Introduction to the Analysis of Complexity

Chapter 27

Complexity Theory

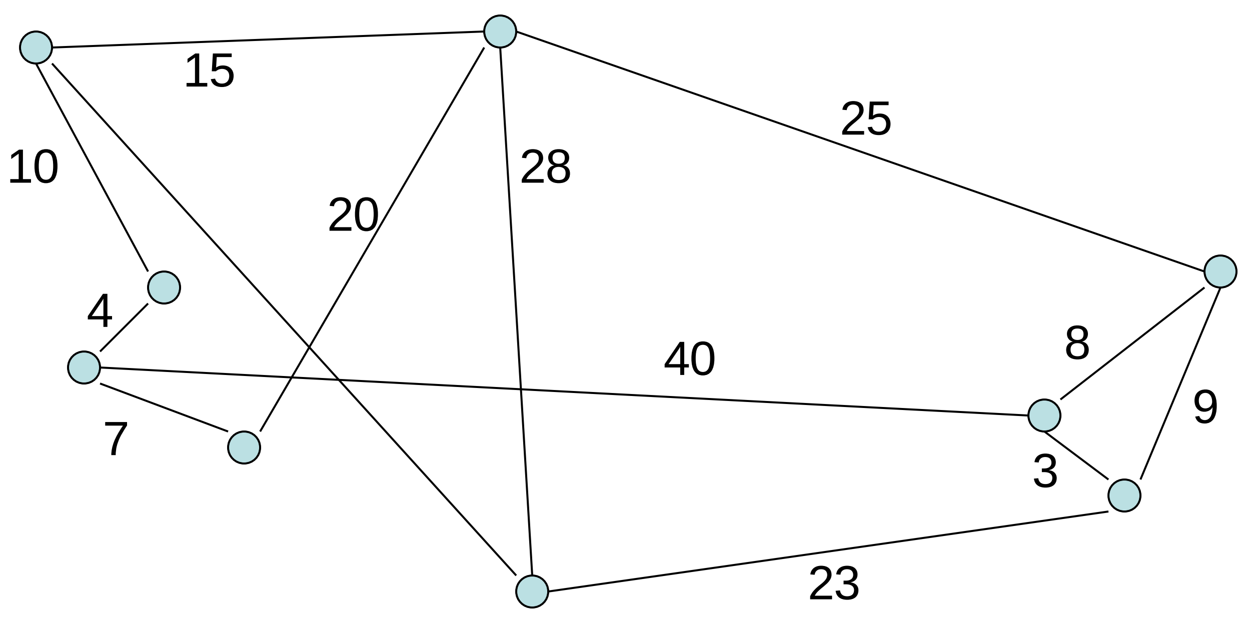
Are all decidable languages equal?

- (ab)*
- $WW^R = \{ww^R : w \in \{a, b\}^*\}$
- $WW = \{ww : w \in \{a, b\}^*\}$
- SAT = {*w* : *w* is a wff in Boolean logic and *w* is satisfiable}

Computability: classifies problems into: solvable and unsolvable
Complexity: classifies solvable problems into: easy ones and hard ones

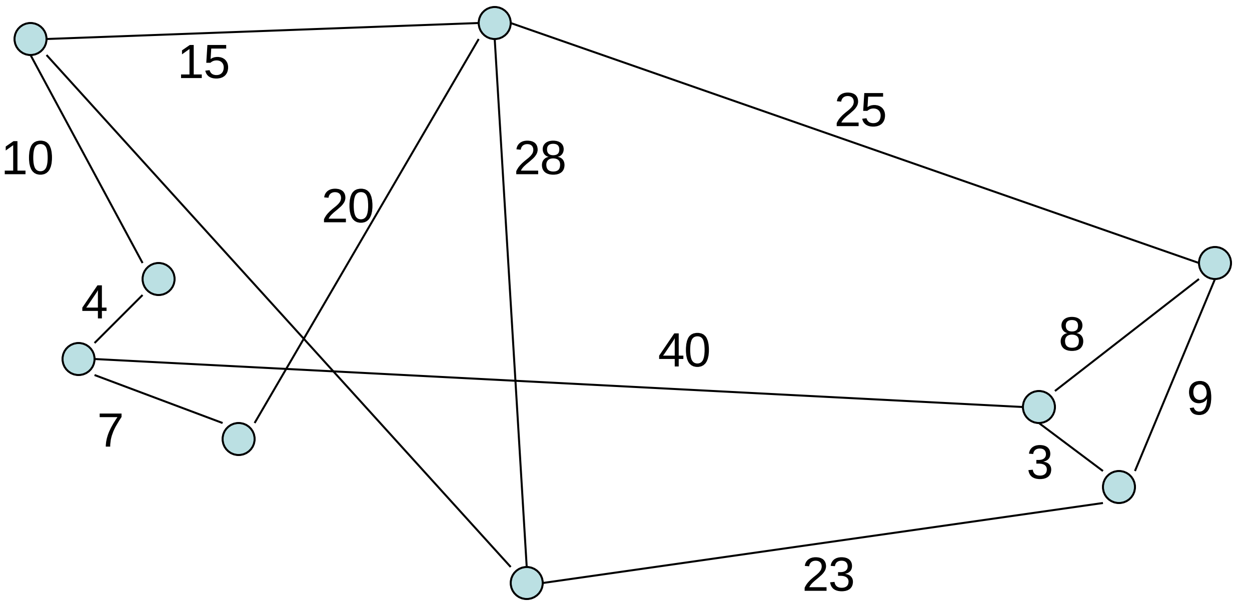
- Easy: tractable
- Hard: intractable
- Complexity zoo: hierarchy of classes

The Traveling Salesman Problem



Given *n* cities and the distances between each pair of them, find the shortest tour that returns to its starting point and visits each other city exactly once along the way.

The Traveling Salesman Problem



Given *n* cities:

Choose a first city	*n*
Choose a second	*n*-1
Choose a third	<u>*n*-2</u>
…	***n*!**

The Growth Rate of *n*!

2	2	11	479001600
3	6	12	6227020800
4	24	13	87178291200
5	120	14	1307674368000
6	720	15	20922789888000
7	5040	16	355687428096000
8	40320	17	6402373705728000
9	362880	18	121645100408832000
10	3628800	19	2432902008176640000
11	39916800	36	$3.6 \cdot 10^{41}$

Tackling Hard Problems

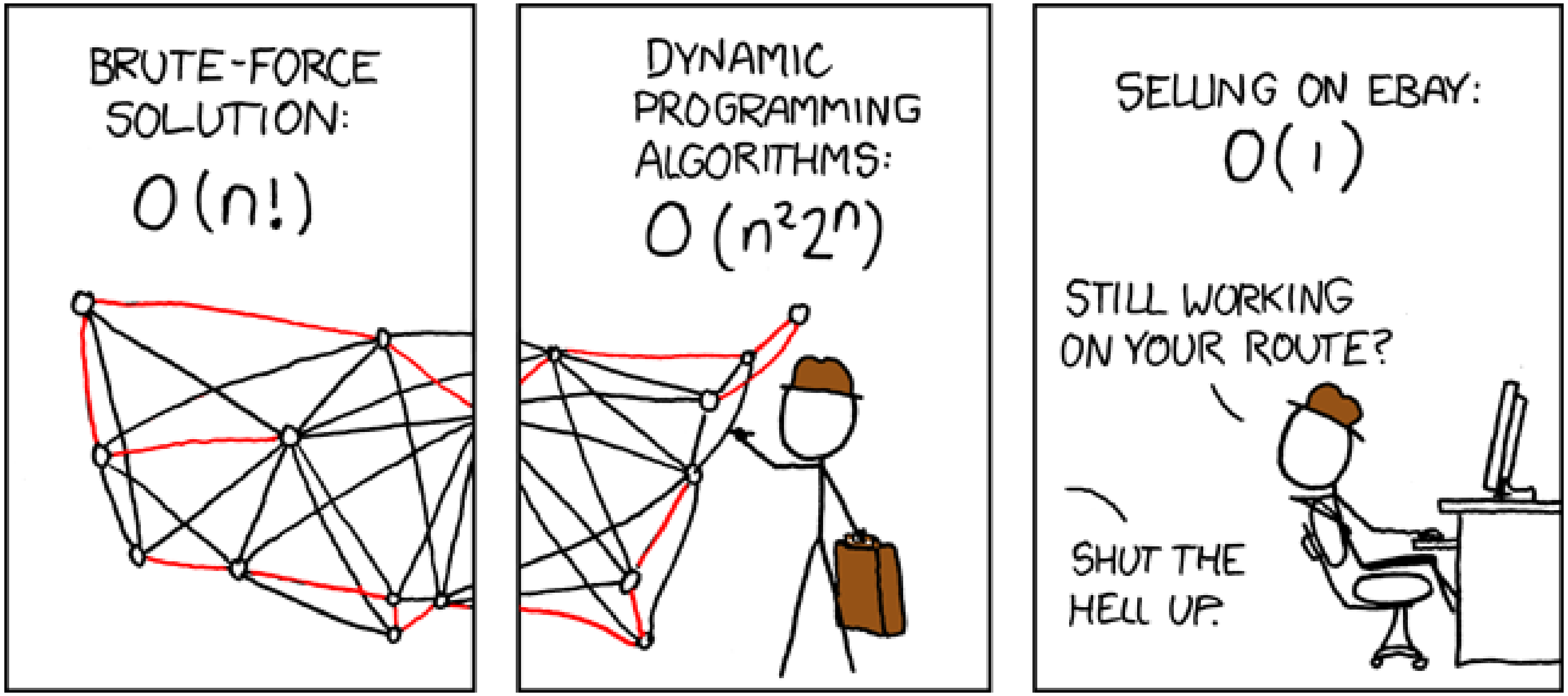
1. Use a technique that is guaranteed to find an optimal solution.

2. Use a technique that is guaranteed to run quickly and find a “good” solution.

   – The World Tour Problem

   Does it make sense to insist on true optimality if the description of the original problem was approximate?

Modern TSP



From: http://xkcd.com/399/

The Complexity Zoo

The attempt to characterize the decidable languages by their complexity:

https://complexityzoo.net/Complexity_Zoo

Notable ones:

P: solvable (decidable) by deterministic TM in polynomial time

- tractable
- context-free (including regular) languages are in P

NP: solvable (decidable) by nondeterministic TM in polynomial time

- given solution can be verified by DTM in polynomial time

NP-complete: as hard as any one in NP & in NP (hardest ones in NP)

- no efficient algorithm is known
- require non-trivial search, as in TSP

NP-hard: as hard as any one in NP (not necessarily in NP)

- Every problem in NP is reducible to it in polynomial time
- L is NP-complete if it is in NP + it is NP-hard
- Intractable = not in P. But since it's believed $P \neq NP$, loosely intractable = NP hard

Note

- NP is a set of decision problems. Computational complexity primarily deals with decision problems.
- There is a link between the "decision" and "optimization" problems in that if there exists a polynomial algorithm that solves the "decision" problem, then one can find the maximum value for the optimization problem in polynomial time by applying this algorithm iteratively while increasing the value of k .
- On the other hand, if an algorithm finds the optimal value of the optimization problem in polynomial time, then the decision problem can be solved in polynomial time by comparing the value of the solution output by this algorithm with the value of k .
- Thus, **both versions of the problem are of similar difficulty**.
- Note that, NP-hard is not restricted to decision problems, it also includes optimization problems.
- When the decision problem is NP-complete, the optimization problem is NP-hard

Methodology

- Characterizing problems as languages to be comparable
  - SAT = {*w* : *w* is a wff in Boolean logic and *w* is satisfiable}
  - only applies to decidable languages. nothing to say about H
  - Including optimization problems (optimization -> verification)
    - TSP-DECIDE = {<*G*, *cost*> : <*G*> encodes an undirected graph with a positive distance attached to each of its edges and *G* contains a Hamiltonian circuit whose total cost is less than *cost*}.
    - minimizing -> at most k
    - maximizing -> at least k

- All problems are decision problems
  - requires at least enough time to write the solution
  - By restricting our attention to decision problems, the length of the answer is not a factor

- Encoding matters as complexity is w.r.t. problem size
  - *L* is regular for one encoding, nonregular for another
  - E.g., for integers, we should use any base other than 1

111111111111 vs 1100
111111111111111111111111111111 vs 11110

Measuring Time and Space Complexity

- Model of computation: TM
- *timereq*(*M*) is a function of *n*:
  - If *M* is ***deterministic***

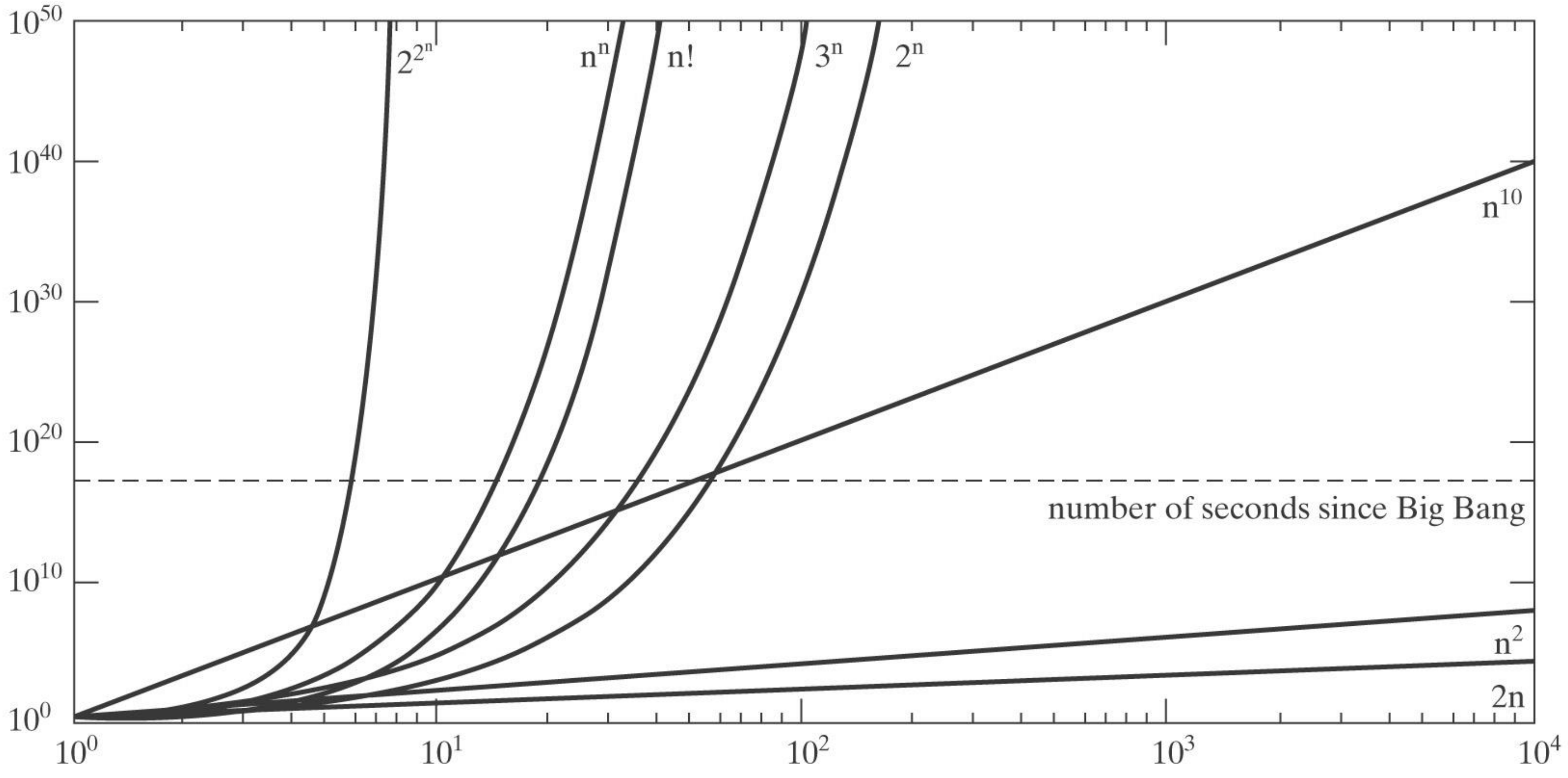
    timereq(*M*) = *f*(*n*) = the maximum number of steps that *M* executes on any input of length *n*.
  - If *M* is ***nondeterministic***

    timereq(*M*) = *f*(*n*) = the number of steps on the longest path that *M* executes on any input of length *n*.
- *spacereq*(*M*) is a function of *n*:
  - If *M* is ***deterministic***

    spacereq(*M*) = *f*(*n*) = the maximum number of tape squares that *M* reads on any input of length *n*.
  - If *M* is ***nondeterministic***

    spacereq(*M*) = *f*(*n*) = the maximum number of tape squares that *M* reads on any path that it executes on any input of length *n*.
- Focus on worst-case performance
  - Interested in upper bound
  - Easy to determine
  - Beware: could be quite different from average-case
  - for most real problems, worst case is rare

Growth Rates of Functions



Asymptotic upper bound - $\mathcal{O}$

$f(n) \in \mathcal{O}(g(n))$ iff there exists a positive integer k and a positive constant c such that:

$$\forall n \geq k\ (f(n) \leq c\, g(n)).$$

In other words, ignoring some number of small cases (all those of size less than k), and ignoring some constant factor c, $f(n)$ is bounded from above by $g(n)$.

Alternatively, if the limit exists: $$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

In this case, we'll say that f is "big-oh" of g
 or g asymptotically dominates f
 or g grows at least as fast as f does

Asymptotic upper bound - $\mathcal{O}$

- $n^3 \in \mathcal{O}(n^3)$
- $n^3 \in \mathcal{O}(n^4)$
- $3n^3 \in \mathcal{O}(n^3)$
- $n^3 \in \mathcal{O}(3^n)$
- $n^3 \in \mathcal{O}(n!)$
- $\log n \in \mathcal{O}(n)$

Summarizing $\mathcal{O}$

$$\mathcal{O}(c) \subseteq \mathcal{O}(\log_a n) \subseteq \mathcal{O}(n^b) \subseteq \mathcal{O}(d^n) \subseteq \mathcal{O}(n!)$$

Asymptotic strong upper bound o

$f(n) \in o(g(n))$ iff, for every positive c, there exists a positive integer k such that:

$$\forall n \geq k\ (f(n) < c\, g(n))$$

Alternatively, if the limit exists:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

In this case, we'll say that *f* is “little-oh” of *g* or that *g* grows strictly faster than *f* does.

Asymptotic lower bound - Ω

$f(n) \in \Omega(g(n))$ iff there exists a positive integer k and a positive constant c such that:

$\forall n \geq k\ (f(n) \geq c\, g(n))$

In other words, ignoring some number of small cases (all those of size less than k), and ignoring some constant factor c, $f(n)$ is bounded from below by $g(n)$.

Alternatively, if the limit exists:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

In this case, we'll say that f is "big-Omega" of g or that g grows no faster than f.

Asymptotic strong lower bound ω

$f(n) \in \omega(g(n))$ iff, for every positive c, there exists a positive integer k such that:

$\forall n \geq k \; (f(n) > c\, g(n))$

Alternatively, if the required limit exists:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

In this case, we'll say that *f* is "little-omega" of *g* or that *g* grows strictly slower than *f* does.

Asymptotic tight bound Θ

$f(n) \in \Theta(g(n))$ iff there exists a positive integer k and positive constants c_1, and c_2 such that:

$$\forall n \geq k \; (c_1\, g(n) \leq f(n) \leq c_2\, g(n))$$

Or:

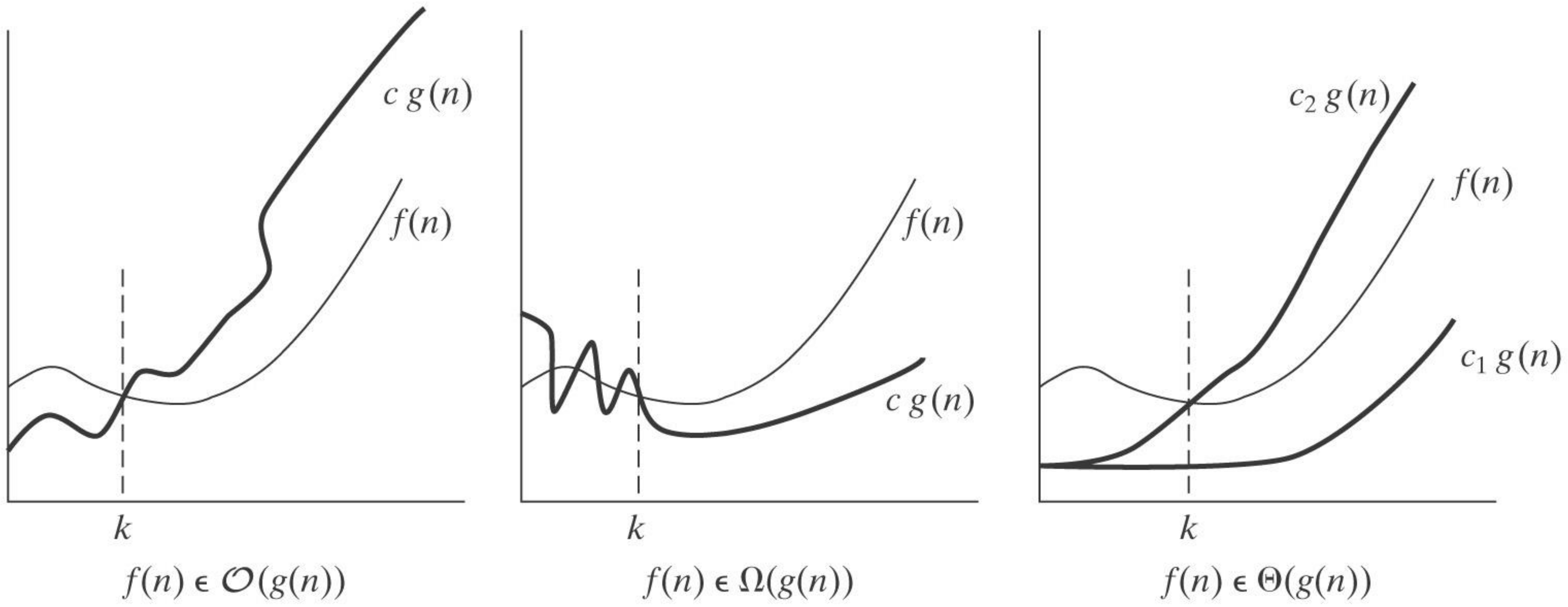
$f(n) \in \Theta(g(n))$ iff:
- $f(n) \in \mathcal{O}(g(n))$, and
- $g(n) \in \mathcal{O}(f(n))$.

Or:

$f(n) \in \Theta(g(n))$ iff:
- $f(n) \in \mathcal{O}(g(n))$, and
- $f(n) \in \Omega(g(n))$.

Is $n^3 \in \Theta(n^3)$?
Is $n^3 \in \Theta(n^4)$?
Is $n^3 \in \Theta(n^5)$?

Asymptotic Dominance



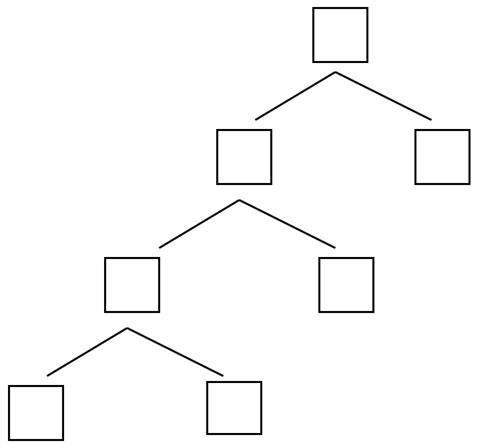
Algorithmic Gaps

We'd like to show:

1. Upper bound: There exists an algorithm that decides L and that has complexity C_1.
2. Lower bound: Any algorithm that decides L must have complexity at least C_2.
3. $C_1 = C_2$

If $C_1 = C_2$, we are done. Often, we're not done.

Time-Space Tradeoffs - Search



Depth-first search: small space, big time

- potential problem: get stuck in one path

Breadth-first search: big space, small time

Iterative-deepening search: compromise

- depth-first on length 1 paths, length 2 paths and so on
- Space same as depth-first, time slightly worse than breath-first

Time Complexity Classes

Chapter 28

The Language Class P

$L \in$ P iff

- there exists some deterministic Turing machine M that decides L, and
- $timereq(M) \in \mathcal{O}(n^k)$ for some k.

We' ll say that L is ***tractable*** iff it is in P.

To show a language is in P:

- describe a polynomial time one-tape, deterministic Turing machine
- state a in polynomial time algorithm that runs on a regular random-access computer
  - easier

Languages That Are in P

- Every regular language
- Every context-free language since there exist context-free parsing algorithms that run in $\mathcal{O}(n^3)$ time.
- Others:
  - $A^nB^nC^n$

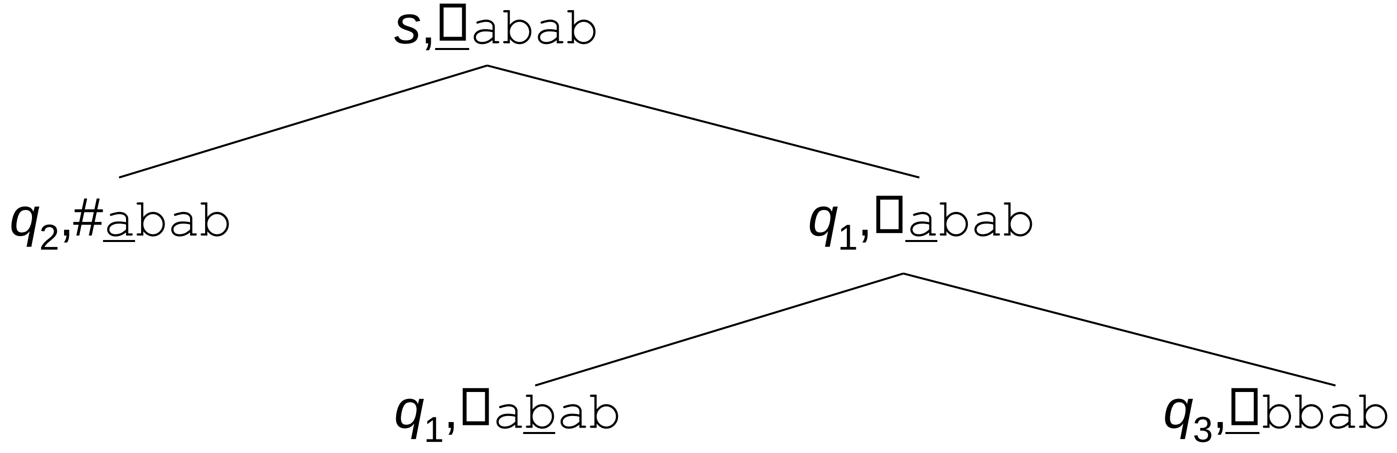
The Language Class NP

Nondeterministic deciding:

$L \in$ NP iff:

- there is some NDTM M that decides L, and
- $timereq(M) \in \mathcal{O}(n^k)$ for some k.

NDTM deciders: longest path is polynomial


s,$\underline{□}$abab
q_2,#$\underline{a}$bab
q_1,□$\underline{a}$bab
q_1,□a$\underline{b}$ab
q_3,$\underline{□}$bbab

Deterministic Verifying

A Turing machine V is a ***verifier*** for a language L iff:
$w \in L$ iff $\exists c$ ($<w, c> \in L(V)$).
We'll call c a ***certificate***.

An alternative definition for the class NP:
$L \in$ NP iff there exists a deterministic TM V such that:

- V is a verifier for L, and
- $timereq(V) \in \mathcal{O}(n^k)$ for some k.

- L is in NP iff it has a polynomial verifier.

- Can think of a nondeterministic algorithm as acting in two phases:
  – guess a solution (**certificate**) from a finite number of possibilities
  – verify whether it indeed solves the problem
- The verification phase takes polynomial time $\Leftrightarrow$ problem is in NP

To Show a Language is in NP

- Exhibit an NDTM to decide it in polynomial time.
  - Only count the time for the longest path

- Exhibit a DTM to verify a certificate in polynomial time
- Practically, state an algorithm that runs on a regular random-access computer that verify a certificate in polynomial time

Example

SAT = {*w* : *w* is a Boolean wff and *w* is satisfiable} is in NP

$F_1 = P \wedge Q \wedge \neg R$?
$F_2 = P \wedge Q \wedge R$?
$F_3 = P \wedge \neg P$?
$F_4 = P \wedge (Q \vee \neg R) \wedge \neg Q$?

SAT-decide(F_4) =

SAT-verify (<F_4, (P = *True*, Q = *False*, R = *False*)>) =

3-SAT

- A ***literal*** is either a variable or a variable preceded by a single negation symbol.
- A ***clause*** is either a single literal or the disjunction of two or more literals.
- A wff is in ***conjunctive normal form*** (or CNF) iff it is either a single clause or the conjunction of two or more clauses.
- A wff is in ***3-conjunctive normal form*** (or 3-CNF) iff it is in conjunctive normal form and each clause contains exactly three literals.

3-SAT

	3-CNF	**CNF**
$(P \vee \neg Q \vee R)$	●	●
$(P \vee \neg Q \vee R) \wedge (\neg P \vee Q \vee \neg R)$	●	●
P		●
$(P \vee \neg Q \vee R \vee S) \wedge (\neg P \vee \neg R)$		●
$P \rightarrow Q$		
$(P \wedge \neg Q \wedge R \wedge S) \vee (\neg P \wedge \neg R)$		
$\neg(P \vee Q \vee R)$		

Every wff can be converted to an equivalent wff in CNF.

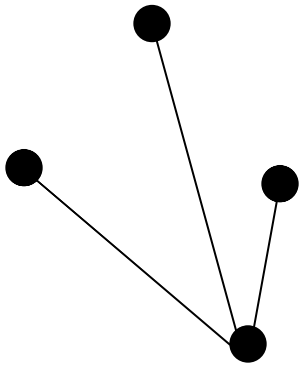
3-SAT = { *w* : *w* is a wff in Boolean logic,
w is in 3-conjunctive normal form, and
w is satisfiable}.

Is 3-SAT in NP?

INDEPENDENT-SET

INDEPENDENT-SET = {<*G*, *k*> : *G* is an undirected graph and *G* contains an independent set of at least *k* vertices}.

An ***independent set*** is a set of vertices no two of which are adjacent (i.e., connected by a single edge).



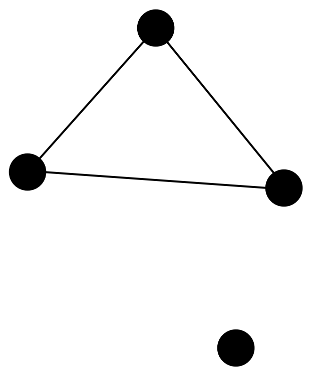
In a scheduling program the vertices represent tasks and are connected by an edge if their corresponding tasks conflict. We can find the largest number of tasks that can be scheduled at the same time by finding the largest independent set in the task graph.

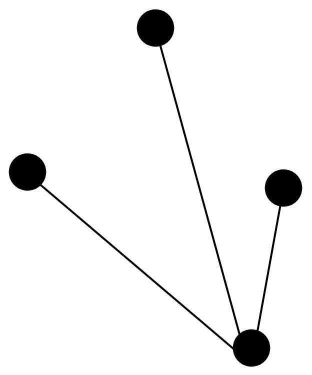
CLIQUE

CLIQUE = {<*G*, *k*> : *G* is an undirected graph with vertices *V* and edges *E*, *k* is an integer, $1 \le k \le |V|$, and *G* contains a *k*-clique}.

A ***clique*** in *G* is a subset of *V* where every pair of vertices in the clique is connected by some edge in *E*.

A ***k*-clique** is a clique that contains exactly *k* vertices.





Other Languages That Are in NP

Graph-based languages:

- TSP-DECIDE
- HAMILTONIAN-PATH = {<*G*> : *G* is an undirected graph and *G* contains a Hamiltonian path}
  - a path that visits each vertex exactly once
- HAMILTONIAN-CIRCUIT = {<*G*> : *G* is an undirected graph and *G* contains a Hamiltonian circuit}
  - Or Hamiltonian cycle. a cycle that visits each vertex exactly once, ending at the starting vertex

BIN-PACKING

- BIN-PACKING = {<*S*, *c*, *k*> : *S* is a set of objects each of which has an associated size and it is possible to divide the objects so that they fit into *k* bins, each of which has size *c*}.

In three dimensions:

In two dimensions:

Brand New! Bigger, Better Mortgage Leads
mont blanc
3 million dollar investment for 2006-07 season !
10 Weeks
3 ways to Save!
Kids stay, ski, eat FREE
Get more money! Buy Sell Home
U.S.Housing & Mortgage Newsletter
eNet advertising
The Largest Online Ad Network

KNAPSACK

KNAPSACK = {<*S*, *v*, *c*> : *S* is a set of objects each of which has an associated cost and an associated value, *v* and *c* are integers, and there exists some way of choosing elements of *S* (duplicates allowed) such that the total cost of the chosen objects is at most *c* and their total value is at least *v*}.

Notice that, if the cost of each item equals its value, then the KNAPSACK problem becomes the SUBSET-SUM problem.

How to pack a knapsack with limited capacity in such as way as to maximize the utility of the contents:

- A thief
- A backpacker
- Choosing ads for a campaign
- What products should a company make?

P and NP

P: languages for which membership can be decided quickly

- Solvable by a DTM in poly-time

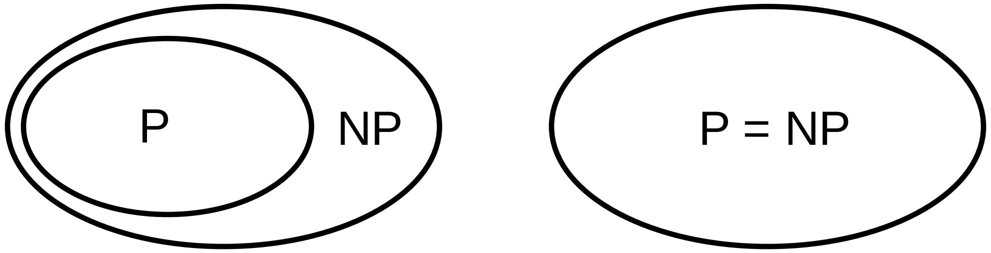
NP: languages for which membership can be verified quickly

- Solvable by a NDTM in poly-time

Greatest unsolved problem in theoretical computer science:

Is P = NP? ***The Millenium Prize***

Two possibilities:



If P = NP, any polynomially verifiable problems would be polynomially decidable

P and NP: What We Know

PSPACE: $L \in$ PSPACE iff there is some deterministic TM M that decides L, and $spacereq(M) \in \mathcal{O}(n^k)$ for some k.

- Solvable by a DTM in poly-space

NPSPACE: $L \in$ NPSPACE iff there is some nondeterministic TM M that decides L, and $spacereq(M) \in \mathcal{O}(n^k)$ for some k.

- Solvable by a NDTM in poly-space

EXPTIME: $L \in$ EXPTIME iff there is some deterministic TM M that decides L, and $timereq(M) \in \mathcal{O}(2^{(n^k)})$ for some k.

- Solvable by a DTM in exponential-time

Here are some things we know:

P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ NPSPACE $\subseteq$ EXPTIME

P $\neq$ EXPTIME

- So there exist decidable but intractable problems
- So at least one of the inclusions shown above must be proper
- Generally assumed all of them are, but no proofs

Some problems are even harder than EXPTIME-complete problems.

P and *NP*

If a Turing Machine were to guess,
With less computational stress,
Polynomially check
That its guess was not dreck,
Thus avoiding enumerative mess.

The machine would then certainly be
Witness to a language in *NP*.
By guessing, we shirk
Exponential-time work.
Nondeterminism can do search for free.

Cook and Levin, I am now recalling
Found a language *L* that's really galling:
If there is time compaction
To prove satisfaction
Then *P*=*NP*--appalling!

It is computational fate
To encounter such barriers we hate.
Yet there are computations
To find estimations
Within factors approximate.

Quantum Computing

- **Quantum computing** is a non-classical model of computation. Whereas traditional models such as Turing machine rely on classical representations of computational memory, quantum computation could transform the memory into a quantum superposition of possible classical states. A quantum computer is a device that could perform such computation.
- **Quantum superposition** is a fundamental principle of quantum mechanics. Much like waves in classical physics, any two (or more) quantum states can be added together ("superposed") and the result will be another valid quantum state.
- Usually, computation on a quantum computer ends with a measurement. This leads to a collapse of quantum state to one of the basis states. It can be said that the quantum state is measured to be in the correct state with high probability.

Quantum Computing

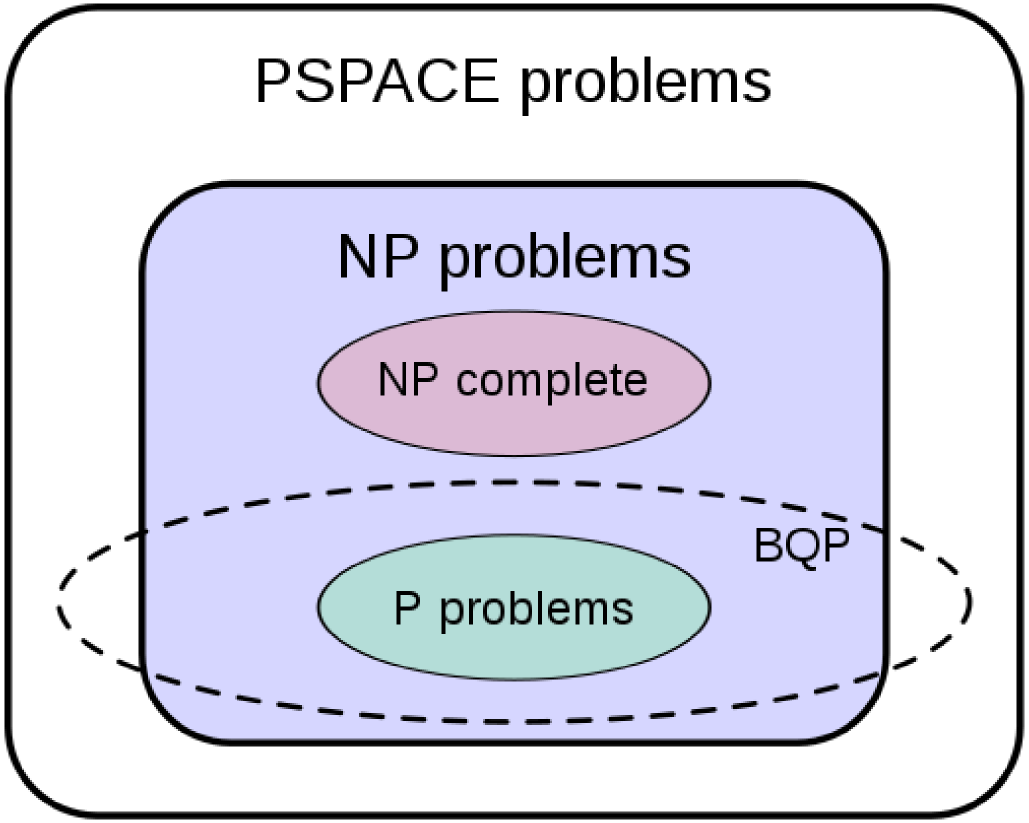
- A **quantum Turing machine** (QTM), is an abstract machine used to model the effect of a quantum computer. It provides a very simple model which captures all of the power of quantum computation. Any quantum algorithm can be expressed formally as a particular quantum Turing machine.

- A language *L* is in **BQP** (bounded-error quantum polynomial time) iff there exists a polynomial quantum Turing machine that accepts *L* with an error probability of at most 1/3 for all instances.

- Quantum computers have gained widespread interest because some problems of practical interest are known to be in BQP, but suspected to be outside P.
  - Integer factorization (see Shor's algorithm)
  - Discrete logarithm
  - Simulation of quantum systems
  - Approximating the Jones polynomial at certain roots of unity

BQP and NP

- The relation between BQP and NP is not known.
  - QTM vs NDTM
  - quantum Turing machines are not deterministic, but different from nondeterministic Turing machines



The suspected relationship of **BQP** to other problem spaces

Mapping Reduction

A ***mapping reduction*** R from L_{old} to L_{new} is a

- Turing machine that implements some computable function f with the property that:

$$\forall x\ (x \in L_{old} \text{ iff } f(x) \in L_{new})$$

  - if $x \in L_{old}$ then $f(x) \in L_{new}$
  - if $x \notin L_{old}$ then $f(x) \notin L_{new}$

If $L_{old} \leq L_{new}$ and M decides L_{new}, then: $C(x) = M(R(x))$ will decide L_{old}

- A mapping reduction is an algorithm that can transform any instance of decision problem L_{old} into an instance of decision problem L_{new}, in such a way that the answer (yes/no) to any L_{old} instance must be the same as the answer to the corresponding L_{new} instance.

Polynomial Reducibility

If *R* is deterministic polynomial then:

$$L_{old} \leq_{\mathrm{P}} L_{new}$$

And, whenever such an *R* exists:

- L_{old} must be in P if L_{new} is: if L_{new} is in P then there exists some deterministic, polynomial-time Turing machine *M* that decides it. So *M*(*R*(*x*)) is also a deterministic, polynomial-time Turing machine and it decides L_{old}

- L_{old} must be in NP if L_{new} is: if L_{new} is in NP then there exists some nondeterministic, polynomial-time Turing machine *M* that decides it. So *M*(*R*(*x*)) is also a nondeterministic, polynomial-time Turing machine and it decides L_{old}

3-SAT and INDEPENDENT-SET

3-SAT $\leq_P$ INDEPENDENT-SET.

Strings in 3-SAT describe formulas that contain literals and clauses.

$$(P \vee Q \vee \neg R) \wedge (R \vee \neg S \vee Q)$$

Strings in INDEPENDENT-SET describe graphs that contain vertices and edges.

```
101/1/11/11/10/10/100/100/101/11/101
```

Gadgets

A ***gadget*** is a structure in the target language that mimics the role of a corresponding structure in the source language.

Example: 3-SAT $\leq_P$ INDEPENDENT-SET.

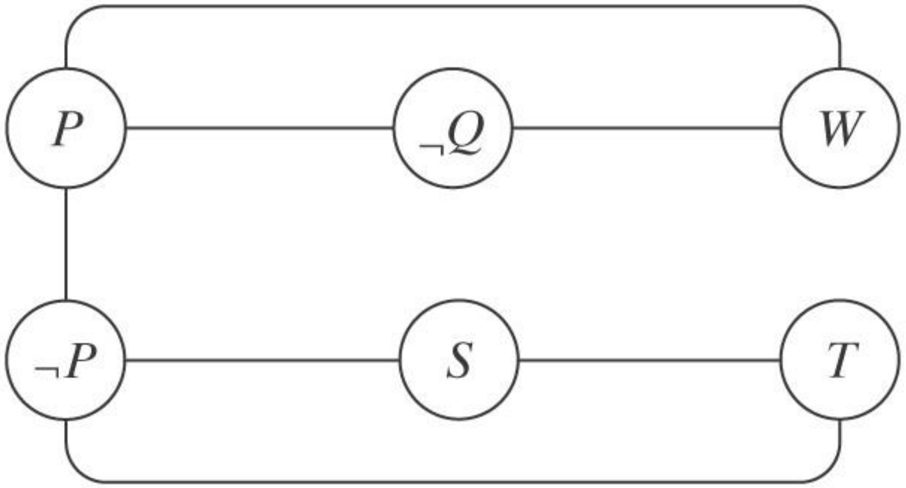
So we need:

- a gadget that looks like a graph but that mimics a literal, and
- a gadget that looks like a graph but that mimics a clause.

3-SAT $\leq_P$ INDEPENDENT-SET

```
R(<f: Boolean formula with k clauses>) =
    1. Build a graph G by doing the following:
        1.1. Create one vertex for each instance of each literal in f. (literal
             gadget)
        1.2. Create an edge between each pair of vertices for symbols in
             the same clause. (clause gadget)
        1.3. Create an edge between each pair of vertices for
                  complementary literals.
    2. Return <G, k>.
```

$(P \vee \neg Q \vee W) \wedge (\neg P \vee S \vee T)$:

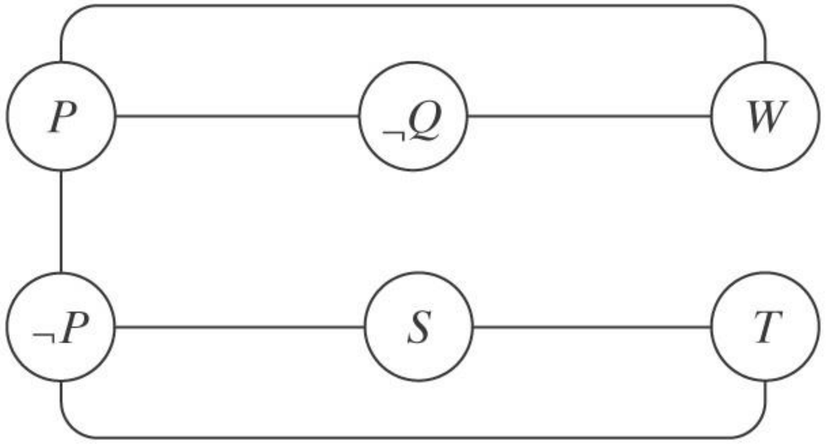


R is Correct

Show: $f \in$ 3-SAT iff $R(<f>) \in$ INDEPENDENT-SET by showing:

- $f \in$ 3-SAT $\rightarrow R(<f>) \in$ INDEPENDENT-SET
- $R(<f>) \in$ INDEPENDENT-SET $\rightarrow f \in$ 3-SAT

One Direction



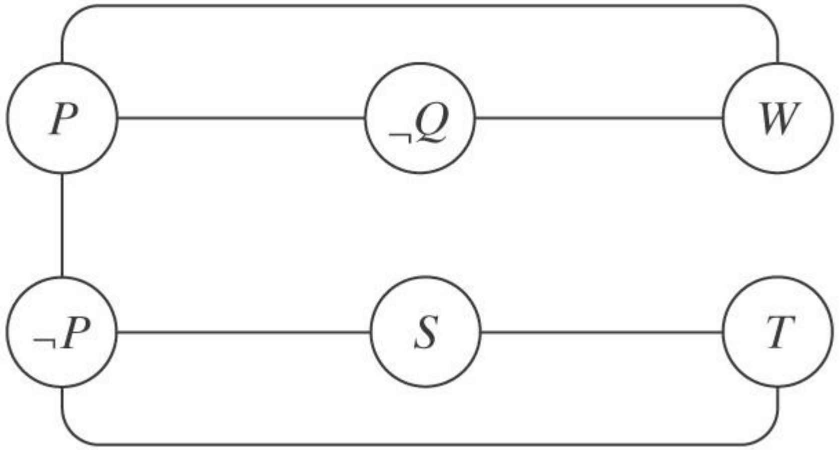
$f \in$ 3-SAT $\rightarrow$ *R*(<*f*>) $\in$ INDEPENDENT-SET:
There is a satisfying assignment *A* to the symbols in *f*.
So *G* contains an independent set *S* of size *k*, built by:

1. From each clause gadget choose one literal that is made positive by *A*.
2. Add the vertex corresponding to that literal to *S*.

S will contain exactly *k* vertices and is an independent set:

- No two vertices come from the same clause so step 1.2 could not have created an edge between them.
- No two vertices correspond to complimentary literals so step 1.3 could not have created an edge between them.

The Other Direction



- *R*(<*f*>) $\in$ INDEPENDENT-SET.
- So the graph *G* that *R* builds contains an independent set *S* of size *k*.
- We prove that there is some satisfying assignment *A* for *f*:

No two vertices in *S* come from the same clause gadget. Since *S* contains at least *k* vertices, no two are from the same clause, and *f* contains *k* clauses, *S* must contain one vertex from each clause.

Build *A* as follows:

1. Assign *True* to each literal that corresponds to a vertex in *S*.
2. Assign arbitrary values to all other literals.

Since each clause will contain at least one literal whose value is *True*, the value of *f* will be *True*.

Why Do Reduction?

Would we ever choose to solve 3-SAT by reducing it to INDEPENDENT-SET?

The way we use reduction is: known 3-SAT (L_{old}) is NP-complete, show INDEPENDENT-SET (L_{new}) is NP-complete.

NP-Completeness

Sections 28.5, 28.6

NP-Completeness

A language *L* might have these properties:

1. L is in NP.
2. Every language in NP is deterministic, polynomial-time reducible to *L*.

- *L* is ***NP-hard*** iff it possesses property 2.

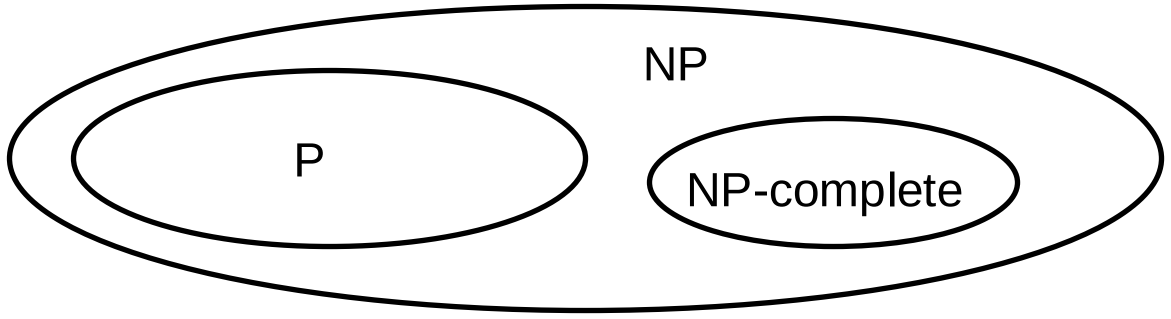
  An NP-hard language is at least as hard as any other language in NP. It may not be in NP.

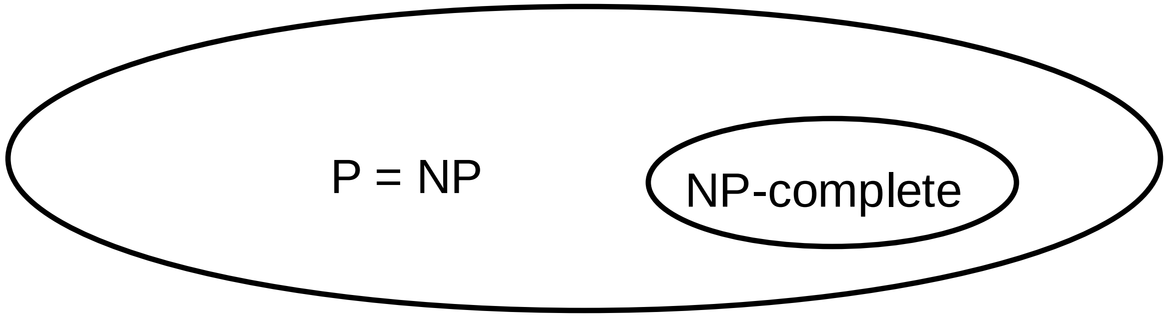
- *L* is NP***-complete*** iff it possesses *both* property 1 and property 2.

  All NP-complete languages can be viewed as being equivalently hard, the hardest ones in NP.

NP-Completeness

- The class of NP-complete is important, many of its members, like TSP-decide, have substantial practical significance.
- Two possibilities:





Showing that L_{new} is NP-Complete

How about: take a list of known NP languages and crank out the reductions?

$NPL_1 \leq_P L_{new}$

$NPL_2 \leq_P L_{new}$

$NPL_3 \leq_P L_{new}$

…

- infinite number of NP languages

Showing that L_{new} is NP-Complete

Suppose we had one NP-complete language L_{old} :

$$\begin{tikzcd} \underbrace{NPL_1 \qquad NPL_2 \qquad NPL_3 \qquad NPL_4 \qquad NPL...}_{L_{old}} \arrow[d] \\ L_{new} \end{tikzcd}$$

Theorem:

If: L_{old} is NP-complete

$L_{old} \leq_P L_{new}$

L_{new} is in NP

Then L_{new} is also NP-complete

So we need a first NP-complete language !

Proving that L_{new} is NP-Complete

Theorem: If L_{old} is NP-complete, $L_{old} \leq_P L_{new}$, and L_{new} is in NP, then L_{new} is also NP-complete.

Proof: If L_{old} is NP-complete then every other NP language is deterministic, polynomial-time reducible to it. So let L be any NP language and let R_L be the Turing machine that reduces L to L_{old}. If $L_{old} \leq_P L_{new}$, let R_2 be the Turing machine that implements that reduction. Then L can be deterministic, polynomial-time reduced to L_{new} by first applying R_L and then applying R_2. Since L_{new} is in NP and every other language in NP is deterministic, polynomial-time reducible to it, it is NP-complete.

The Cook-Levin Theorem

Define: SAT = {*w* : *w* is a wff in Boolean logic and *w* is satisfiable}

Theorem: SAT is NP-complete.

Proof:

- SAT is in NP.
- SAT is NP-hard.

SAT is NP-Hard

- Let *L* be any language in NP.
- Let *M* be one of the NDTMs that decides *L*.

Define an algorithm that, given *M*, constructs a reduction *R* with the property that:

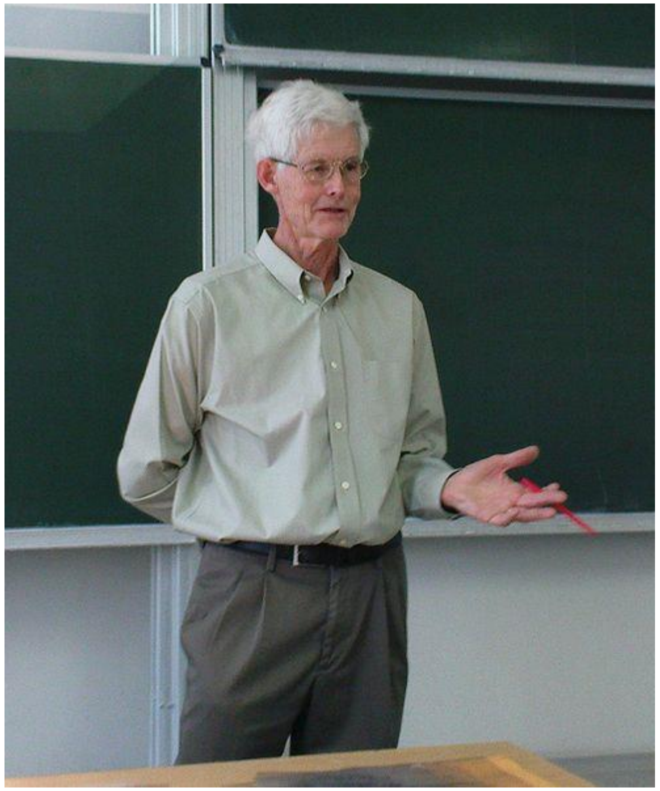
$$w \in L \text{ iff } R(w) \in \text{SAT}.$$

R takes a string *w* and returns a Boolean wff that is satisfiable iff $w \in L$.

Stephen Cook

- Bachelor, U of Michigan
- Ph.D. in math, Harvard, 1966
- Taught at Berkley, now at Toronto
- Formalized notion of NP-completeness
- Turing award in 1982, citation reads:

For his advancement of our understanding of the complexity of computation in a significant and profound way. His seminal paper, The Complexity of Theorem Proving Procedures, presented at the 1971 ACM SIGACT Symposium on the Theory of Computing, laid the foundations for the theory of NP-Completeness. The ensuring exploration of the boundaries and nature of NP-complete class of problems has been one of the most active and important research activities in computer science for the last decade.



1939 -

From 1966 to 1970, Assistant Professor at Berkley, math department, which infamously denied him tenure. In a speech celebrating the 30th anniversary of the Berkeley EECS department, fellow Turing Award winner and Berkeley professor Richard Karp said that, "It is to our everlasting shame that we were unable to persuade the math department to give him tenure. Perhaps they would have done so if he had published his proof of the NP-completeness of satisfiability a little earlier.”

Leonid Levin

- 1st Ph.D, Moscow University, 1972
    - with Andrey Kolmogorov
- 2nd Ph.D, MIT, 1979
- Now professor at BU
- Levin's journal article on the theorem was published in 1973; he had lectured on the ideas in it for some years before that time, though complete formal writing of the results took place after Cook's publication.

1948 -

Andrey Kolmogorov (1903 – 1987): Soviet Russian mathematician, preeminent in the 20th century, who advanced various scientific fields:

- probability theory
- topology
- intuitionistic logic
- turbulence
- classical mechanics
- computational complexity

NP-Complete Languages

- **INDEPENDENT-SET** = {<*G*, *k*> : *G* is an undirected graph and *G* contains an independent set of at least *k* vertices}.

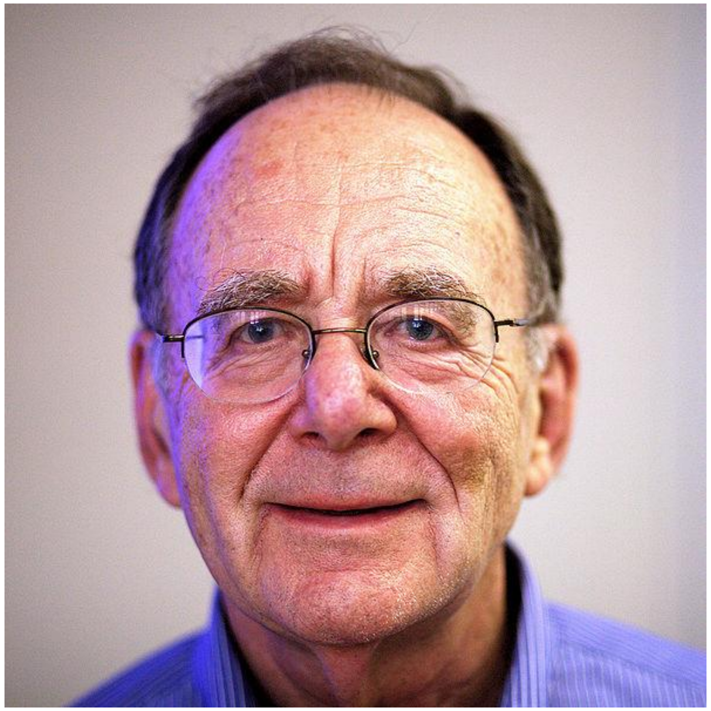
- **CLIQUE** = {<*G*, *k*> : *G* is an undirected graph with vertices *V* and edges *E*, *k* is an integer, $1 \le k \le |V|$, and *G* contains a *k*-clique}.

NP-Complete Languages

- SUBSET-SUM = {<*S*, *k*> : *S* is a set of integers, *k* is an integer, and there exists some subset of *S* whose elements sum to *k*}.
- SET-PARTITION = {<*S*> : *S* is a set of objects each of which has an associated cost and there exists a way to divide *S* into two subsets, *A* and *S* - *A*, such that the sum of the costs of the elements in *A* equals the sum of the costs of the elements in *S* - *A*}.
- TSP-DECIDE.
- HAMILTONIAN-PATH = {<*G*> : *G* is an undirected graph and *G* contains a Hamiltonian path}.
- HAMILTONIAN-CIRCUIT = {<*G*> : *G* is an undirected graph and *G* contains a Hamiltonian circuit}.
- KNAPSACK = {<*S*, *v*, *c*> : *S* is a set of objects each of which has an associated cost and an associated value, *v* and *c* are integers, and there exists some way of choosing elements of *S* (duplicates allowed) such that the total cost of the chosen objects is at most *c* and their total value is at least *v*}.
- BIN-PACKING = {<*S*, *c*, *k*> : *S* is a set of objects each of which has an associated size and it is possible to divide the objects so that they fit into *k* bins, each of which has size *c*}.

Richard Karp

- Bachelor, U of Michigan
- Ph.D. in applied math, Harvard, 1959
- Now at Berkley
- 1972, published a landmark paper, "Reducibility Among Combinatorial Problems", proved 21 problems to be NP-complete
- Standardized proving methodology
- National medal of science
- Turing award in 1985, citation reads:


1935 -

For his continuing contributions to the theory of algorithms including the development of efficient algorithms for network flow and other combinatorial optimization problems, the identification of polynomial-time computability with the intuitive notion of algorithmic efficiency, and, most notably, contributions to the theory of NP-completeness. Karp introduced the now standard methodology for proving problems to be NP-complete which has led to the identification of many theoretical and practical problems as being computationally difficult.

Strategy for Proving NP-completeness of L_{new}

- Show that L_{new} belongs to NP
  - Exhibit an NDTM to decide it in polynomial time.

    Or, equivalently,
  - Exhibit a DTM to verify it in polynomial time
  - This establishes an upper bound on the complexity of L_{new}
- Show that L_{new} is NP-hard by finding another NP-hard language L_{old} such that

$$L_{old} \leq_{\mathrm{P}} L_{new}$$

  - This establishes a lower bound on the complexity of L_{new}

Proving 3-SAT is NP-Complete

Define: 3-SAT = {<*w*> : *w* is a wff in Boolean logic, *w* is in 3-conjunctive normal form and *w* is satisfiable}.

$$(P \vee R \vee \neg T) \wedge (S \vee \neg R \vee W)$$

Theorem: 3-SAT is NP-complete.

Proof: We have shown that 3-SAT is in NP.

What about NP-hard?

3-SAT

First we try a reduction from SAT:

R(*w*: wff of Boolean logic) =

1. Use *conjunctiveBoolean* to construct w', where w' is in conjunctive normal form and w' is equivalent to w.
2. Use *3-conjunctiveBoolean* to construct w'', where w'' is in 3-conjunctive normal form and w'' is satisfiable iff w' is.
3. Return w''.

Does *R* run in polynomial time?

Converting to CNF

$$\begin{tikzcd}
((p \wedge q) \vee (r \wedge s)) \arrow[d] & \vee & (t \wedge v) \vee (w \wedge x)) \arrow[d] \\
((p \vee r) \wedge (q \vee r) \wedge (p \vee s) \wedge (q \vee s)) & \vee & {}
\end{tikzcd}$$

3-SAT is NP-Hard

Idea 1: Retain the idea of reducing SAT to 3-SAT.

For *R* to be a reduction from SAT to 3-SAT, it is sufficient to assure that *w′* is satisisfiable iff *w* is.

There exists a polynomial-time algorithm that constructs, from any wff *w*, a *w′* that meets that requirement.

If we replace step one of *R* with that algorithm, *R* is a polynomial-time reduction from SAT to 3-SAT.

So 3-SAT is NP-hard.

3-SAT is NP-Hard

Idea 2: Prove that 3-SAT is NP-hard directly.

It is possible to modify the reduction *R* that proves the Cook-Levin Theorem so that it constructs a formula in conjunctive normal form.

R will still run in polynomial time.

Once *R* has constructed a conjunctive normal form formula *w*, we can use *3-conjunctiveBoolean* to construct w', where w' is in 3-conjunctive normal form and w' is satisfiable iff *w* is.

This composition of *3-conjunctiveBoolean* with *R* shows that any NP language can be reduced to 3-SAT.

So 3-SAT is NP-hard.

Proving INDEPENDENT-SET is NP-Complete

Theorem: INDEPENDENT-SET is NP-complete.

- INDEPENDENT-SET is in NP:

```
Ver(<G, k, c>) =
1. Check that the number of vertices in c is at least k and no
   more than |V|.  If it is not, reject.
2. For each vertex v in c:
       For each edge e in E that has v as one endpoint:
           Check that the other endpoint of e is not in c.
```

Timereq(*Ver*) $\in \mathcal{O}(|c| \cdot |E| \cdot |c|)$.

$|c|$ and $|E|$ are polynomial in $|<G, k>|$.

So *Ver* runs in polynomial time.

- INDEPENDENT-SET is NP-hard:

  3-SAT $\leq_P$ INDEPENDENT-SET.

$$\begin{tikzcd}
\text{SAT} \arrow[d] \\
\text{3-SAT} \arrow[d] \\
\text{INDEPENDENT-SET}
\end{tikzcd}$$

Example Reductions

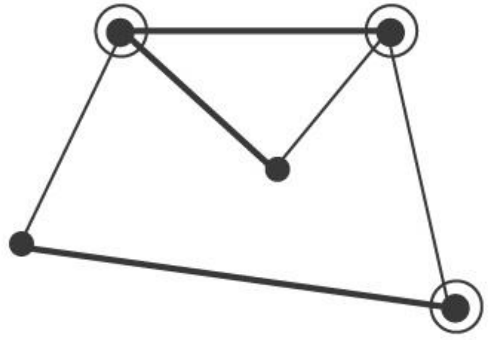
$$\begin{tikzcd}
\text{SAT} \arrow[d] & & \text{SAT} \arrow[d] \\
\text{3-SAT} \arrow[d] & & \text{3-SAT} \arrow[d] \\
\text{INDEPENDENT-SET} & & \text{HAMILTONIAN-CIRCUIT} \arrow[d] \\
& & \text{TSP}
\end{tikzcd}$$

Hitler and P = NP

VERTEX-COVER

- VERTEX-COVER = {<*G, k*>: *G* is an undirected graph and there exists a vertex cover of *G* that contains at most *k* vertices}.

A ***vertex cover*** *C* of a graph *G* = (*V*, *E*) is a subset of *V* such that every edge in *E* touches at least one of the vertices in *C*.



- (V – C) is an independent set

why?

To be able to test every link in a network, it suffices to place monitors at a set of vertices that form a vertex cover of the network.

VERTEX-COVER

Theorem: VERTEX-COVER is NP-complete.

Proof: We must prove:

- VERTEX-COVER is in NP, and
- VERTEX-COVER is NP-hard.

VERTEX-COVER is in NP

Proof: *Ver*(<*G*, *k*, *c*>) =

1. Check that the number of vertices in *c* is at most *min*(*k*, |*V*|). If not, reject.
2. For each vertex *v* in *c* do:

   Find all edges in *E* that have *v* as one endpoint and mark each such edge.
3. Make one final pass through *E* and check whether every edge is marked. If all of them are, accept; otherwise reject.

Timereq(*Ver*) $\in \mathcal{O}(|c| \cdot |E|)$.

Both |*c*| and |*E*| are polynomial in |<*G*, *k*>|. So *Ver* runs in polynomial time.

VERTEX-COVER is NP-Hard

Proof: By reduction from 3-SAT:

Given a wff *f, R* will exploit two kinds of gadgets:

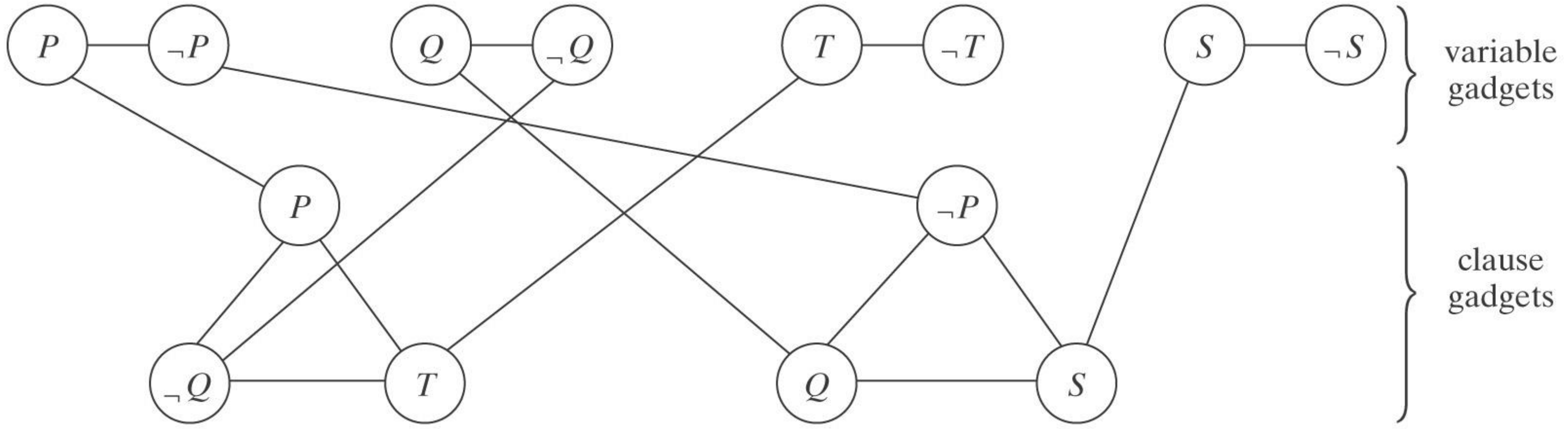
• A variable gadget: For each variable *x* in *f*, *R* will build a simple graph with two vertices and one edge between them. Label one of the vertices *x* and the other one $\neg x$.

• A clause gadget: For each clause *c* in *f*, *R* will build a graph with three vertices, one for each literal in *c*. There will be an edge between each pair of vertices in this graph.

Then *R* will build an edge from every vertex in a clause gadget to the vertex of the variable gadget with the same label.

VERTEX-COVER

$(P \vee \neg Q \vee T) \wedge (\neg P \vee Q \vee S)$



VERTEX-COVER

R(<*f*>) =

1. Build a graph *G* as described above.
2. Let ***k* = *v* + 2*c***. (# variables + 2 * # clauses)
3. Return <*G*, *k*>.

R runs in polynomial time. To show that it is correct, we must show that:

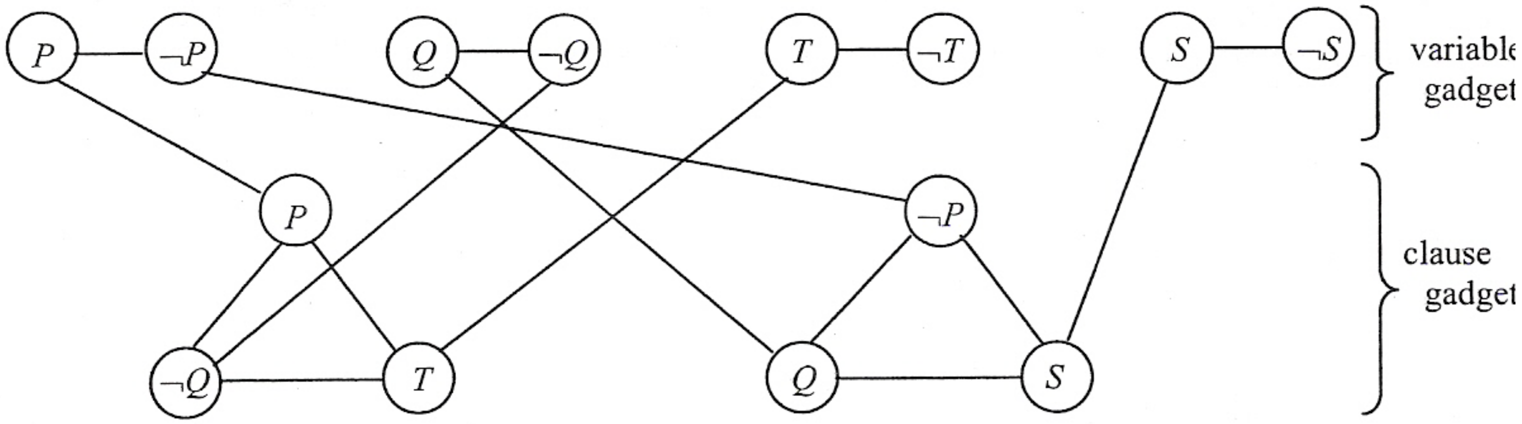
<*f*> $\in$ 3-SAT iff *R*(<*f*>) $\in$ VERTEX-COVER.

VERTEX-COVER

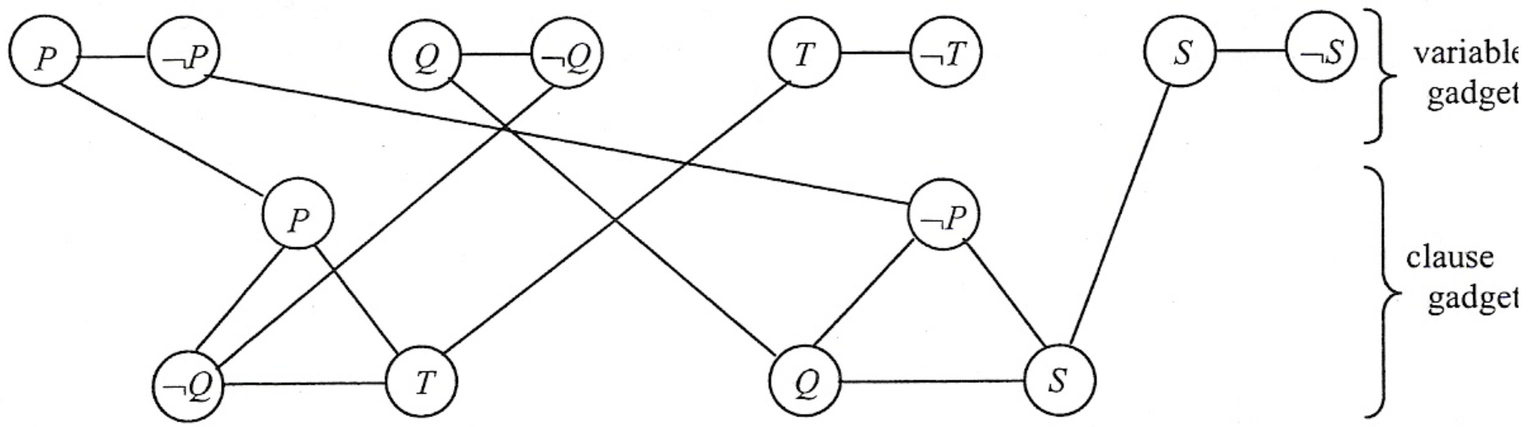
<*f*> ∈ 3-SAT → *R*(<*f*>) ∈ VERTEX-COVER: There exists a satisfying assignment *A* for *f*. *G* contains a vertex cover *C* of size *k*:

1. **From each variable gadget, add to *C* the vertex that corresponds to the literal that is true in *A*.**
2. **Since *A* is a satisfying assignment, there must exist at least one true literal in each clause. Pick one and put the vertices corresponding to the other two into *C*.**

C contains exactly *k* vertices. And it is a cover of *G*:



VERTEX-COVER



C is a cover of *G* because:

• One vertex from every variable gadget is in *C* so all the edges that are internal to the variable gadgets are covered.

• Two vertices from every clause gadget are in *C* so all the edges that are internal to the clause gadgets are covered.

• All the vertices that connect variable gadgets to clause gadgets are covered:

- Two categories: true (connecting to true literal in variable gadget) and false ones
- A true edge must be covered by the true literal (added to C) in variable gadget
- A false edge must be covered by one of the chosen literals in clause gadget because any false literal must have been added to C.

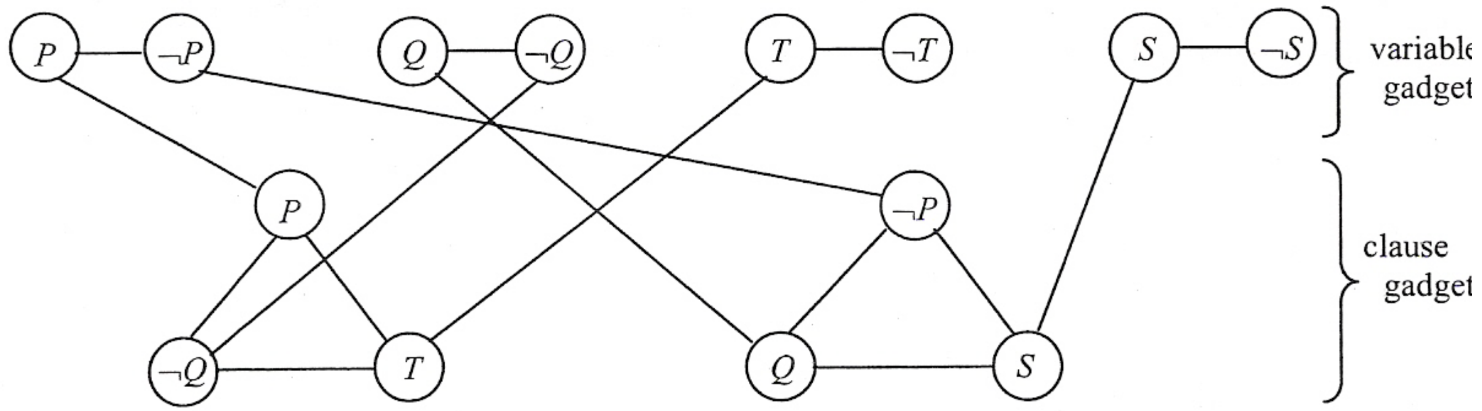
• $(P \vee \neg Q \vee T) \wedge (\neg P \vee Q \vee S)$ P =1, Q=1, T = 0, S = 0

VERTEX-COVER

***R*(<*f*>) ∈ VERTEX-COVER → <*f*> ∈ 3-SAT:** The graph *G* that *R* builds contains a vertex cover *C* of size *k*. *C* must:

- **Contain at least one vertex from each variable gadget** in order to cover the internal edge in the variable gadget.
- **Contain at least two vertices from each clause gadget** in order to cover all three internal edges in the clause gadget.

Satisfying those two requirements uses up all *k* = *v* + 2*c* vertices, so the vertices we have just described are the only vertices in *C*.



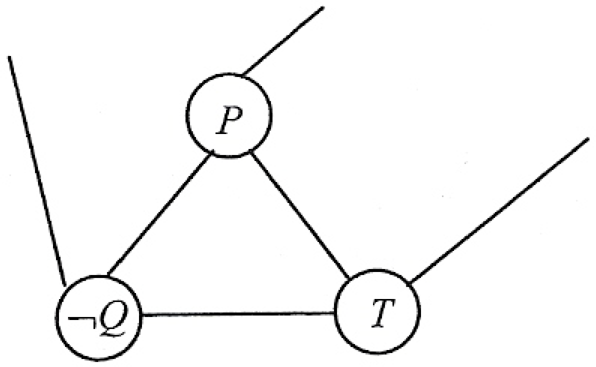
VERTEX-COVER

We can use *C* to show that there exists some satisfying assignment *A* for *f*

• To build *A,* **assign *True* to the vertex (literal) in each variable gadget that is in *C***

  • in each variable gadget, only one vertex can be in C, otherwise, there won't be enough vertices for C to cover all the edges

- Since *C* is a cover for *G*, all six of the edges that connect to vertices in this clause gadget must be covered.
- But we know that only two of the vertices in the gadget are in *C*. They can cover the three internal edges.



- But the three edges that connect to the variable gadgets must also be covered. Only two can be covered by a vertex in the clause gadget. The other one must be covered by its other endpoint, which is in some variable gadget, and in C.
- Since only one vertex in each variable gadget can be in C, no two of these edges (true edges) would connect to the same variable gadget, thus their covering vertices in the variable gadgets can all be assigned True without conflicts.

Small Differences Matter

- Circuit problems
- SAT problems
- Path problems
- Covering problems
- Map coloring problems
- Linear programming problems
- Diophantine equation problems

Two Similar Circuit Problems

- EULERIAN-CIRCUIT, in which we check that there is a circuit that visits every *edge* exactly once, is in P.
- HAMILTONIAN-CIRCUIT, in which we check that there is a circuit that visits every *vertex* exactly once, is NP-complete.

Two Similar SAT Problems

- 2-SAT = {<*w*> : *w* is a wff in Boolean logic, *w* is in 2-conjunctive normal form and *w* is satisfiable} is in P.

$$(\neg P \vee R) \wedge (S \vee \neg T)$$

- 3-SAT = {<*w*> : *w* is a wff in Boolean logic, *w* is in 3-conjunctive normal form and *w* is satisfiable} is NP-complete.

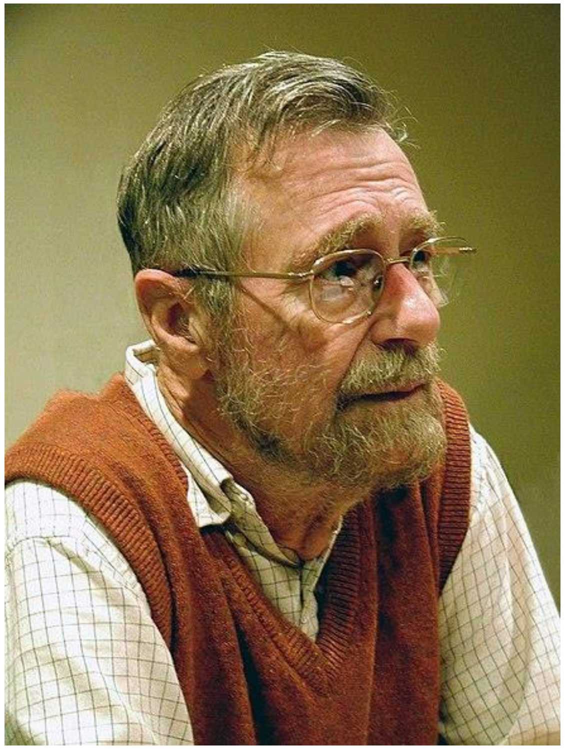
$$(\neg P \vee R \vee T) \wedge (S \vee \neg T \vee \neg W)$$

Two Similar Path Problems

- SHORTEST-PATH = {<*G, u, v, k*>: *G* is an undirected graph, *u* and *v* are vertices in *G, k* $\geq$ 0, and there exists a path from *u* to *v* whose length is **at most** *k*} is in P.

- LONGEST-PATH = {<*G, u, v, k*>: *G* is an undirected graph, *u* and *v* are vertices in *G, k* $\geq$ 0, and there exists a path with no repeated edges from *u* to *v* whose length is **at least** *k*} is NP-complete.

Edsger W. Dijkstra



Edsger Wybe Dijkstra (1930 – 2002)

- Dutch computer scientist and an early pioneer in many research areas of computing science
- Held Schlumberger Centennial Chair in Computer Sciences at the University of Texas at Austin from 1984 until his retirement in 1999.
- **1972 Turing Award**

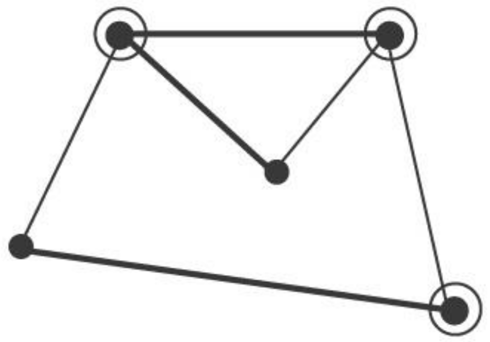
- Background in math and physics, one of the driving forces behind the acceptance of computer programming as a scientific discipline
- Software engineering
- Concurrent programming: semaphores, mutex
- Shortest paths

- E. Allen Emerson **2007 Turing Award**
- Bob Metcalfe **2022 Turing Award**

Two Similar Covering Problems

- An ***edge cover*** *C* of a graph *G* is a subset of the edges of *G* with the property that every vertex of *G* is an endpoint of one of the edges in *C*.

- A ***vertex cover*** *C* of a graph *G* is a subset of the vertices of *G* with the property that every edge of *G* touches one of the vertices in *C*.



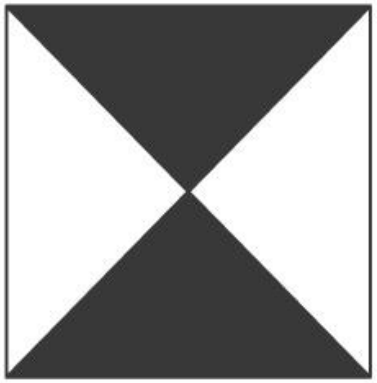
Two Similar Covering Problems

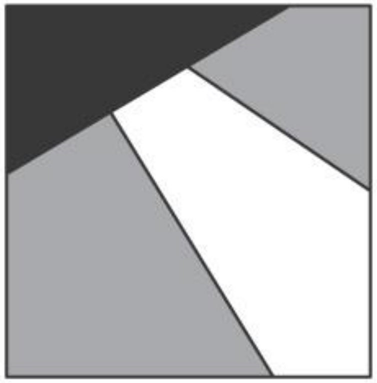
- EDGE-COVER = {<*G, k*>: *G* is an undirected graph and there exists an edge cover of *G* that contains at most *k* edges} is in P.

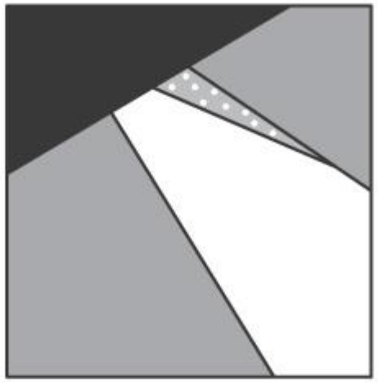
- VERTEX-COVER = {<*G, k*>: *G* is an undirected graph and there exists a vertex cover of *G* that contains at most *k* vertices} is NP-complete.

Three Similar Coloring Problems

Color a planar map so that no two adjacent regions (countries, states, or whatever) have the same color.


(a)


(b)


(c)

How many colors are required?

Three Similar Coloring Problems

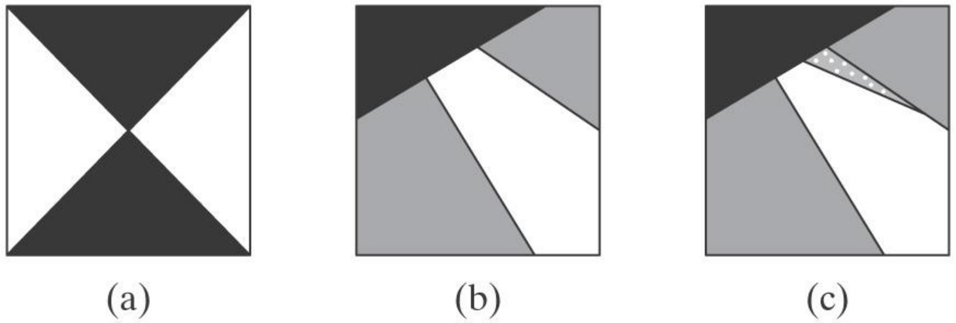
- 2-COLORABLE = {<*m*> : *m* can be colored with 2 colors}.

- 3-COLORABLE = {<*m*> : *m* can be colored with 3 colors}.

- 4-COLORABLE = {<*m*> : *m* can be colored with 4 colors}.

Three Similar Coloring Problems

- 2-COLORABLE = {<*m*> : *m* can be colored with 2 colors}
  - A map is 2-colorable iff it does not contain any point that is the junction of an odd number of regions. In P
- 3-COLORABLE = {<*m*> : *m* can be colored with 3 colors}.
  - 3-COLORABLE is NP-complete.
- 4-COLORABLE = {<*m*> : *m* can be colored with 4 colors}.
  - in P



Chromatic Number

The ***chromatic number*** of a graph is the smallest number of colors required to color its vertices, subject to the constraint that no two adjacent vertices may be assigned the same color.

CHROMATIC-NUMBER = {<*G*, *k*> : *G* is an undirected graph whose chromatic number is no more than *k*}.

CHROMATIC-NUMBER is NP-complete.