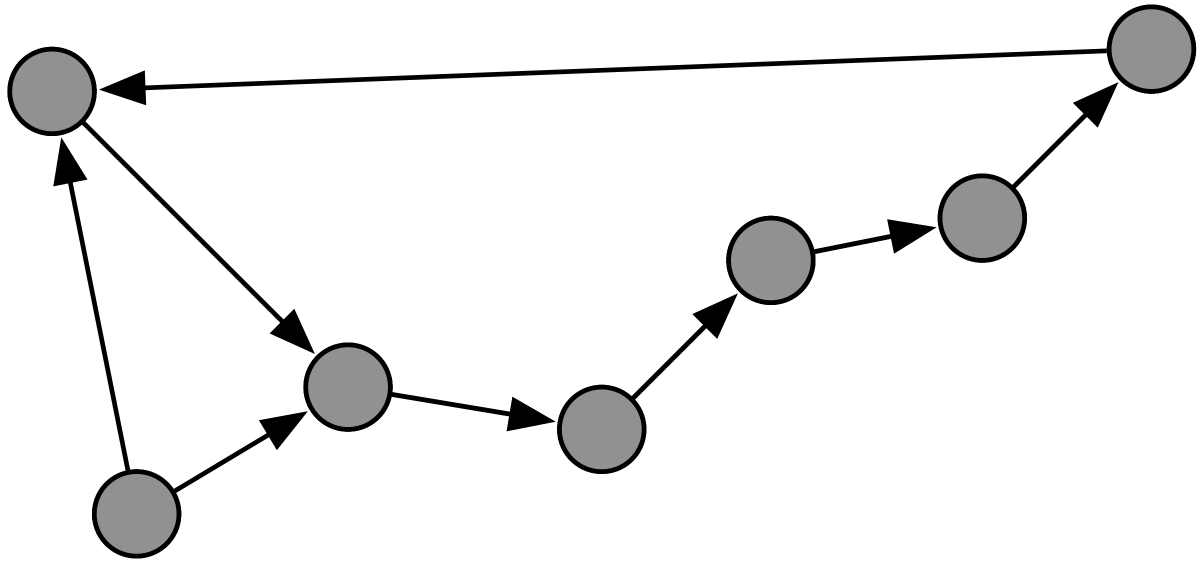
When greed is good

Doing the best thing in the short term can never lead us down the wrong path.



The primrose path

The Traveling Salesman Problem



Landscapes

A single optimum, that we can find by climbing:

Landscapes

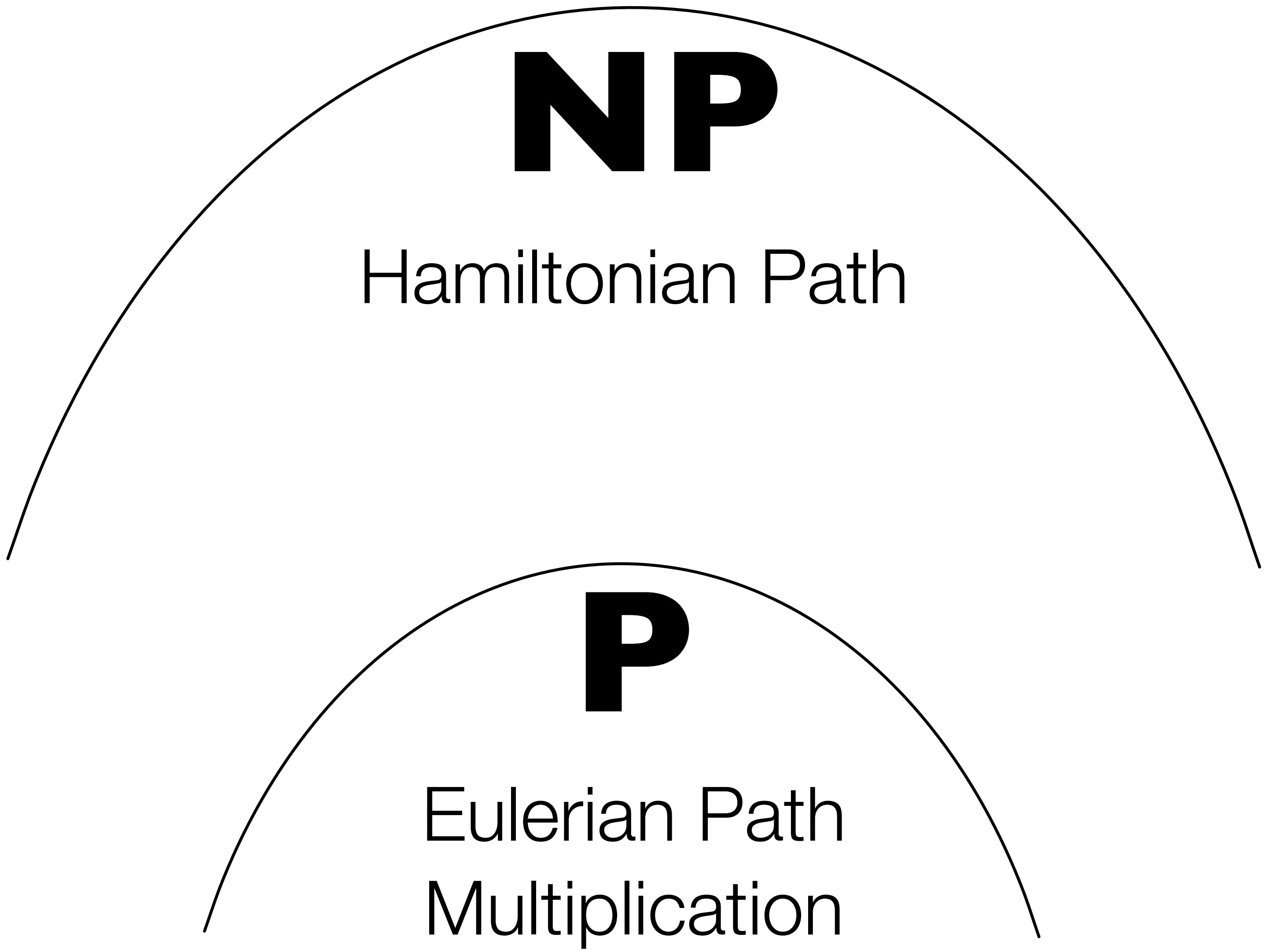
Many local optima where we can get stuck

Needles in haystacks

P: we can find a solution efficiently

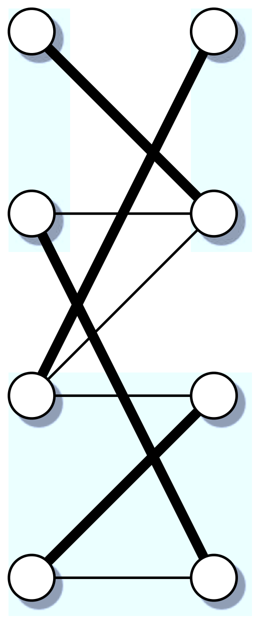
NP: we can *check* a solution efficiently

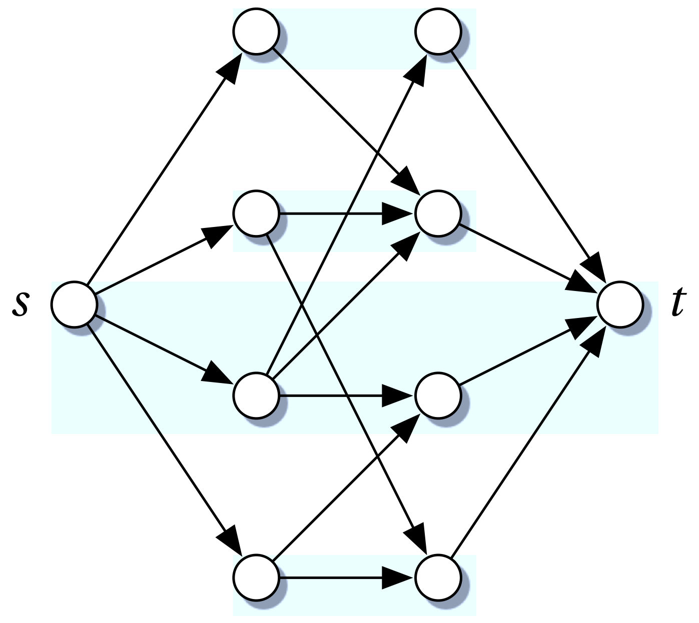
Complexity classes



“Reducing” one problem to another

Change a new problem into one we already know how to solve:





Bipartite Matching ≤ Max Flow

If Max Flow is easy, then so is Bipartite Matching

NP-completeness

Some problems B have the amazing property that *any* problem in NP can be reduced to them: $A \leq B$ for all A in NP

But if $A \leq B$ and A is hard, then B is hard too

So if P≠NP, then B can't be solved in polynomial time

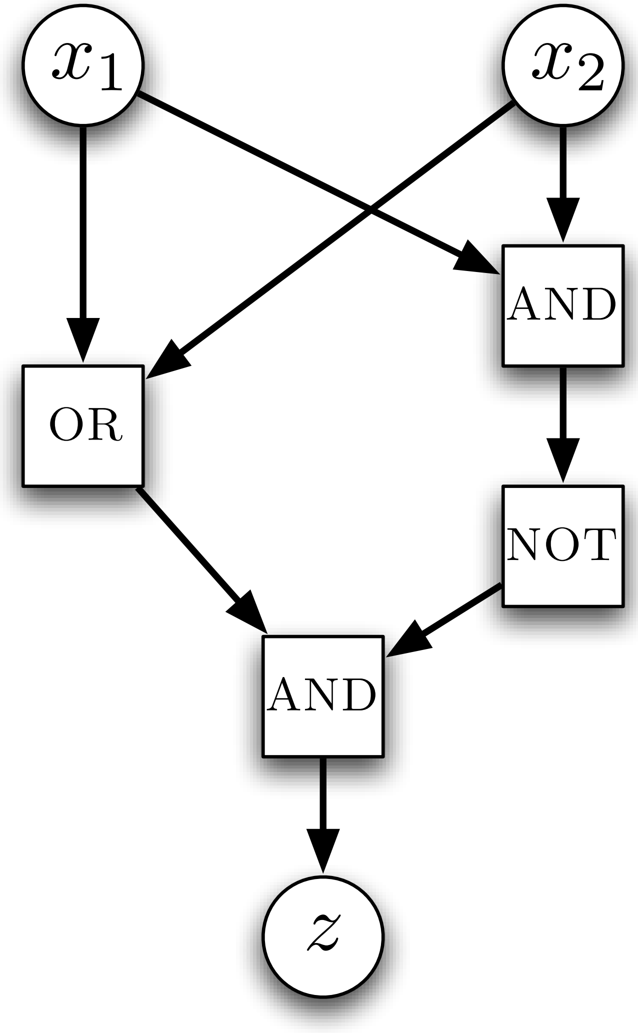
How can a single problem express every other problem in NP?

Satisfying a circuit

Any program that tests solutions (e.g. paths) can be “compiled” into a Boolean circuit

The circuit outputs “true” if an input solution works

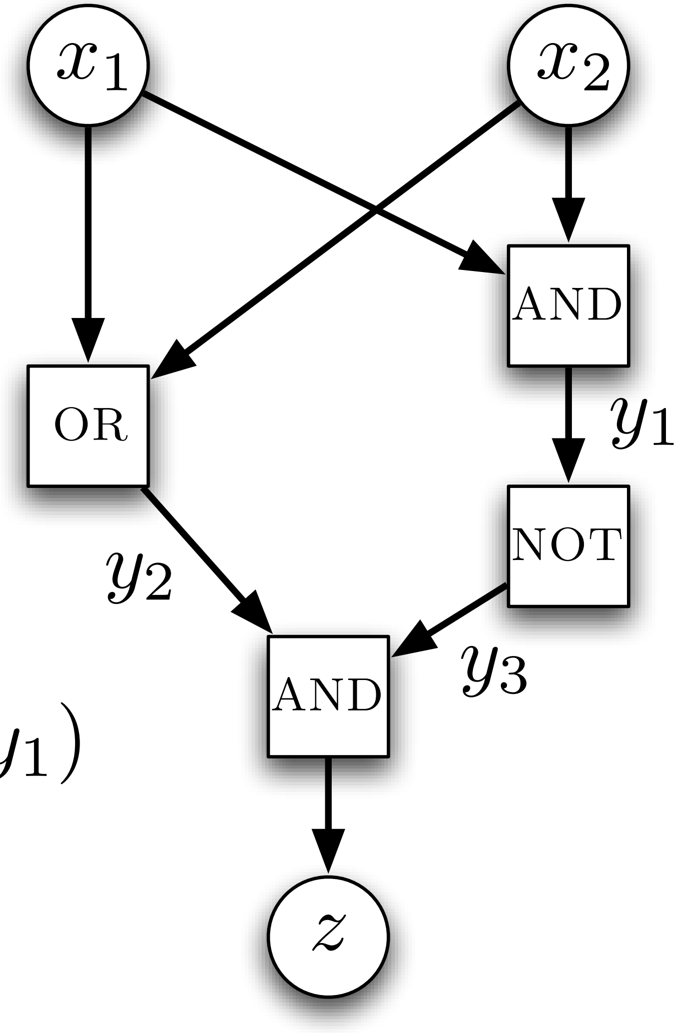
Is there a set of values for the inputs that makes the output true?



From circuits to formulas

The condition that each AND or OR gate works, and the output is “true,” can be written as a Boolean formula:

$$(x_1 \vee \overline{y}_1) \wedge (x_2 \vee \overline{y}_1) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee y_1)$$
$$\wedge \cdots \wedge z \quad .$$



3-SAT

Our first NP-complete problem!

Given a set of *clauses* with 3 variables each,

$$(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee x_{17} \vee \overline{x}_{293}) \wedge \cdots$$

does a set of truth values for the x_i exist such that all the clauses are satisfied?

k-SAT (with *k* variables per clause) is NP-complete for *k* ≥ 3

If 3-SAT were easy...

we could convert any problem in NP to a circuit that checks solutions,

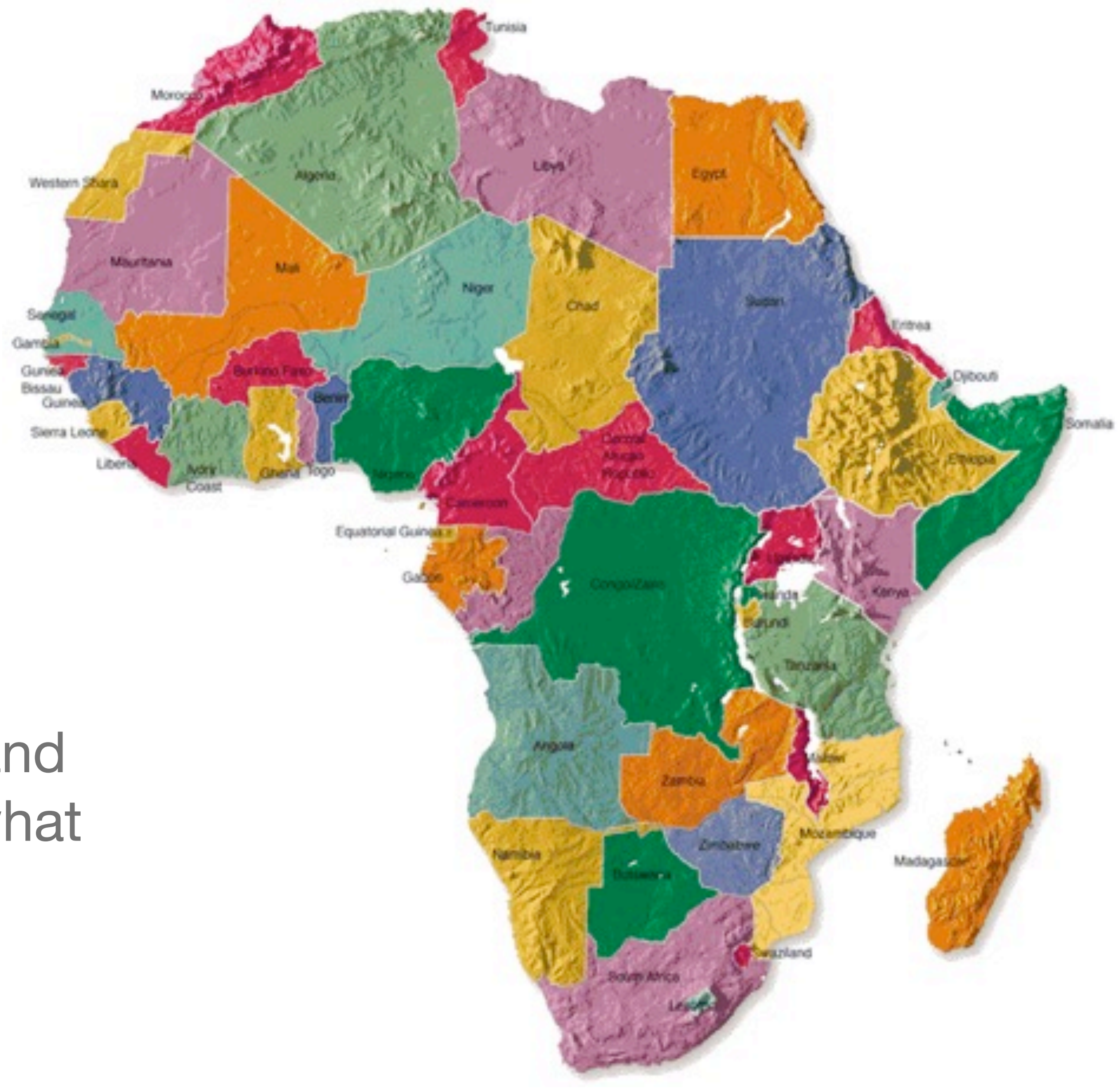
convert that circuit to a 3-SAT formula which is satisfiable if a solution exists,

and use our efficient algorithm for 3-SAT to solve it!

So, if 3-SAT is in **P**, then all of **NP** is too, and **P=NP**

Conversely, if P≠NP, then 3-SAT cannot be solved in polynomial time: something like exhaustive search is needed

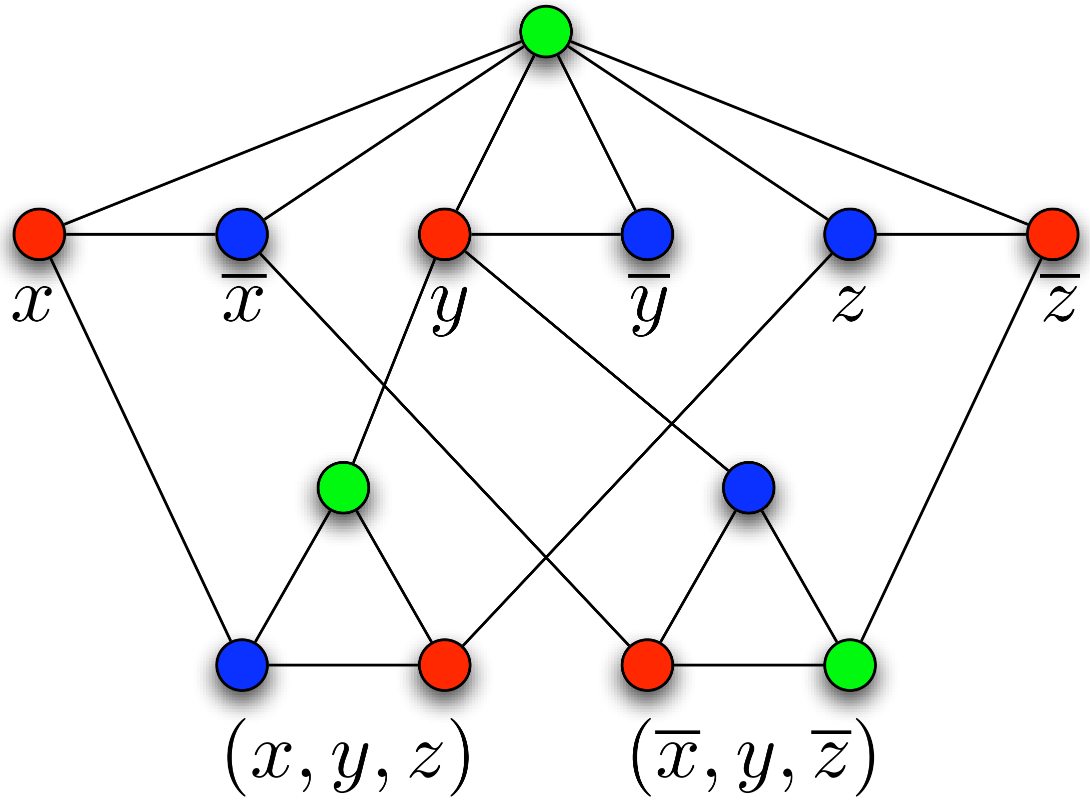
Graph coloring



Given a set of countries and borders between them, what is the smallest number of colors we need?

From SAT to Coloring

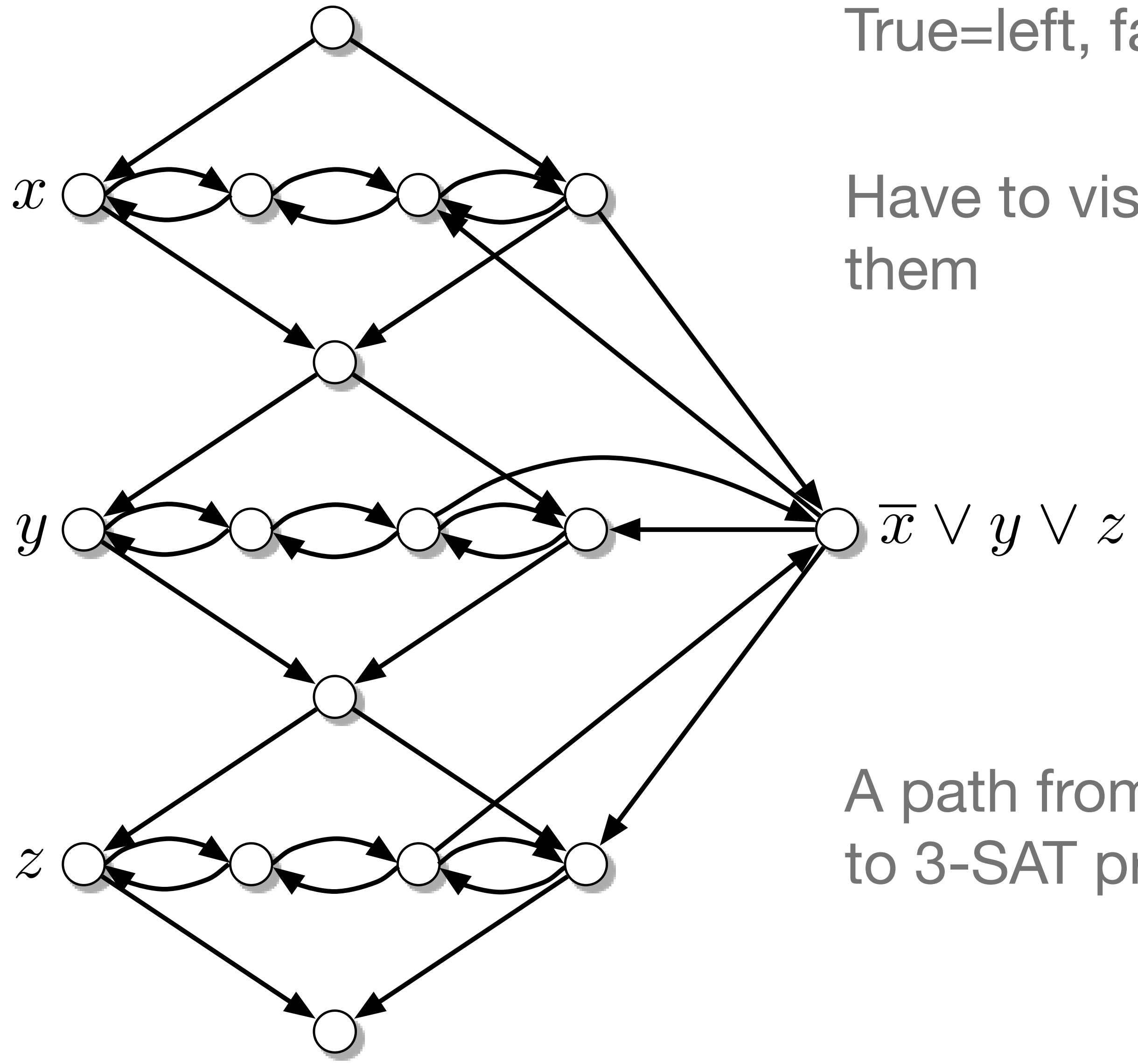
"Gadgets" enforce constraints:



Graph 3-Coloring is NP-complete

Graph 2-Coloring is in **P** (why?)

Traveling Salespeople

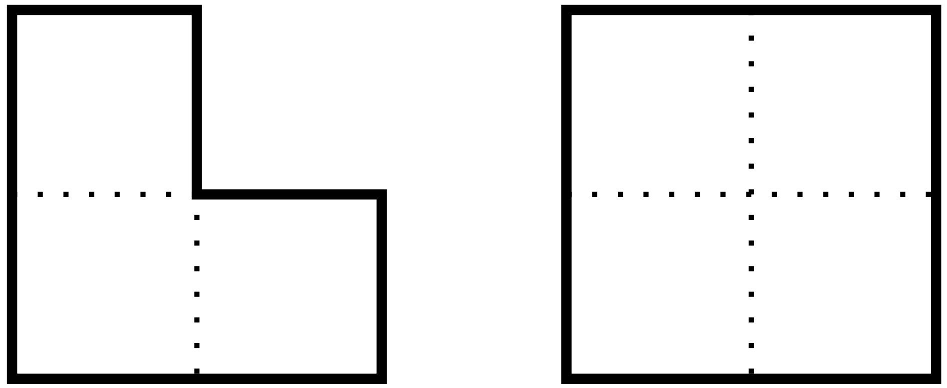


True=left, false=right

Have to visit clause vertices to satisfy them

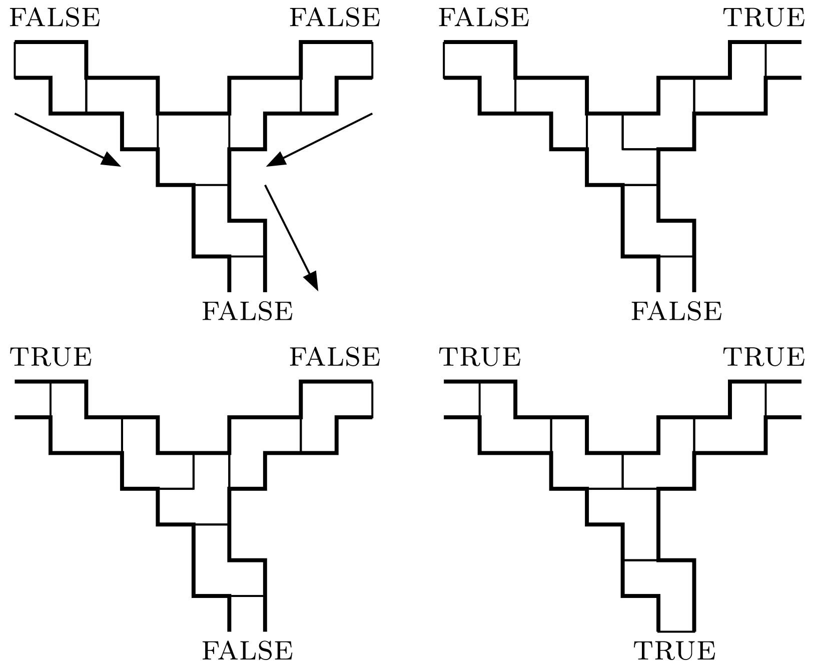
A path from top to bottom = solution to 3-SAT problem!

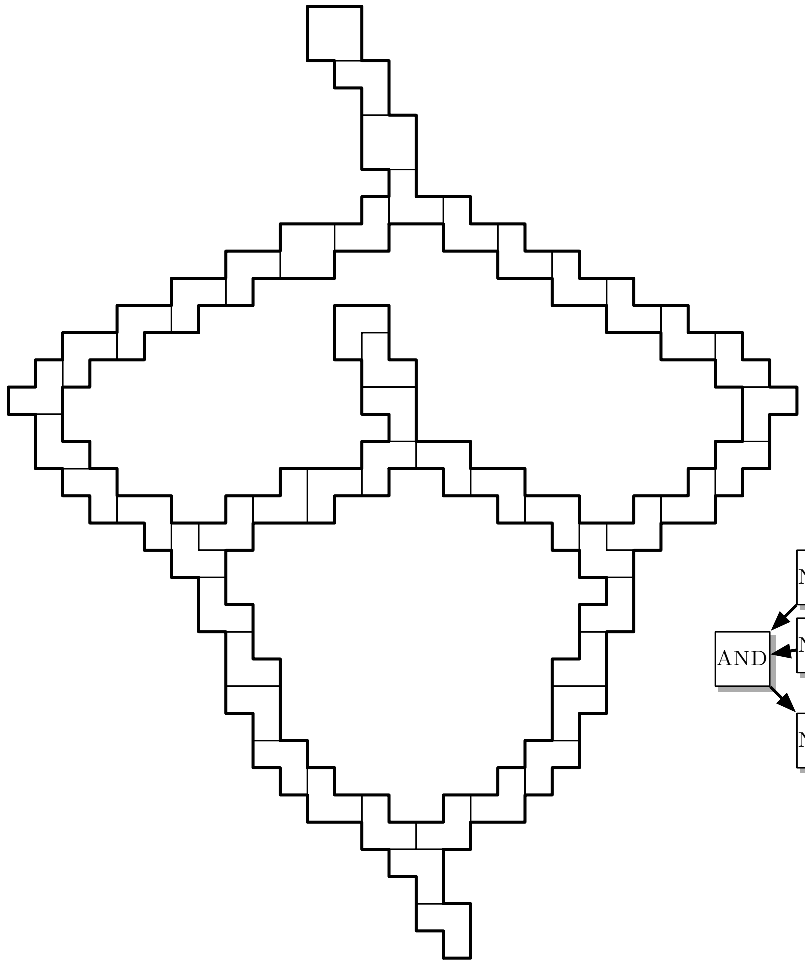
Why are some problems NP-complete?

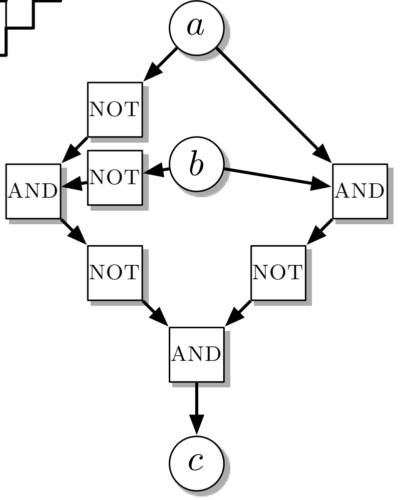


Can we tile a shape with these tiles?

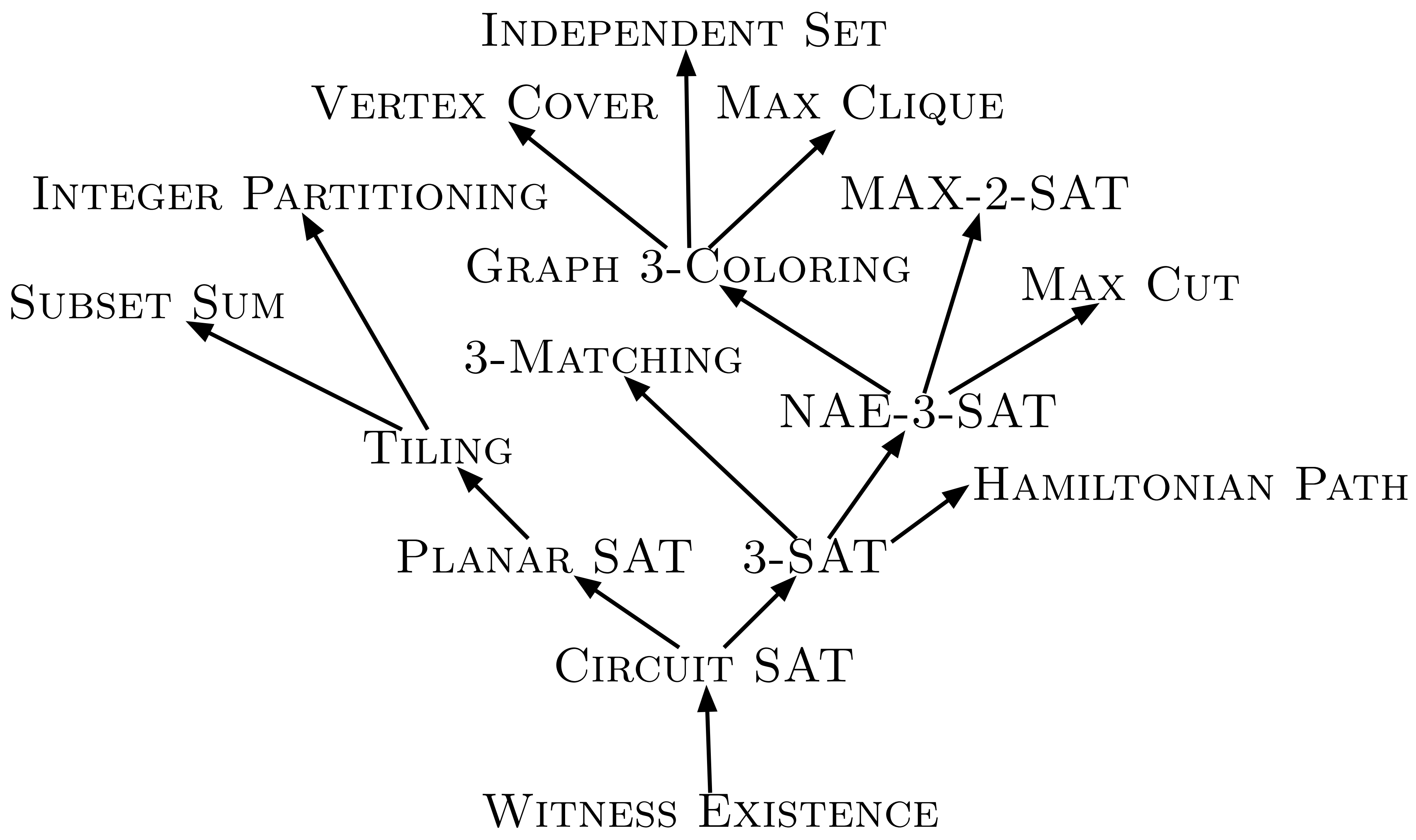
Because we can use them to build a computer.







Thousands of NP-complete problems



The P vs. NP problem

If any NP-complete problem can be solved in polynomial time, then they all can be, and P=NP

That would mean that *anything that is easy to check* — like a needle in a haystack — *is also easy to find.*

Better traveling salesmen; easier to tile bathroom floor, and pack our luggage; every cryptosystem can be broken.

But P vs. NP turns out to be question about the nature of mathematical truth and creativity.

Gödel to Von Neumann

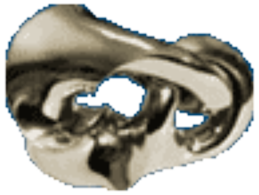
Let $\varphi(n)$ be the time it takes to decide whether a proof of length n exists. Gödel writes:

The question is, how fast does $\varphi(n)$ grow for an optimal machine. If there actually were a machine with, say, $\varphi(n) \sim n^2$, this would have consequences of the greatest magnitude. That is to say, it would clearly indicate that, despite the unsolvability of the *Entscheidungsproblem*, **the mental effort of the mathematician in the case of yes-or-no questions could be completely replaced by machines**. One would simply have to select an n large enough that, if the machine yields no result, there would then be no reason to think further about the problem.

P≠NP if we can't just "turn the crank"

P≠NP matters.

Millennium problems



P=NP?

Poincaré Conjecture

Riemann Hypothesis

Yang-Mills Theory

Navier-Stokes Equations

Birch and Swinnerton-Dyer Conjecture

Hodge Conjecture

Upper Bounds are Easy; Lower Bounds are Hard

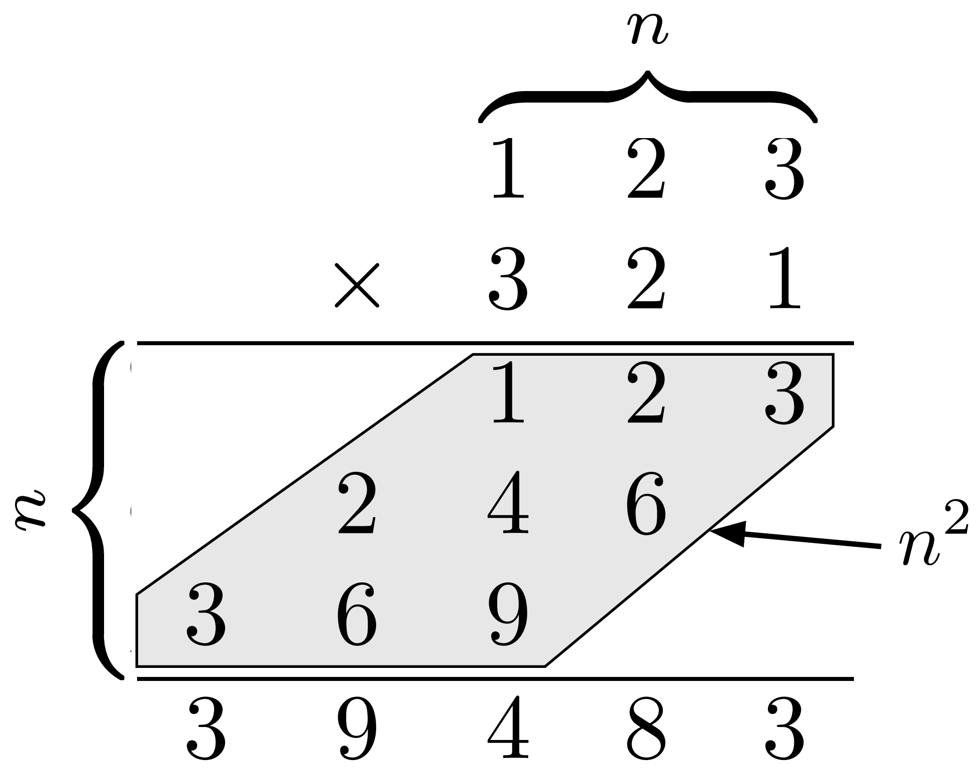
Why is the P vs. NP question so hard?

Algorithms are upper bounds on complexity...

...but how do you know if you have the best algorithm?

Algorithmic surprises

Grade school multiplication takes O(n^2) time:



Can we do better?

Algorithmic surprises

Divide and conquer:

$$x = 2^{n/2}a + b,\ y = 2^{n/2}c + d$$

$$xy = 2^n ac + 2^{n/2}(ad + bc) + bd$$

Looks like we need ac, ad, bc, bd . But

$$(a+b)(c+d) - ac - bd = ad + bc$$

so we only need three products. Running time:

$$T(n) \approx 3T(n/2)$$

so $T(n) \sim n^{\log_2 3} = n^{1.58}$. How low can we go?

Beyond NP

Which of these puzzles are in NP? Which has a solution that is easy to check?

the Original I.Q. TESTER ©
Made in U.S.A.
© 1976 VENTURE MFG. CO., INC. DALTON, GA
Deluxe Edition
RULES OF THE GAME
START WITH ANY ONE HOLE EMPTY.
AS YOU JUMP THE PEGS REMOVE
THEM FROM THE BOARD.
TRY TO LEAVE ONLY ONE PEG.
SCORING FOR BEGINNERS
-LEAVE THREE - JUST SO-SO 10 POINTS
-LEAVE TWO-ABOVE AVERAGE-25 POINTS
-LEAVE ONE-VERY SMART-50 POINTS
SCORING FOR ADVANCED PLAYERS
LEAVE LAST PEG IN THE EXACT
HOLE YOU INITIALLY LEFT EMPTY
-BRILLIANT-100 POINTS.
LEAVE 8 PEGS ON THE BOARD WITH
NO REMAINING JUMPS-GENIUS
200 POINTS
olutions at www.pegame.com
Each game a Solitary Adventure

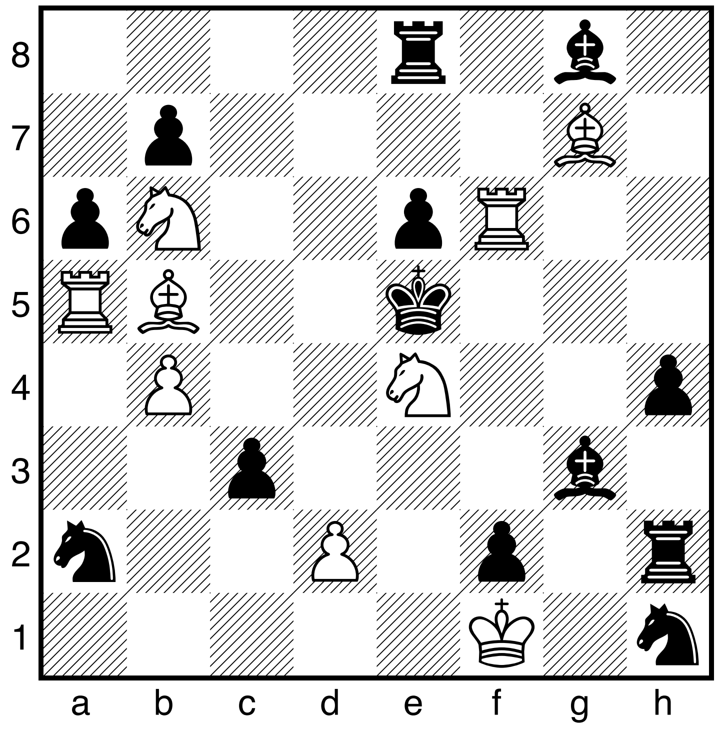
Deeper logical structures

I can win if there exists a move for me,

such that for all of your replies,

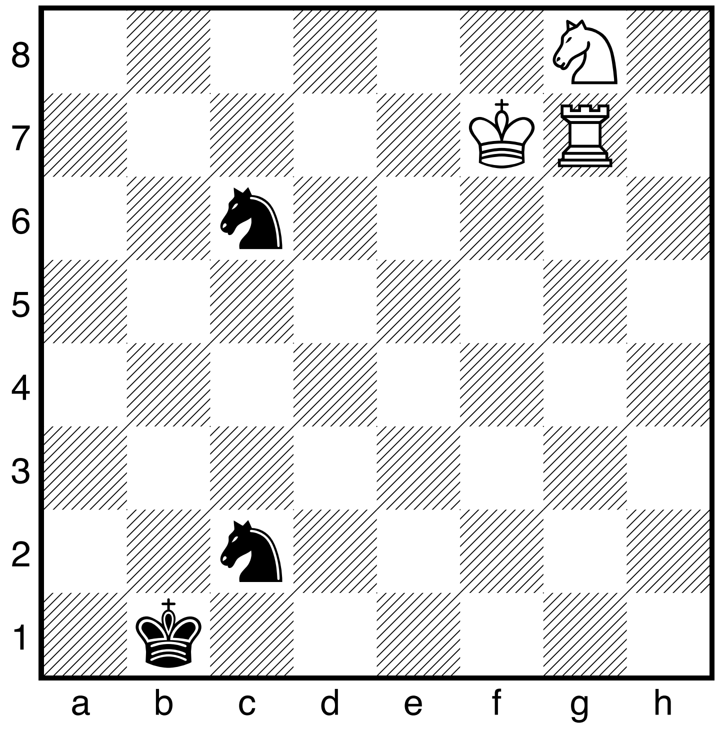
there exists a move for me...

Sam Loyd (1903)



Mate in 3

Lewis Stiller (1995)



Mate in 262

Undecidability

Suppose we could tell whether a program *p* will ever halt. This would be really handy!

```
Fermat
begin
    $t := 3$;
    repeat
        for $n = 3$ to $t$ do
            for $x = 1$ to $t$ do
                for $y = 1$ to $t$ do
                    for $z = 1$ to $t$ do
                        if $x^n + y^n = z^n$ then return $(x, y, z, n)$;
                    end
                end
            end
        end
        $t := t + 1$;
    until forever;
end
```

I have discovered a marvelous proof that this program will run forever, but it is too small to fit on this slide...

Undecidability

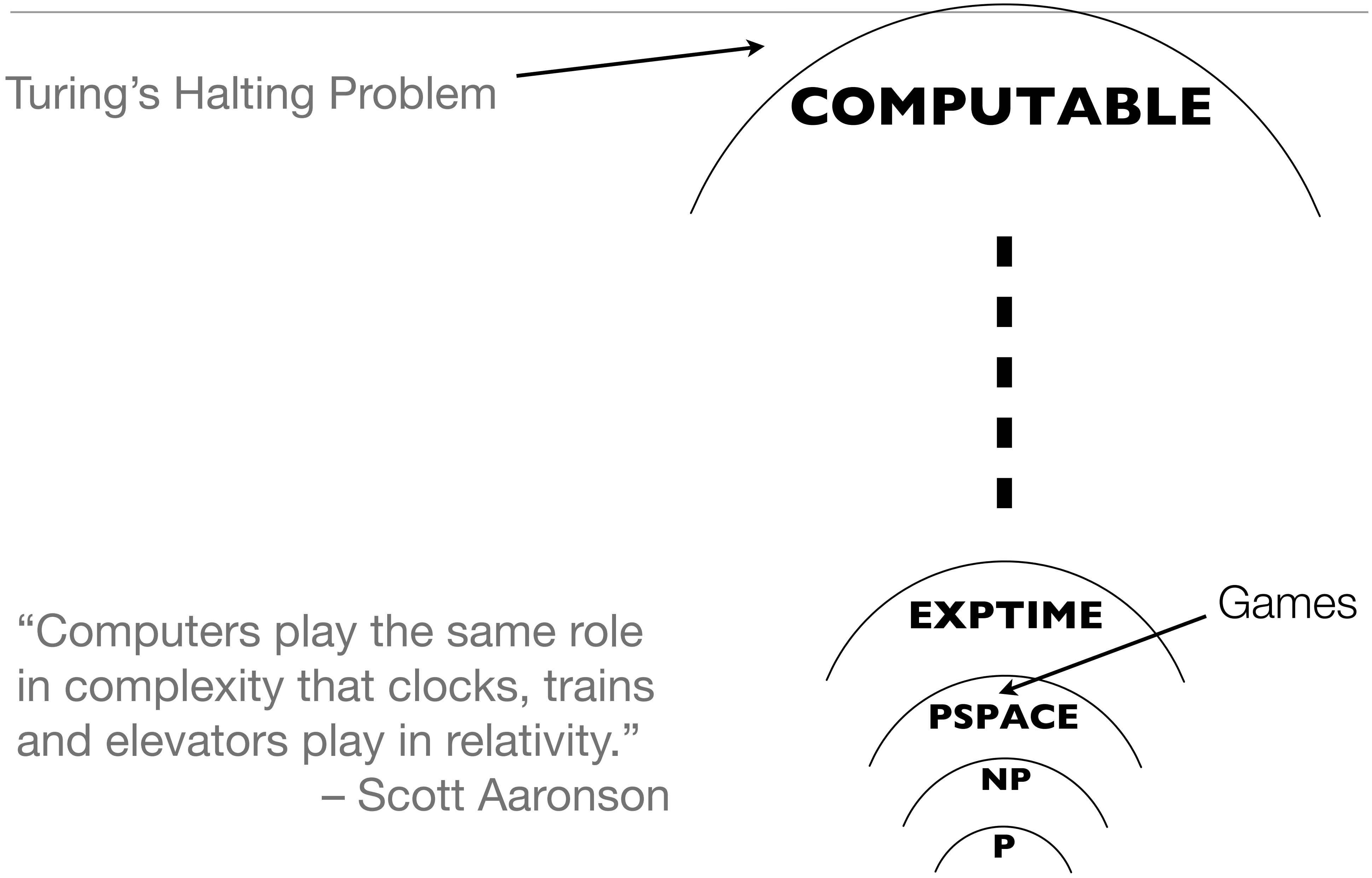
Suppose `halt(p,x)` can tell whether *p*, given input *x*, will halt.
Then we could feed it to itself, and run this program instead:

```
trouble(p):
  if halt(p,p) loop forever
  else halt
```

Will `trouble(trouble)` halt or not?

Undecidable problems ⇒ unprovable truths!

An infinite hierarchy



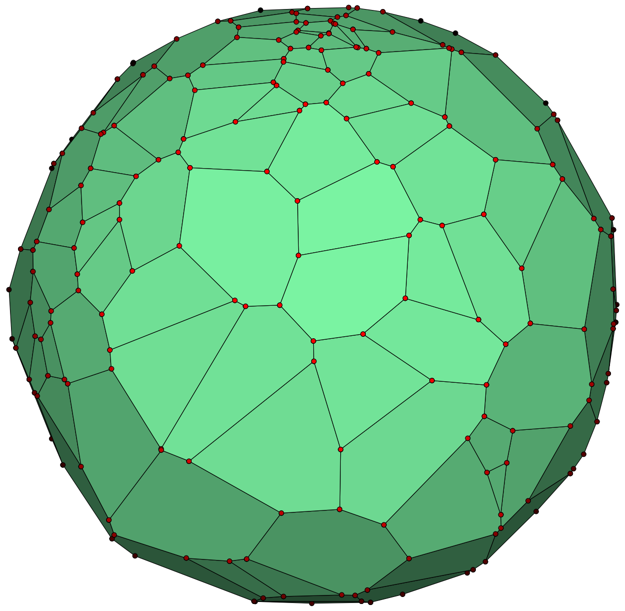
"Computers play the same role in complexity that clocks, trains and elevators play in relativity."
– Scott Aaronson

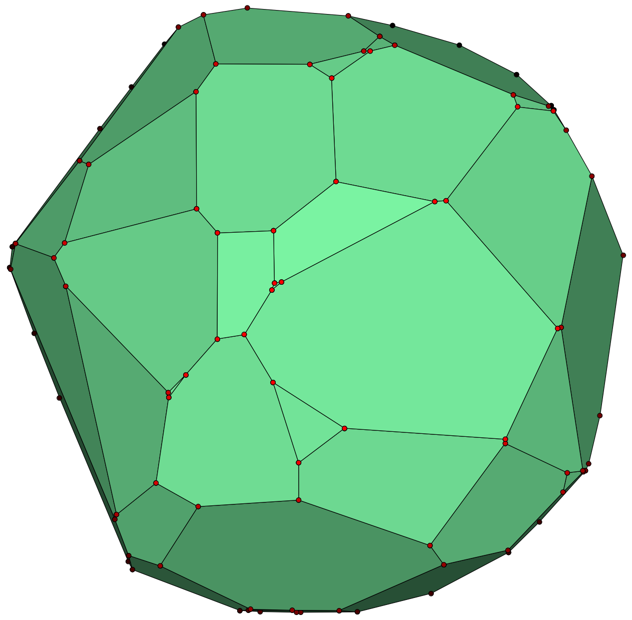
From theory to the real world

Many algorithms that take exponential time in the worst case are efficient in practice

Optimization problems are like exploring a high-dimensional jewel

If we add noise to the problem, the number of facets goes down, and the path to the top gets shorter





Deep questions

Is finding solutions harder than checking them? When can we avoid exhaustive search?

How much memory do we need?

What if we only need good answers, instead of the best ones? Are there problems where even finding good answers is hard?

How much does it help if we can do many things at once?

If we are working together to solve a problem, how much do we need to communicate?

How much does randomness help? Can we foil the adversary by being unpredictable?

How much does quantum physics help?

Shameless Plug

This book rocks! You somehow manage to combine the fun of a popular book with the intellectual heft of a textbook.

— Scott Aaronson

A treasure trove of information on algorithms and complexity, presented in the most delightful way.

— Vijay Vazirani

A creative, insightful, and accessible introduction to the theory of computing, written with a keen eye toward the frontiers of the field and a vivid enthusiasm for the subject matter.

— Jon Kleinberg

Oxford University Press, 2011