Computational complexity

Jim Hefferon

University of Vermont

hefferon.net

Review of Big-O

Definitions

DEFINITION A complexity function f is one that inputs real number arguments and outputs real number values, and (1) has an unbounded domain, so that there is a number $N \in \mathbb{R}^+$ such that $x \geq N$ implies that $f(x)$ is defined, and (2) is eventually nonnegative, so that there is a number $M \in \mathbb{R}^+$ so that $x \geq M$ implies that $f(x) \geq 0$.

DEFINITION Let g be a complexity function. Then Big $\mathcal{O}$ of g, $\mathcal{O}(g)$, is the set of complexity functions f satisfying that there are constants $N, C \in \mathbb{R}^+$ so that if $x \geq N$ then both $g(x)$ and $f(x)$ are defined and $C \cdot g(x) \geq f(x)$. We say that f is $\mathcal{O}(g)$, or that $f \in \mathcal{O}(g)$, or that f is of order at most g, or that $f = \mathcal{O}(g)$.

Think of 'f is $\mathcal{O}(g)$' as meaning that f's order of growth is less than or equal to g's.

Definitions

DEFINITION A complexity function f is one that inputs real number arguments and outputs real number values, and (1) has an unbounded domain, so that there is a number $N \in \mathbb{R}^+$ such that $x \geq N$ implies that $f(x)$ is defined, and (2) is eventually nonnegative, so that there is a number $M \in \mathbb{R}^+$ so that $x \geq M$ implies that $f(x) \geq 0$.

DEFINITION Let g be a complexity function. Then Big $\mathcal{O}$ of g, $\mathcal{O}(g)$, is the set of complexity functions f satisfying that there are constants $N, C \in \mathbb{R}^+$ so that if $x \geq N$ then both $g(x)$ and $f(x)$ are defined and $C \cdot g(x) \geq f(x)$. We say that f is $\mathcal{O}(g)$, or that $f \in \mathcal{O}(g)$, or that f is of order at most g, or that $f = \mathcal{O}(g)$.

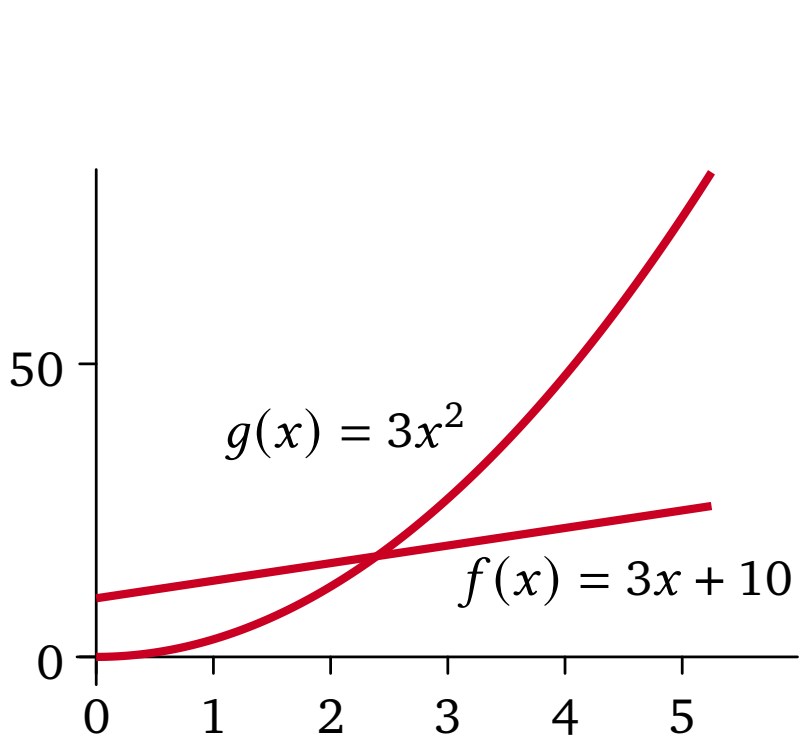
Think of 'f is $\mathcal{O}(g)$' as meaning that f's order of growth is less than or equal to g's.

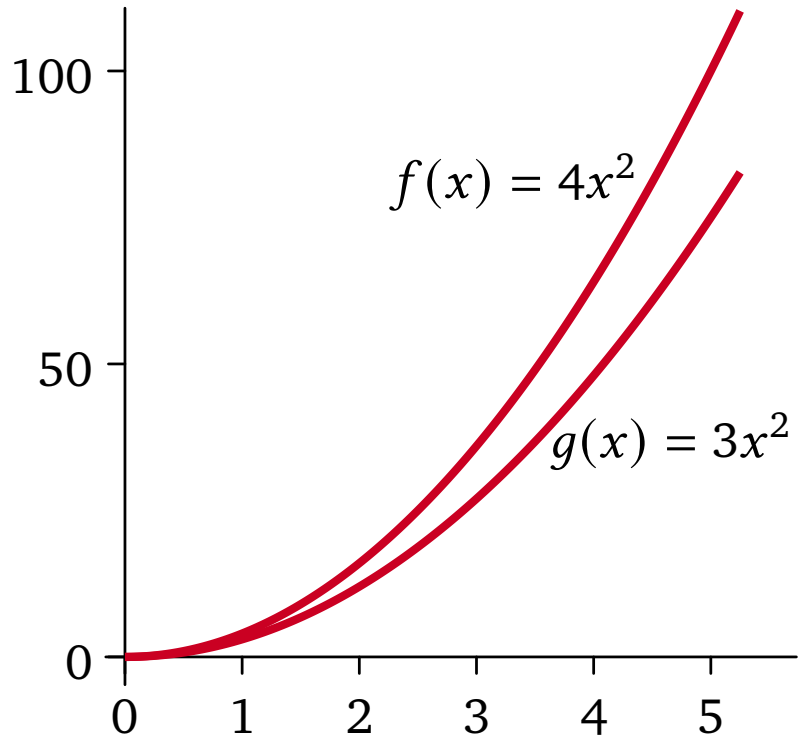
Points:

- Although Turing machines are discrete devices, we find that functions with real number inputs and outputs are suitable. (The term 'complexity function' is not standard but we will find it convenient.)
- We use the letter '$\mathcal{O}$' because this is about the order of growth.
- The '$f = \mathcal{O}(g)$' notation is awkward but standard. The notation '$f \in \mathcal{O}(g)$' is better but not as common.

$\mathcal{O}$ compares orders of growth

For both of these f is $\mathcal{O}(g)$, that is, f's growth rate is less than or equal to g's rate.





- On the left, f is $\mathcal{O}(g)$ despite that $g(x) < f(x)$ for small x's. As x goes to infinity, g races ahead of f. So f's order of growth is strictly less than g's.
- On the right, f is $\mathcal{O}(g)$ despite that $g(x) < f(x)$ for all $x > 0$. As x goes to infinity the two functions track together, so f's order of growth is less than or equal to g's.

Properties of Big-$\mathcal{O}$

LEMMA (ALGEBRAIC PROPERTIES) Let these be complexity functions. (1) If f is $\mathcal{O}(g)$ then for any constant $a \in \mathbb{R}^+$, the function $a \cdot f$ is $\mathcal{O}(g)$. (2) If f_0 is $\mathcal{O}(g_0)$ and f_1 is $\mathcal{O}(g_1)$ then the sum $f_0 + f_1$ is $\mathcal{O}(g)$ where $g(x) = \max(g_0(x), g_1(x))$. So if both f_0 and f_1 are $\mathcal{O}(g)$ then $f_0 + f_1$ is also $\mathcal{O}(g)$. (3) If f_0 is $\mathcal{O}(g_0)$ and f_1 is $\mathcal{O}(g_1)$ then the product $f_0 f_1$ is $\mathcal{O}(g_0 g_1)$.

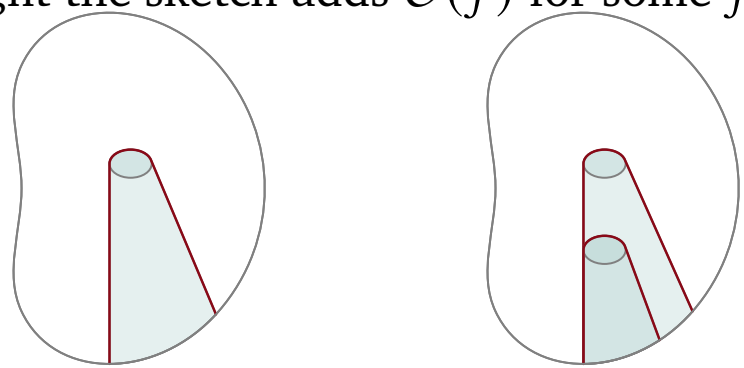
Properties of Big-$\mathcal{O}$

LEMMA (ALGEBRAIC PROPERTIES) Let these be complexity functions. (1) If f is $\mathcal{O}(g)$ then for any constant $a \in \mathbb{R}^+$, the function $a \cdot f$ is $\mathcal{O}(g)$. (2) If f_0 is $\mathcal{O}(g_0)$ and f_1 is $\mathcal{O}(g_1)$ then the sum $f_0 + f_1$ is $\mathcal{O}(g)$ where $g(x) = \max(g_0(x), g_1(x))$. So if both f_0 and f_1 are $\mathcal{O}(g)$ then $f_0 + f_1$ is also $\mathcal{O}(g)$. (3) If f_0 is $\mathcal{O}(g_0)$ and f_1 is $\mathcal{O}(g_1)$ then the product $f_0 f_1$ is $\mathcal{O}(g_0 g_1)$.

DEFINITION Complexity functions f and g have equivalent growth rates or the same order of growth if f is $\mathcal{O}(g)$ and also g is $\mathcal{O}(f)$. We say that f is $\Theta(g)$ (read 'f is Big-Theta of g'), or, what is the same thing, that g is $\Theta(f)$.

LEMMA The Big-$\mathcal{O}$ relation is reflexive, so f is $\mathcal{O}(f)$. It is also transitive, so that if f is $\mathcal{O}(g)$ and g is $\mathcal{O}(h)$ then f is $\mathcal{O}(h)$. Thus having equivalent growth rates, which forces symmetry, is an equivalence relation between functions.

Below, the beans enclose the complexity functions. Slower growing functions are at the bottom. On the left the shaded area contains all of the functions in $\mathcal{O}(g)$, with the set $\Theta(g)$ at the top. On the right the sketch adds $\mathcal{O}(f)$ for some f in $\mathcal{O}(g)$.



Computing Big-$\mathcal{O}$

For many familiar functions the next result is a convenient way to compute orders of growth.

THEOREM Let f, g be complexity functions. Suppose that $\lim_{x\to\infty} f(x)/g(x)$ exists and equals L, which is a member of $\mathbb{R} \cup \{\infty\}$. (1) If $L = 0$ then g grows faster than f, that is, f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$. (2) If $L = \infty$ then f grows faster than g, so that g is $\mathcal{O}(f)$ but f is not $\mathcal{O}(g)$. (3) If L is between 0 and ∞ then the two functions have something like the same growth rates, so that f is $\Theta(g)$ and g is $\Theta(f)$.

Computing Big-$\mathcal{O}$

For many familiar functions the next result is a convenient way to compute orders of growth.

THEOREM Let f, g be complexity functions. Suppose that $\lim_{x\to\infty} f(x)/g(x)$ exists and equals L, which is a member of $\mathbb{R} \cup \{\infty\}$. (1) If $L = 0$ then g grows faster than f, that is, f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$. (2) If $L = \infty$ then f grows faster than g, so that g is $\mathcal{O}(f)$ but f is not $\mathcal{O}(g)$. (3) If L is between 0 and ∞ then the two functions have something like the same growth rates, so that f is $\Theta(g)$ and g is $\Theta(f)$.

To compute the limits we often apply this theorem from Calculus.

THEOREM (L'HÔPITAL'S RULE) Let f and g be complexity functions such that both $f(x) \to \infty$ and $g(x) \to \infty$ as $x \to \infty$, and such that both are differentiable for large enough inputs. If $\lim_{x\to\infty} f'(x)/g'(x)$ exists and equals $L \in \mathbb{R} \cup \{\infty\}$ then $\lim_{x\to\infty} f(x)/g(x)$ also exists and also equals L.

EXAMPLE Consider $f(x) = 3x^2 - 2x + 4$ and $g(x) = (1/2)x^5 - 6$. Take derivatives twice and apply the theorem above to conclude that f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$.

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{3x^2 - 2x + 4}{(1/2)x^5 - 6} = \lim_{x\to\infty} \frac{6x - 2}{(5/2)x^4} = \lim_{x\to\infty} \frac{6}{10x^3} = 0$$

The Order of Growth Hierarchy

LEMMA Logarithmic functions grow more slowly than polynomial functions: if $f(x) = \log_b(x)$ for some base b and $g(x) = a_m x^m + \cdots + a_0$ then f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$. Polynomial functions grow more slowly than exponential functions: where $h(x) = b^x$ for some base $b > 1$ then then g is $\mathcal{O}(h)$ but h is not $\mathcal{O}(g)$.

EXAMPLE Where g is the polynomial function $g(x) = x^2$, this shows that $f(x) = \lg(x)$ is $\mathcal{O}(g)$.

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{1/(x \cdot \ln(2))}{2x} = \lim_{x\to\infty} \frac{1}{x^2 \cdot 2\ln(2)} = 0$$

EXAMPLE This shows that the polynomial $g(x) = x^2$ is $\mathcal{O}(h)$ where h is the exponential $h(x) = 2^x$.

$$\lim_{x\to\infty} \frac{g(x)}{h(x)} = \lim_{x\to\infty} \frac{x^2}{2^x} = \lim_{x\to\infty} \frac{2x}{2^x \cdot \ln(2)} = \lim_{x\to\infty} \frac{2}{2^x \cdot \ln(2)^2} = 0$$

Some orders of growth appear often in practice. Here they are in ascending order, so $\mathcal{O}(f)$ is listed earlier than $\mathcal{O}(g)$ if f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$.

Order	*Name*	*Examples*
$\mathcal{O}(1)$	Bounded	$f(n) = 15$
$\mathcal{O}(\lg(\lg(n)))$	Double logarithmic	$f(n) = \ln(\ln(n))$
$\mathcal{O}(\lg(n))$	Logarithmic	$f_0(n) = \ln(n)$, $f_1(n) = \lg(n^3)$
$\mathcal{O}\big((\lg(n))^c\big)$	Polylogarithmic	$f(n) = (\lg(n))^3$
$\mathcal{O}(n)$	Linear	$f(n) = 3n + 4$
$\mathcal{O}(n\lg(n))$	Log-linear	$f_0(n) = 5n\lg(n) + n$, $f_1(n) = \lg(n!)$
$\mathcal{O}(n^2)$	Polynomial (quadratic)	$f(n) = 5n^2 + 2n + 12$
$\mathcal{O}(n^3)$	Polynomial (cubic)	$f(n) = 2n^3 + 12n^2 + 5$
$\vdots$		
$\mathcal{O}(2^{\text{poly}(\lg(n))})$	Quasipolynomial	$f_0(n) = 2^{(\lg(n))^2 + 3\lg(n)}$, $f_1(n) = n^{\lg(n)}$
$\vdots$		
$\mathcal{O}(2^n)$	Exponential	$f(n) = 10 \cdot 2^n$
$\mathcal{O}(3^n)$	Exponential	$f(n) = 6 \cdot 3^n + n^2$
$\vdots$		
$\mathcal{O}(n!)$	Factorial	$f(n) = 5 \cdot n! + n^{15} - 7$
$\mathcal{O}(n^n)$	–No standard name–	$f(n) = 2 \cdot n^n + 3 \cdot 2^n$

Cobham's thesis

We often split that hierarchy between the polynomial and exponential functions.

A modern computer runs at about 10 GHz, 10 000 million ticks per second, and there are about 3.16×10^7 seconds in a year. The chart shows how long a job will take if we use an algorithm that runs in time $\lg n$, or time n, etc., on inputs of various sizes. Inside the table the times are in years.

	$n = 1$	$n = 10$	$n = 50$	$n = 100$
$\lg n$	–	1.05×10^{-17}	1.79×10^{-17}	2.11×10^{-17}
n	3.17×10^{-18}	3.17×10^{-17}	1.58×10^{-16}	3.17×10^{-16}
$n \lg n$	–	1.05×10^{-16}	8.94×10^{-16}	2.11×10^{-15}
n^2	3.17×10^{-18}	3.17×10^{-16}	7.92×10^{-15}	3.17×10^{-14}
n^3	3.17×10^{-18}	3.17×10^{-15}	3.96×10^{-13}	3.17×10^{-12}
2^n	6.34×10^{-18}	3.24×10^{-15}	3.57×10^{-3}	4.02×10^{12}

Cobham's thesis

We often split that hierarchy between the polynomial and exponential functions.

A modern computer runs at about 10 GHz, 10 000 million ticks per second, and there are about 3.16×10^7 seconds in a year. The chart shows how long a job will take if we use an algorithm that runs in time $\lg n$, or time n, etc., on inputs of various sizes. Inside the table the times are in years.

	$n = 1$	$n = 10$	$n = 50$	$n = 100$
$\lg n$	–	1.05×10^{-17}	1.79×10^{-17}	2.11×10^{-17}
n	3.17×10^{-18}	3.17×10^{-17}	1.58×10^{-16}	3.17×10^{-16}
$n \lg n$	–	1.05×10^{-16}	8.94×10^{-16}	2.11×10^{-15}
n^2	3.17×10^{-18}	3.17×10^{-16}	7.92×10^{-15}	3.17×10^{-14}
n^3	3.17×10^{-18}	3.17×10^{-15}	3.96×10^{-13}	3.17×10^{-12}
2^n	6.34×10^{-18}	3.24×10^{-15}	3.57×10^{-3}	4.02×10^{12}

In the $n = 100$ column, between the first few rows the relative change is an order of magnitude but the absolute times are small. Then we get to the final row. That's not a typo — the last entry really is on order of 10^{12} years. It is huge — the universe is 14×10^9 years old so this computation, even with input size of only 100, would take longer than the age of the universe.

Cobham's thesis

We often split that hierarchy between the polynomial and exponential functions.

A modern computer runs at about 10 GHz, 10 000 million ticks per second, and there are about 3.16×10^7 seconds in a year. The chart shows how long a job will take if we use an algorithm that runs in time $\lg n$, or time n, etc., on inputs of various sizes. Inside the table the times are in years.

	$n = 1$	$n = 10$	$n = 50$	$n = 100$
$\lg n$	–	1.05×10^{-17}	1.79×10^{-17}	2.11×10^{-17}
n	3.17×10^{-18}	3.17×10^{-17}	1.58×10^{-16}	3.17×10^{-16}
$n \lg n$	–	1.05×10^{-16}	8.94×10^{-16}	2.11×10^{-15}
n^2	3.17×10^{-18}	3.17×10^{-16}	7.92×10^{-15}	3.17×10^{-14}
n^3	3.17×10^{-18}	3.17×10^{-15}	3.96×10^{-13}	3.17×10^{-12}
2^n	6.34×10^{-18}	3.24×10^{-15}	3.57×10^{-3}	4.02×10^{12}

In the $n = 100$ column, between the first few rows the relative change is an order of magnitude but the absolute times are small. Then we get to the final row. That's not a typo — the last entry really is on order of 10^{12} years. It is huge — the universe is 14×10^9 years old so this computation, even with input size of only 100, would take longer than the age of the universe.

Cobham's thesis is that the tractable problems — those that are at least conceivably solvable in practice — are the ones for which there is an algorithm whose resource consumption is $\mathcal{O}(p)$ for some polynomial p.

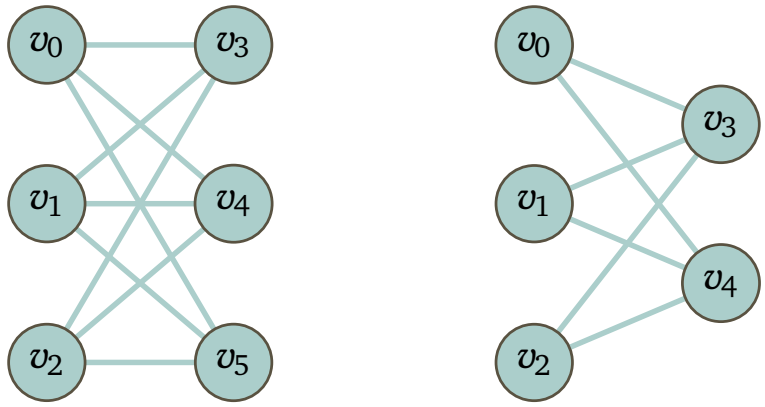
Problems drive development

Problems

We shall list a number of problems that are typical of the kind that researchers in this subject work on. All of the problems here are very well known.

PROBLEM (Hamiltonian Circuit) Given a graph, decide if it contains a cyclic path that includes each vertex once and only once.

EXAMPLE The graph on the left has a Hamiltonian circuit. The one on the right does not.



On the left we can easily trace out a Hamiltonian circuit. On the right although we can visit each vertex once and only once, we cannot get back to the start to make a circuit. For, all edges connect a vertex on the left with one on the right and there are more vertices on the left than the right. To make a circuit we'd have to visit two left vertices in a row, but the cross-side arrangement of edges doesn't allow that.

Knight's tour

A special case is the Knight's Tour problem, to use a chess knight to make a circuit of the squares on the board. (Recall that a knight moves three squares at a time, with the first two squares in one direction and then the third one perpendicular to that direction.)

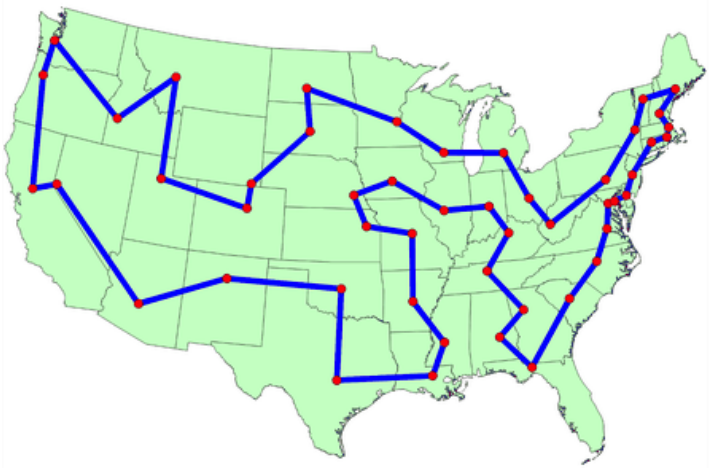
42	59	44	9	40	21	46	7
61	10	41	58	45	8	39	20
12	45	60	55	22	57	6	47
53	62	11	30	25	28	19	38
32	13	54	27	56	23	48	5
63	52	31	24	29	26	37	18
14	33	2	51	16	35	4	49
1	64	15	34	3	50	17	36

This is the solution given by L Euler. In graph terms, there are sixty four vertices, representing the board squares. An edge goes between two vertices if they are connected by a single knight move. Knight's Tour asks for a Hamiltonian circuit of that graph.

Traveling Salesman

If the graph's edges have weights then we can look for the Hamiltonian circuit of least total weight.

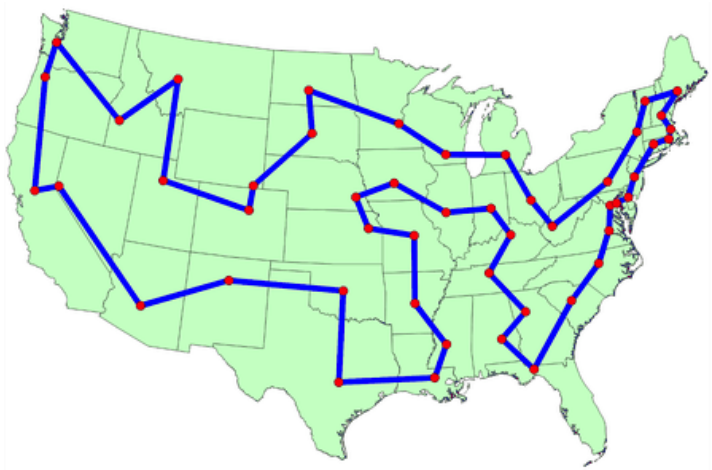
PROBLEM (Traveling Salesman) Given a weighted undirected graph, where we call the vertices $S = \{c_0, \ldots c_{k-1}\}$ 'cities' and we call the edge weight $d(c_i, c_j) \in \mathbb{N}^+$ for $i \neq j$ the 'distance' between the cities, find the shortest-distance circuit that visits every city and returns back to the start.



Traveling Salesman

If the graph's edges have weights then we can look for the Hamiltonian circuit of least total weight.

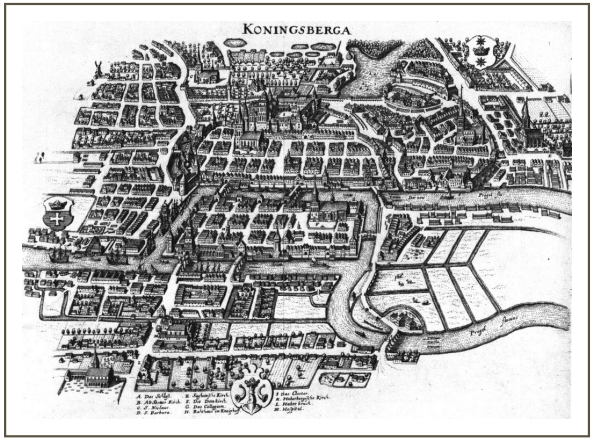
PROBLEM (Traveling Salesman) Given a weighted undirected graph, where we call the vertices $S = \{c_0, \ldots c_{k-1}\}$ 'cities' and we call the edge weight $d(c_i, c_j) \in \mathbb{N}^+$ for $i \neq j$ the 'distance' between the cities, find the shortest-distance circuit that visits every city and returns back to the start.

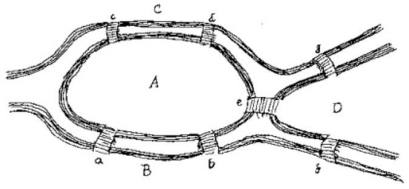


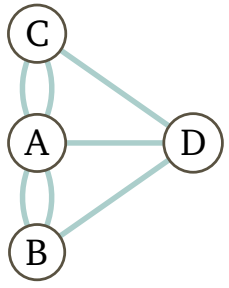
As stated this is an optimization problem, but we can recast it as a decision problem. Introduce a threshold bound $B \in \mathbb{N}$ and change the problem to 'decide if there is a circuit of length less than B'. If we have an algorithm to solve that decision problem then we can use it to find the length of the shortest circuit by first deciding if there is a trip bounded in length by $B = 1$, then deciding if there is a trip bounded in length by $B = 2$, etc.

Euler Circuit

PROBLEM (Euler Circuit) Given a graph, find a circuit that traverses each edge once and only once, or find that no such circuit exists.



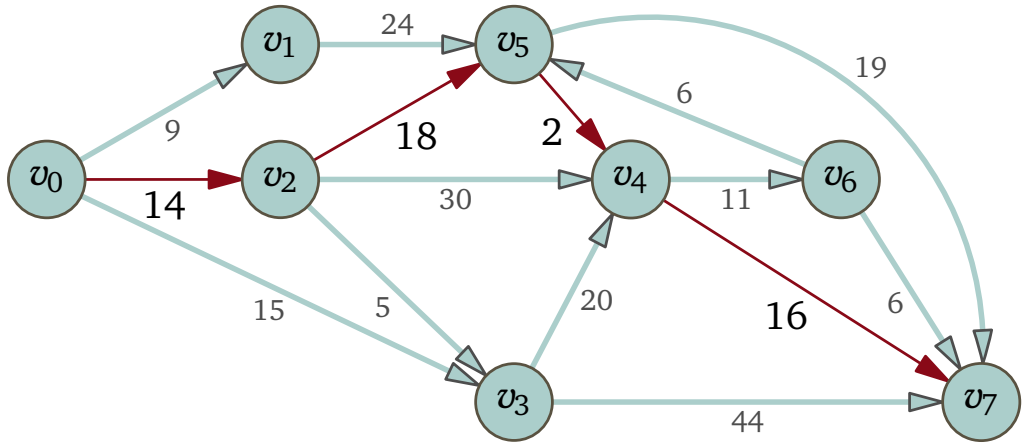




The Euler Circuit problem and the Hamiltonian Circuit problem seem much alike. In the one we try to traverse each edge while in the other we try to visit each vertex. Perhaps it is a surprise then that we know of a fast algorithm to solve the first problem but we know of no fast algorithm to solve the second.

Shortest path

PROBLEM (Shortest Path) Given a weighted graph and two vertices, find the least-weight path between them, or find that no path exists.



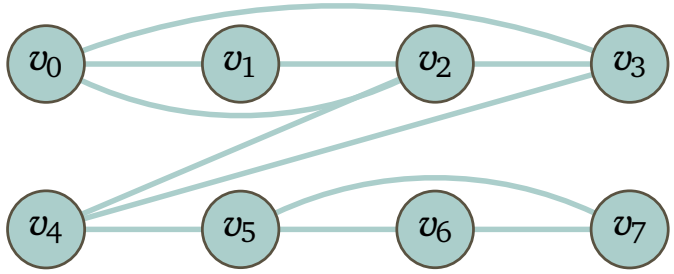
Between v_0 and v_7 this is the shortest path.

$$v_0 \longrightarrow v_2 \longrightarrow v_5 \longrightarrow v_4 \longrightarrow v_7$$

Graph Colorability

PROBLEM (Graph Colorability) Given a graph and a number $k \in \mathbb{N}$, decide whether the graph is k-colorable, whether we can partition its vertices into k-many sets, $\mathcal{N} = \mathcal{C}_0 \cup \cdots \cup \mathcal{C}_{k-1}$, such that no two same-set vertices are connected.

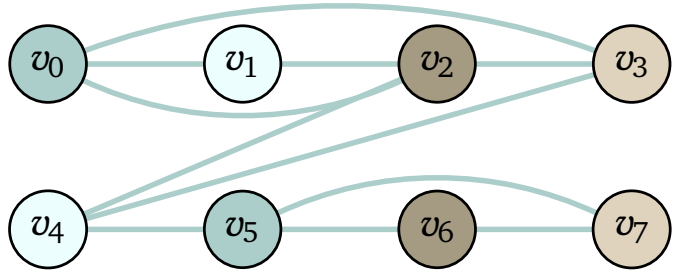
EXAMPLE A marina rents boats. Today there are eight reservations, modelled here as eight vertices. If reservations have overlapping times then we connect the vertices. Can we cover with four boats?



Graph Colorability

PROBLEM (Graph Colorability) Given a graph and a number $k \in \mathbb{N}$, decide whether the graph is k-colorable, whether we can partition its vertices into k-many sets, $\mathcal{N} = \mathcal{C}_0 \cup \cdots \cup \mathcal{C}_{k-1}$, such that no two same-set vertices are connected.

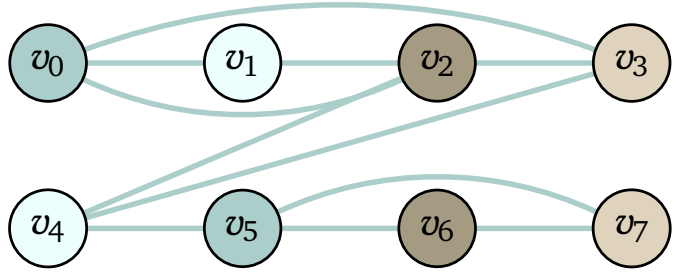
EXAMPLE A marina rents boats. Today there are eight reservations, modelled here as eight vertices. If reservations have overlapping times then we connect the vertices. Can we cover with four boats? Yes.



Graph Colorability

PROBLEM (Graph Colorability) Given a graph and a number $k \in \mathbb{N}$, decide whether the graph is k-colorable, whether we can partition its vertices into k-many sets, $\mathcal{N} = \mathcal{C}_0 \cup \cdots \cup \mathcal{C}_{k-1}$, such that no two same-set vertices are connected.

EXAMPLE A marina rents boats. Today there are eight reservations, modelled here as eight vertices. If reservations have overlapping times then we connect the vertices. Can we cover with four boats? Yes.



PROBLEM (Chromatic Number) Given a graph, find the smallest number $k \in \mathbb{N}$ such that the graph is k-colorable.

EXAMPLE Sudoku ask you to construct a 9-coloring of a graph, given a partial 9-coloring. The graph has vertex for each cell, so there are 81 vertices. As for edges, there is an edge between the distinct vertices if and only if their cells are in the same column, or row, or the same 3×3 grid.

Satisfiability

Problem (Satisfiability, SAT) Decide if a given Boolean expression is satisfiable.

Problem (3-Satisfiability, 3-SAT) Given a propositional logic formula in Conjunctive Normal form in which each clause has at most three variables, decide if it is satisfiable.

Example Is $(P \vee Q \vee R) \wedge (\neg P \vee \neg Q \vee R) \wedge (P \vee \neg Q \vee \neg R) \wedge (\neg P \vee Q \vee \neg R)$ satisfiable?

P	Q	R	$P \vee Q \vee R$	$\neg P \vee \neg Q \vee R$	$P \vee \neg Q \vee \neg R$	$\neg P \vee Q \vee \neg R$	φ
F	F	F					
F	F	T					
F	T	F					
F	T	T					
T	F	F					
T	F	T					
T	T	F					
T	T	T					

Satisfiability

PROBLEM (Satisfiability, SAT) Decide if a given Boolean expression is satisfiable.

PROBLEM (3-Satisfiability, 3-SAT) Given a propositional logic formula in Conjunctive Normal form in which each clause has at most three variables, decide if it is satisfiable.

EXAMPLE Is $(P \vee Q \vee R) \wedge (\neg P \vee \neg Q \vee R) \wedge (P \vee \neg Q \vee \neg R) \wedge (\neg P \vee Q \vee \neg R)$ satisfiable?

P	Q	R	$P \vee Q \vee R$	$\neg P \vee \neg Q \vee R$	$P \vee \neg Q \vee \neg R$	$\neg P \vee Q \vee \neg R$	φ
F	F	F	F	T	T	T	F
F	F	T	T	T	T	T	T
F	T	F	T	T	T	T	T
F	T	T	T	T	F	T	F
T	F	F	T	T	T	T	T
T	F	T	T	T	T	F	F
T	T	F	T	F	T	T	F
T	T	T	T	T	T	T	T

Problems: graphs

PROBLEM (Vertex-to-Vertex Path) Given a graph and two vertices, find if the second is reachable from the first.

(This problem is often known as st-conn.)

PROBLEM (Minimum Spanning Tree) Given a weighted undirected graph, find a subgraph containing all the vertices of the original graph such that its edges have a minimum total.

PROBLEM (Vertex Cover) Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a size B set of vertices, C, such that for any edge, at least one of its ends is a member of C.

PROBLEM (Clique) Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a size B set vertices such that any two are connected.

PROBLEM (Max Cut) A graph cut partitions the vertices into two disjoint subsets. The cut set contains the edges with a vertex in each subset. The Max Cut problem is to find the partition with the largest cut set.

Problems: combinatorics

Problem (Three Dimensional Matching) Let the sets X, Y, Z all have the same number of elements, n. Given as input a set $M \subseteq X \times Y \times Z$, decide if there is a matching, a set $\hat{M} \subseteq M$ containing n elements such that no two of the triples in $\hat{M}$ agree on their first coordinates, or their second or third coordinates either.

Problem (Subset Sum) Given a multiset of natural numbers $S = \{ n_0, \ldots n_{k-1} \}$ and a target $T \in \mathbb{N}$, decide if a subset of S sums to the target.

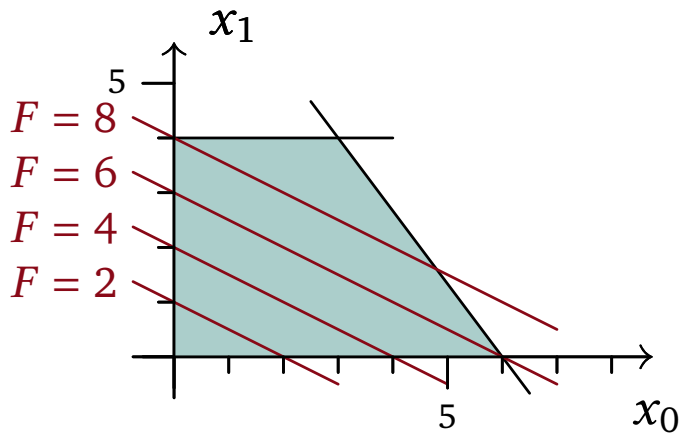
Problem (Knapsack) Given a finite multiset S whose elements s have a natural number weight $w(s)$ and value $v(s)$, and also given a weight bound B and a value target T, find a subset $\hat{S} \subseteq S$ whose elements have a total weight less than or equal to the bound and total value greater than or equal to the target.

Problem (Partition) Given a finite multiset A such that each element has an associated natural number size $s(a)$, decide if the set splits into two, $\hat{A}$ and $A - \hat{A}$, so that the total of the sizes is the same, $\sum_{a \in \hat{A}} s(a) = \sum_{a \notin \hat{A}} s(a)$.

Problems: linear programming

PROBLEM (Linear Progamming) Optimize a linear function $F(x_0, \ldots x_n) = c_0 x_0 + \cdots + c_n x_n$ subject to linear constraints, ones of the form $a_{i,0} x_0 + \cdots + a_{i,n} x_n \leq b_i$ or $a_{i,0} x_0 + \cdots + a_{i,n} x_n \geq b_i$.

EXAMPLE Maximize $F(x_0, x_1) = x_0 + 2x_1$ subject to $4x_0 + 3x_1 \leq 24$, $x_1 \leq 4$, $x_0 \geq 0$ and $x_1 \geq 0$. The shaded region has the points that satisfy all the inequalities; these are said to be 'feasible' points.



The level lines of F indicate that the maximum is at $(x_0, x_1) = (3, 4)$.

Problems: games

PROBLEM (Crossword) Given an $n \times n$ grid and a set of $2n$-many strings, each of length n, decide if the words can be packed into the grid.

PROBLEM (Fifteen Game) Given an $n \times n$ grid holding tiles numbered $1, \ldots n^2 - 1$, and a blank, find the minimum number of moves that will put the tile numbers into ascending order. A move consists of switching a tile with an adjacent blank.

Problems: divisibility

Problem (Divisor) Given a number $n \in \mathbb{N}$, find a nontrivial divisor.

Problem (Prime Factorization) Given a number $n \in \mathbb{N}$, produce its decomposition into a product of primes.

Problem (Primality) Given a number $n \in \mathbb{N}$, determine if it is prime; that is, decide if there are no numbers a that divide n and such that $1 < a < n$.

Problems: divisibility

PROBLEM (Divisor) Given a number $n \in \mathbb{N}$, find a nontrivial divisor.

PROBLEM (Prime Factorization) Given a number $n \in \mathbb{N}$, produce its decomposition into a product of primes.

PROBLEM (Primality) Given a number $n \in \mathbb{N}$, determine if it is prime; that is, decide if there are no numbers a that divide n and such that $1 < a < n$.

For many years the consensus among experts was that Primality was probably quite hard. After all, for centuries many very smart people had worked on the question and none of them had produced a fast test. But in 2002 M Agrawal, N Kayal, and N Saxena proved that primality testing can be done in time polynomial in the number of digits of the number. At this point refinments of their technique run in $\mathcal{O}(n^6)$.

Nitin Saxena (b 1981), Neeraj Kayal (b 1979), Manindra Agrawal (b 1966)

Reflections on problems

Problems, algorithms, and programs

- A problem is a job, a task. It is a usually uniform family of tasks, with an unbounded number of instances. For a sense of 'family', contrast the general Shortest Path problem with that of finding the shortest path between Los Angeles and New York. The first is a family while the second is an instance. We are more likely to talk about the family, both because any conclusion about the first subsumes the second and also because the first feels more natural. We are focused on problems that can be solved with a mechanism, although we continue to be interested to learn that a problem cannot be solved mechanically at all.
- An algorithm is an effective way to solve a problem. An algorithm is not a program, although it should be described in a way that is detailed enough that implementing it is routine for an experienced professional. The description should be complete enough to analyze its Big $\mathcal{O}$ behavior.
  One subtle point about algorithms is that while they are abstractions, they are nonetheless based on a computing model. An algorithm that is based on a Turing machine model for adding one to an input would be very different than an algorithm to do the same task on a model that is like an everyday computer.
- A program is an implementation of an algorithm, expressed in a formal computer language and often designed to be executed on a specific computing platform.

Other models of computation

Besides the Turing machine, another model that is widely used in this context is the Random Access machine (RAM). Whereas a Turing machine cell stores only a single symbol, so that big numbers need multiple cells, on a RAM model machine each register holds an entire integer. And whereas to get to a cell a Turing machine may spend lots of steps traversing the tape, the RAM model gets each register's contents in a single step.

Representations

We shall assume that for our algorithms all of the inputs and outputs are in some reasonably efficient bitstring representation.

Thus, we shall understand "given two numbers $n_0, n_1 \in \mathbb{N}$" to mean 'given the bitstring representation of two numbers'. And we shall take "let $\mathcal{G}$ be a graph" to mean 'given a reasonably efficient representation of a graph $\mathcal{G}$'.

Having noted it, we will from now on generally ignore it.

Types of problems

- A function problem asks that the algorithm have a single output for each input.

Types of problems

- A function problem asks that the algorithm have a single output for each input.
- A search problem asks for an algorithm that, while there may be many solutions in the search space, will stop when it has found any one.

Types of problems

- A function problem asks that the algorithm have a single output for each input.
- A search problem asks for an algorithm that, while there may be many solutions in the search space, will stop when it has found any one.
- An optimization problem asks for an algorithm that gives a solution that is best according to some metric.

Types of problems

- A function problem asks that the algorithm have a single output for each input.
- A search problem asks for an algorithm that, while there may be many solutions in the search space, will stop when it has found any one.
- An optimization problem asks for an algorithm that gives a solution that is best according to some metric.
- A decision problem asks for an algorithm with a 'Yes' or 'No' answer. (So it is a Boolean function. But is important enough to list on its own, instead of subsumed as a function problem.)

Types of problems

- A function problem asks that the algorithm have a single output for each input.
- A search problem asks for an algorithm that, while there may be many solutions in the search space, will stop when it has found any one.
- An optimization problem asks for an algorithm that gives a solution that is best according to some metric.
- A decision problem asks for an algorithm with a 'Yes' or 'No' answer. (So it is a Boolean function. But is important enough to list on its own, instead of subsumed as a function problem.)
- Often a decision problem is expressed as a language recognition problem, where we are given some language and asked for an algorithm to decide if its input is a member of that language.

The last is the one that we will work with most often.

Problems

In this chapter, problem usually means a decision problem for a language. If we get a problem that isn't a language decision problem then we will often recast it as one.

EXAMPLE Subset Sum inputs a set S of numbers and a target number T, and then decides whether a subset of S adds to the target. We can recast it as the problem of deciding membership in this language.

$$\mathcal{L} = \{ \langle S, T \rangle \mid \text{A subset of } S \text{ adds to } T \}$$

Restating the problem as this language leaves it essentially unchanged. Instead of asserting 'there is a subset of $S = \{19, 21, 52, 106\}$ that adds to $T = 40$' we may say '$\langle S, T \rangle \in \mathcal{L}$'.

EXAMPLE Consider Vertex-to-Vertex Path, where the input is a graph and two vertices in the graph and the problem is to decide if there is a path between the two. We recast it as the decision problem for this language.

$$\mathcal{L} = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{there is a path in } \mathcal{G} \text{ between the vertices } v \text{ and } \hat{v} \}$$

EXAMPLE To recast Shortest Path, introduce a bound $B \in \mathbb{N}$ parameter.

$$\mathcal{L}_B = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{there is a path between the vertices of length bounded by } B \}$$

To find the length of the shortest path, see if there is a $B = 1$ path, then see if there is a $B = 2$ path, etc.

Classes

DEFINITION A complexity class is a collection of languages.

The term 'complexity' is there because these collections are often associated with some resource specification, so that for instance one class is the collection of languages that are accepted by a Turing machine in quadratic time.

Another example is that we might consider all language decision problems for which there is an algorithm that runs in space $\mathcal{O}(n)$. Still another is the set of languages where the decision problem for the language's complement has an algorithm that runs in polynomial time.

Complexity classes

P

DEFINITION A language decision problem $\mathcal{L}$ is a member of the class P if there is an algorithm to decide membership in $\mathcal{L}$ that on a deterministic Turing machine runs in polynomial time.

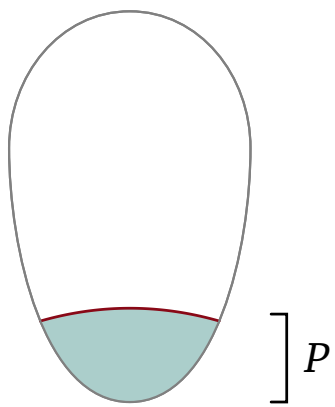
Here are some languages whose decision problems are in P: (1) the language of correct sums, $\mathcal{L}_1 = \{ \langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c \}$, (2) the language of sorted strings, $\mathcal{L}_2 = \{ \sigma \in \{ \mathsf{a}, \ldots \mathsf{z} \}^* \mid \sigma \text{ is in alphabetical order} \}$, (3) the language of correct matrix multiplications, $\mathcal{L}_3 = \{ \langle A, B, C \rangle \mid \text{the matrices are such that } AB = C \}$, (4) the language of primes, $\mathcal{L}_4 = \{ 1^k \mid k \text{ is prime} \}$, (5) the language of nodes with five neighbors, $\mathcal{L}_5 = \{ \langle \mathcal{G}, v \rangle \mid \text{vertex } v \text{ in } \mathcal{G} \text{ has exactly five neighbors} \}$.

P

DEFINITION A language decision problem $\mathcal{L}$ is a member of the class P if there is an algorithm to decide membership in $\mathcal{L}$ that on a deterministic Turing machine runs in polynomial time.

Here are some languages whose decision problems are in P: (1) the language of correct sums, $\mathcal{L}_1 = \{ \langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c \}$, (2) the language of sorted strings, $\mathcal{L}_2 = \{ \sigma \in \{ \mathsf{a}, \ldots \mathsf{z} \}^* \mid \sigma \text{ is in alphabetical order} \}$, (3) the language of correct matrix multiplications, $\mathcal{L}_3 = \{ \langle A, B, C \rangle \mid \text{the matrices are such that } AB = C \}$, (4) the language of primes, $\mathcal{L}_4 = \{ 1^k \mid k \text{ is prime} \}$, (5) the language of nodes with five neighbors, $\mathcal{L}_5 = \{ \langle \mathcal{G}, v \rangle \mid \text{vertex } v \text{ in } \mathcal{G} \text{ has exactly five neighbors} \}$.

Here, the bean encloses all all decision problems $\mathcal{L} \subseteq \mathbb{B}^*$. Problems with slower-growing solution algorithms are at the bottom. Shaded is the class P. This class contains problems for which there is a solution algorithm that is $\mathcal{O}(1)$, problems for which there is an algorithm that is $\mathcal{O}(\lg n)$, problems with an algorithm that is $\mathcal{O}(n)$, etc.



Effect of the model of computation

Different models of computation work at different speeds. In particular, our experience working with Turing machines leads us to perceive that they are slow; does the machine we standardize on affect how we classify problems?

Close analysis shows that Turing machines are slower than other natural models by no more than a factor of n^2 or n^3. (We take a model to be 'natural' if it was not invented just to be a counterexample to this.) Thus if we have a problem for which there is a $\mathcal{O}(n)$ algorithm on another model then when translated to a Turing machine algorithm it may be $\mathcal{O}(n^4)$. Under this scenario, a problem that falls in the class P with one natural model, including Turing machines, also falls there using other natural models.

Effect of the model of computation

Different models of computation work at different speeds. In particular, our experience working with Turing machines leads us to perceive that they are slow; does the machine we standardize on affect how we classify problems?

Close analysis shows that Turing machines are slower than other natural models by no more than a factor of n^2 or n^3. (We take a model to be 'natural' if it was not invented just to be a counterexample to this.) Thus if we have a problem for which there is a $\mathcal{O}(n)$ algorithm on another model then when translated to a Turing machine algorithm it may be $\mathcal{O}(n^4)$. Under this scenario, a problem that falls in the class P with one natural model, including Turing machines, also falls there using other natural models.

REMARK There is a possible counterexample to the contention about there being an at-most polytime delay between natural models of physically-realizable devices. Under the Quantum Computing model there are several problems with polytime solutions, including integer factorization, for which we do not know of any polytime solution in a natural non-quantum model. Whether quantum computers will ever be practical physical devices is not at this moment perfectly clear but scientists and engineers are making great progress. (Also, conceivably there are polytime solutions for these problems on Turing machines.)

Naturalness of the class P

We will talk a lot about P so we will take a moment to reflect on why.

- Cobham's Thesis asserts that problems for which there is a polytime solution form a very important class in practice.
- As just mentioned, if a problem is in P based on any familiar model including Turing machines, RAM, and Racket programs, then it is in P for all of them.
- We'd like that if two things, f and g, are easy to compute then a simple combination of the two is also easy.
  More precisely, fix two total functions $f, g\colon \mathbb{N} \to \mathbb{N}$ and consider these.

$$\mathcal{L}_f = \{\,\text{str}(\langle n, f(n)\rangle) \in \mathbb{B}^* \times \mathbb{B}^* \;\big|\; n \in \mathbb{N}\,\}$$
$$\mathcal{L}_g = \{\,\text{str}(\langle n, g(n)\rangle) \in \mathbb{B}^* \times \mathbb{B}^* \;\big|\; n \in \mathbb{N}\,\}$$

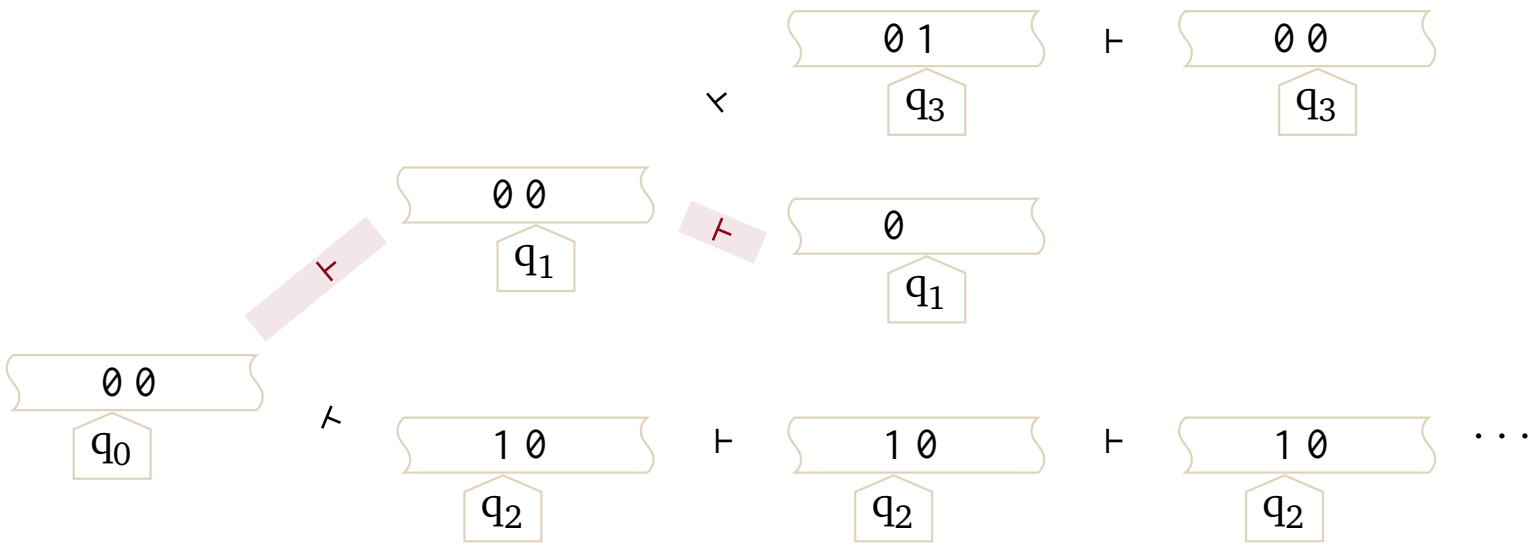
  (Recall that str(...) means that we represent the argument reasonably efficiently as a bitstring.) With that recasting of functions as languages, P is closed under function addition, scalar multiplication by an integer, subtraction, multiplication, and composition. It is also closed under language concatenation and the Kleene star operator. It is the smallest nontrivial class with these appealing properties.

Nondeterministic polynomial time

Nondeterminism

Recall from the chapter on Finite State machines that a machine is nondeterminstic if from a present configuration and input it may pass to a next configuration where the number of next states is zero, or one, or more than one. This contrasts with a deterministic machine where the number of next states is always exactly one.

DEFINITION A nondeterministic Turing machine $\mathcal{P}$ is a finite set of instructions $q_p T_p T_n q_n \in Q \times \Sigma \times (\Sigma \cup \{\mathsf{L}, \mathsf{R}\}) \times Q$, where Q is a finite set of states and Σ is a finite set of tape alphabet characters, which contains at least two members, including blank, and does not contain the characters L or R. Some of the states, $F \subseteq Q$, are accepting states. The association of the present state and tape character with what happens next is given by the transition function, $\Delta : Q \times \Sigma \to \mathcal{P}((\Sigma \cup \{\mathsf{L}, \mathsf{R}\}) \times Q)$.



Recall also that we have two mental models of how these devices operate. In the first, the machine is unboundedly parallel — when it is called on to go to multiple states, it forks a child process for each one. That is, the machine's computation history is a tree and it simultaneously computes all branches.

In the second model, the machine guesses which next state to follow, or is told which next state to follow by a demon, and then it deterministically computes a check of that branch of the computation tree.

Recall also that we have two mental models of how these devices operate. In the first, the machine is unboundedly parallel — when it is called on to go to multiple states, it forks a child process for each one. That is, the machine's computation history is a tree and it simultaneously computes all branches.

In the second model, the machine guesses which next state to follow, or is told which next state to follow by a demon, and then it deterministically computes a check of that branch of the computation tree.

Described in the language of the first model, we say that an input string is accepted if one of the branches on the tree is an accepting branch, and it does not accept the input if no branch ends in an accepting state. In terms of the second model, this means that the input is accepted if there is a sequences of guesses that the machine could make, or a sequence of branching hints that the demon could produce, that the machine can verify leads to an accepting state.

These two models are equivalent in that for a given nondeterministic machine they yield the same collection of accepted strings.

Language decider

The description on the prior slide raises two points. First, once some branch accepts the input we might as well stop the computation, so we can take that accepting computations always halt. Second, "no branch ends" seemingly could mean that in a non-accepting computation some branches do not accept because their computation fails to halt. But we are using these machines as language deciders and we shall want to time them, including how long they take to not accept. So we will only define language-deciding when all branches halt.

DEFINITION Let $\mathcal{P}$ be a nondeterministic Turing machine such that every branch in the computation tree, every sequence of valid transitions from the starting configuration, is finite. Then $\mathcal{P}$ accepts an input string if at least one branch ends in an accepting state and otherwise rejects it. The machine decides a language $\mathcal{L}$ when for every input string σ, if $\sigma \in \mathcal{L}$ then $\mathcal{P}$ accepts it while if $\sigma \notin \mathcal{L}$ then $\mathcal{P}$ rejects it.

Because we require that every branch halts, we can determine in a finite time that the machine rejects the input.

LEMMA For Turing machines, deterministic and nondeterministic machines decide the same languages.

PF. One direction is easy. A deterministic Turing machine is a special case of a nondeterministic one. So if a deterministic machine decides a language then a nondeterministic one does also.

Conversely, let the nondeterministic Turing machine $\mathcal{P}$ decide the language $\mathcal{L}$. Consider a deterministic machine $\mathcal{Q}$ that does a breadth-first search of $\mathcal{P}$'s computation tree. We will show that it decides the same language. Fix an input σ. If $\mathcal{P}$ accepts σ then the search done by $\mathcal{Q}$ will eventually find that node in the computation tree and then $\mathcal{Q}$ accepts σ. If $\mathcal{P}$ does not accept σ then every branch in its computation tree halts in a state that is not accepting. There is a longest such branch by König's lemma. So the breadth-first search will eventually exhaust all branches and then $\mathcal{Q}$ rejects σ. ∎

Speed

Is this propositional logic expression satisfiable?

$$E = (P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee \neg R) \wedge (Q \vee R) \qquad (*)$$

The natural approach is to compute a truth table. The table below shows that it is satisfiable, because the TTF row ends in a T.

P	Q	R	$P \vee Q$	$P \vee \neg Q$	$\neg P \vee Q$	$\neg P \vee \neg Q \vee \neg R$	$Q \vee R$	$(*)$
F	F	F	F	T	T	T	F	F
F	F	T	F	T	T	T	T	F
F	T	F	T	F	T	T	T	F
F	T	T	T	F	T	T	T	F
T	F	F	T	T	F	T	F	F
T	F	T	T	T	F	T	T	F
T	T	F	T	T	T	T	T	T
T	T	T	T	T	T	F	T	F

Speed

Is this propositional logic expression satisfiable?

$$E = (P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee \neg R) \wedge (Q \vee R) \qquad (*)$$

The natural approach is to compute a truth table. The table below shows that it is satisfiable, because the TTF row ends in a T.

P	Q	R	$P \vee Q$	$P \vee \neg Q$	$\neg P \vee Q$	$\neg P \vee \neg Q \vee \neg R$	$Q \vee R$	$(*)$
F	F	F	F	T	T	T	F	F
F	F	T	F	T	T	T	T	F
F	T	F	T	F	T	T	T	F
F	T	T	T	F	T	T	T	F
T	F	F	T	T	F	T	F	F
T	F	T	T	T	F	T	T	F
T	T	F	T	T	T	T	T	T
T	T	T	T	T	T	F	T	F

As to runtime, the number of table rows grows exponentially: it is 2 raised to the number of input variables. Going through the rows one at a time would be very slow.

Each line of the truth table is easy; the issue is that there are lots of lines. This situation is perfectly suited for a machine that is unboundedly parallel. For each line we could fork a child process. Each of these children is done quickly, certainly in polytime, and if they are working in parallel then the whole thing is polytime. If at the end any child is holding a T then the expression as a whole is satisfiable. That is, while a serial machine appears to require exponential time, a nondeterministic machine does this job in polytime.

NP

DEFINITION The complexity class *NP* is the set of languages for which there is a nondeterministic Turing machine decider $\mathcal{P}$ that runs in polytime, meaning that there is a polynomial p such that on input σ, all branches of $\mathcal{P}$'s computation halt in time $p(|\sigma|)$.

The following is immediate because a deterministic Turing machine is a special case of a nondeterministic one.

LEMMA $P \subseteq NP$

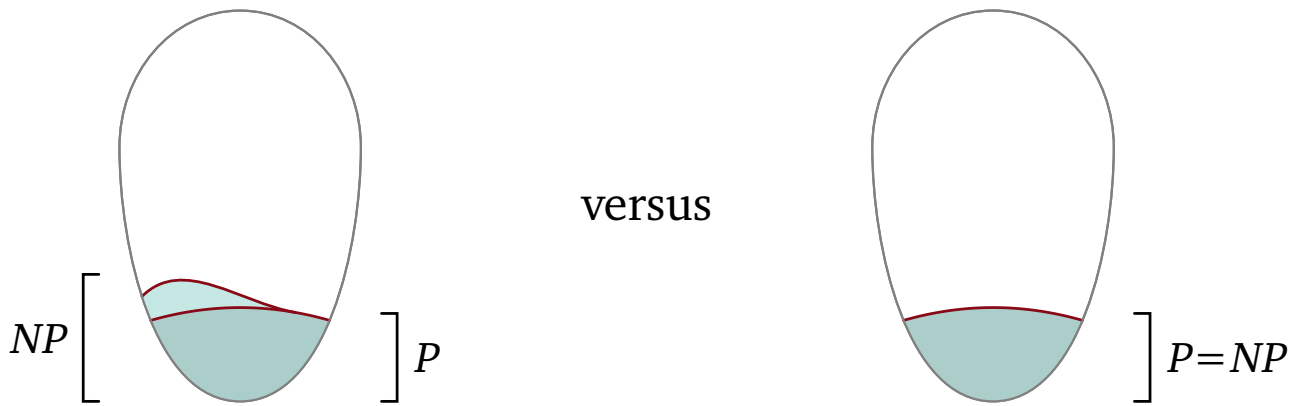
NP

DEFINITION The complexity class *NP* is the set of languages for which there is a nondeterministic Turing machine decider $\mathcal{P}$ that runs in polytime, meaning that there is a polynomial p such that on input σ, all branches of $\mathcal{P}$'s computation halt in time $p(|\sigma|)$.

The following is immediate because a deterministic Turing machine is a special case of a nondeterministic one.

LEMMA $P \subseteq NP$

We will have much more to say about this later but in short, we don't know whether the two sets are equal. On the left below *NP* is shown as a strict superset of *P* while on the right the two are shown as equal. (In all of these bean diagrams, problems with faster solution algorithms are nearer the bottom.)



Verifiers

Mathematical presentations often introduce an idea with a definition that is natural or conceptually clear and then follow it with a result giving a characterization that is more convenient for working on problems. The next lemma is an example; to show that a problem is in *NP* we almost always apply it, not the definition.

DEFINITION A verifier for a language $\mathcal{L}$ is a deterministic Turing machine $\mathcal{V}$ that inputs $\langle \sigma, \omega \rangle \in \mathbb{B}^2$ and is such that $\sigma \in \mathcal{L}$ if and only if there exists an ω so that $\mathcal{V}$ accepts $\langle \sigma, \omega \rangle$. The string ω is called the witness or certificate.

LEMMA A language is in *NP* if and only if it has a verifier that runs in time polynomial in $|\sigma|$. That is, $\mathcal{L} \in NP$ if and only if there is a polynomial p and a deterministic Turing machine $\mathcal{V}$ that halts on all inputs $\langle \sigma, \omega \rangle$ in $p(|\sigma|)$ time, and is such that $\sigma \in \mathcal{L}$ exactly when there is a witness ω where $\mathcal{V}$ accepts $\langle \sigma, \omega \rangle$.

Verifiers

Mathematical presentations often introduce an idea with a definition that is natural or conceptually clear and then follow it with a result giving a characterization that is more convenient for working on problems. The next lemma is an example; to show that a problem is in *NP* we almost always apply it, not the definition.

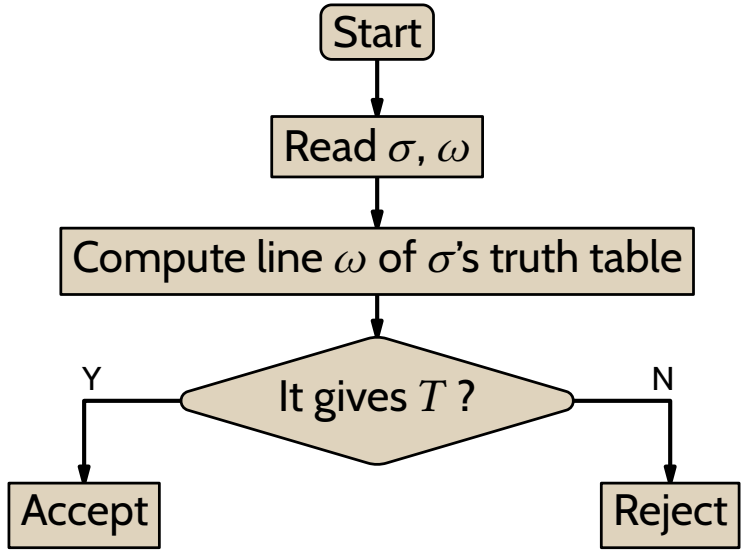
DEFINITION A verifier for a language $\mathcal{L}$ is a deterministic Turing machine $\mathcal{V}$ that inputs $\langle\sigma,\omega\rangle \in \mathbb{B}^2$ and is such that $\sigma \in \mathcal{L}$ if and only if there exists an ω so that $\mathcal{V}$ accepts $\langle\sigma,\omega\rangle$. The string ω is called the witness or certificate.

LEMMA A language is in *NP* if and only if it has a verifier that runs in time polynomial in $|\sigma|$. That is, $\mathcal{L} \in NP$ if and only if there is a polynomial p and a deterministic Turing machine $\mathcal{V}$ that halts on all inputs $\langle\sigma,\omega\rangle$ in $p(|\sigma|)$ time, and is such that $\sigma \in \mathcal{L}$ exactly when there is a witness ω where $\mathcal{V}$ accepts $\langle\sigma,\omega\rangle$.

(The proof will follow some examples.) The definition says 'there exists' a witness ω. This formalizes the mental model of the machine guessing, or of a demon providing the answer. Consider the satisfiability example above. Imagine the demon whispering, "Psst! Look at the ω = TTF line." We can then deterministically verify that hint, quickly.

Example: Satisfiability is in *NP*

Our touchstone is the Satisfiability problem. Using the lemma to show that this problem is in *NP* requires that we produce a deterministic Turing machine verifier. In the flowchart below the first input σ is a Boolean expression while the second, the ω, is a string that $\mathcal{V}$ interprets as describing a line of σ's truth table.



If a candidate expression σ is satisfiable then there is a suitable witness, a line from the truth table, so that $\mathcal{V}$ can check that the named line gives a result of T. As an example, for the expression $(*)$ from above, take $\omega = \texttt{TTF}$. Clearly the verifier can do the checking in polytime. On the other hand, if a candidate σ is not satisfiable, for example with the expression $\sigma = P \wedge \neg P$, then no ω will cause $\mathcal{V}$ to accept.

Comments on the verifier definition and lemma

- The most striking thing about the definition is that it says that 'there exists' a witness ω but it does not say where the witness comes from. A person with a computational mindset may well ask, "but how will we calculate the ω's?" The point is not how to find them. The point is whether there exists a deterministic Turing machine $\mathcal{V}$ that can leverage a given hint ω to verify in polytime that $\sigma \in \mathcal{L}$. That is, we don't find the ω's, we just use them.

Comments on the verifier definition and lemma

- The most striking thing about the definition is that it says that 'there exists' a witness ω but it does not say where the witness comes from. A person with a computational mindset may well ask, "but how will we calculate the ω's?" The point is not how to find them. The point is whether there exists a deterministic Turing machine $\mathcal{V}$ that can leverage a given hint ω to verify in polytime that $\sigma \in \mathcal{L}$. That is, we don't find the ω's, we just use them.
- Second, if $\sigma \notin \mathcal{L}$ then the definition does not require a witness to that. Instead, what's required is that from among all possible strings ω there is none such that the verifier accepts $\langle \sigma, \omega \rangle$.

- The third comment relates to this asymmetry. Imagine that a demon hands you some papers and claims that they contain an unbeatable strategy for chess. Verifying requires stepping through the responses to each move, and responses to the responses, etc., so there is lots of branching. To prove that the strategy is unbeatable we appear to have to check all of the branches, not just find one good one. It seems that a deterministic verifier must take exponential time. That would make the demon's papers, in a sense, useless. So this chess strategy is not like the problems that we have been considering.

  Thus, the lemma explains something about the class *NP*: while *P* contains problems where we can find the answer in polytime, *NP* contains the problems whose answers are useful in that we can at least verify them in polytime.

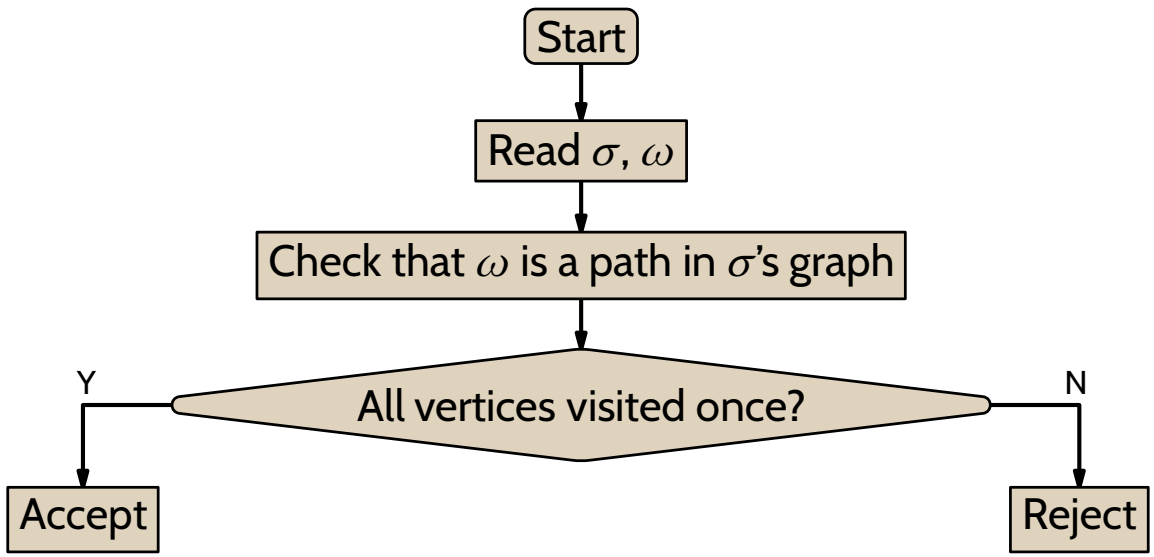
  Also reflecting this asymmetry, it is not clear that $\mathcal{L} \in NP$ implies that its complement $\mathcal{L}^{\mathsf{c}}$ is a member of *NP*. Consider Satsifiability. If a propositional logic expression σ is satisfiable then a witness to that is a pointer to a line of the truth table. But for non-satisfiability there is no natural witness; instead, the natural thing is to check all lines. As far as we know today, verifying that a Boolean formula is not satisfiable takes more than polytime. Consequently, where the complexity class *co-NP* contains the complements of languages from *NP*, we suspect that $NP \neq co\text{-}NP$.

- Finally, the lemma requires that the runtime of the verifier is polynomial in $|\sigma|$, not polynomial in the length of its input, $\langle \sigma, \omega \rangle$. If it said the latter then we could check the chess strategy just by using a witness that is exponentially long, which consequently makes $\langle \sigma, \omega \rangle$ exponentially long. Also observe that because $\mathcal{V}$ runs in time polynomial in $|\sigma|$, for the verifier to accept there must exist a witness whose length is at most polynomial in $|\sigma|$, because with ω's that are too long the verifier cannot even input them before its runtime bound expires.

EXAMPLE The Hamiltonian Path problem is like the Hamiltonian Circuit problem except that instead of requiring that the starting vertex equals the ending one, it inputs two vertices. It is the problem of determining membership in this set.

$$\mathcal{L} = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{path in } \mathcal{G} \text{ between } v \text{ and } \hat{v} \text{ visits every vertex exactly once} \}$$

We will show that this problem is in the class *NP*. We must produce a deterministic Turing machine verifier $\mathcal{V}$. It is sketched below. It takes as input $\langle \sigma, \omega \rangle$, where the candidate for membership in $\mathcal{L}$ is $\sigma = \langle \mathcal{G}, v, \hat{v} \rangle$. The verifier interprets the witness to be a path, $\omega = \langle v, v_1, \ldots \hat{v} \rangle$.



If there is a Hamiltonian path then there exists a witness ω, and so there is input that $\mathcal{V}$ will accept. Clearly, if given acceptable input then $\mathcal{V}$ runs in polytime. On the other hand, if σ has no Hamiltonian path then for no ω will $\mathcal{V}$ be able to verify that ω is such a path, and thus it will not accept any input pair starting with σ.

EXAMPLE Vertex-to-Vertex Path takes as input a graph and two vertices and the triple is in the language if there is a path between the vertices.

$$\mathcal{L} = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{there is a path in } \mathcal{G} \text{ from } v \text{ to } \hat{v} \}$$

This problem is in *NP*. For the witness ω, use the path. We can write a verifier $\mathcal{V}$ that accepts a triple $\sigma = \langle \mathcal{G}, v, \hat{v} \rangle$ along with a witness path ω, and tries to verify that the path lies in the graph between the two vertices. If there is such a path then there is a pair $\langle \sigma, \omega \rangle$ that $\mathcal{V}$ can accept. If there is no such path then there is no such pair. Clearly verification is fast, polytime.

EXAMPLE Vertex-to-Vertex Path takes as input a graph and two vertices and the triple is in the language if there is a path between the vertices.

$$\mathcal{L} = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{there is a path in } \mathcal{G} \text{ from } v \text{ to } \hat{v} \}$$

This problem is in *NP*. For the witness ω, use the path. We can write a verifier $\mathcal{V}$ that accepts a triple $\sigma = \langle \mathcal{G}, v, \hat{v} \rangle$ along with a witness path ω, and tries to verify that the path lies in the graph between the two vertices. If there is such a path then there is a pair $\langle \sigma, \omega \rangle$ that $\mathcal{V}$ can accept. If there is no such path then there is no such pair. Clearly verification is fast, polytime.

EXAMPLE Linear programming optimizes some linear function $f(x_0, \ldots x_{n-1}) = a_0 x_0 + \cdots + a_{n-1} x_{n-1}$, called the objective function, subject to a list C of constraints of the form $b_{i,0} x_0 + \cdots + b_{i,n-1} x_{n-1} \leq d_i$ and $x_i \geq 0$. To make this a language decision problem, recast it using a parameter lower bound B.

$$\mathcal{L}_B = \{ \langle f, C \rangle \mid \text{some } \langle x_0, \ldots x_{n-1} \rangle \text{ satisfies the constraints } C \text{ and } f(x_0, \ldots x_{n-1}) \geq B \}$$

We can write a verifier $\mathcal{V}$ that takes in a pair $\langle \sigma, \omega \rangle$, where σ is a pair $\langle f, C \rangle$ and ω is a tuple, $\langle x_0, \ldots x_{n-1} \rangle$, and where the verifier checks whether the tuple satisfies the constraints and also makes the objective function larger than B. If there is such a sequence then $\mathcal{V}$ can verify it, in polytime. If there is no such sequence then $\mathcal{V}$ can never accept its input.

EXAMPLE Vertex-to-Vertex Path takes as input a graph and two vertices and the triple is in the language if there is a path between the vertices.

$$\mathcal{L} = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{there is a path in } \mathcal{G} \text{ from } v \text{ to } \hat{v} \}$$

This problem is in *NP*. For the witness ω, use the path. We can write a verifier $\mathcal{V}$ that accepts a triple $\sigma = \langle \mathcal{G}, v, \hat{v} \rangle$ along with a witness path ω, and tries to verify that the path lies in the graph between the two vertices. If there is such a path then there is a pair $\langle \sigma, \omega \rangle$ that $\mathcal{V}$ can accept. If there is no such path then there is no such pair. Clearly verification is fast, polytime.

EXAMPLE Linear programming optimizes some linear function $f(x_0, \ldots x_{n-1}) = a_0 x_0 + \cdots + a_{n-1} x_{n-1}$, called the objective function, subject to a list C of constraints of the form $b_{i,0} x_0 + \cdots + b_{i,n-1} x_{n-1} \leq d_i$ and $x_i \geq 0$. To make this a language decision problem, recast it using a parameter lower bound B.

$$\mathcal{L}_B = \{ \langle f, C \rangle \mid \text{some } \langle x_0, \ldots x_{n-1} \rangle \text{ satisfies the constraints } C \text{ and } f(x_0, \ldots x_{n-1}) \geq B \}$$

We can write a verifier $\mathcal{V}$ that takes in a pair $\langle \sigma, \omega \rangle$, where σ is a pair $\langle f, C \rangle$ and ω is a tuple, $\langle x_0, \ldots x_{n-1} \rangle$, and where the verifier checks whether the tuple satisfies the constraints and also makes the objective function larger than B. If there is such a sequence then $\mathcal{V}$ can verify it, in polytime. If there is no such sequence then $\mathcal{V}$ can never accept its input.

EXAMPLE 3-Coloring takes a graph as input and decides if it can be colored with at most 3 colors. For ω use a purported 3-coloring. The machine $\mathcal{V}$ can verify that ω actually is a 3-coloring in time polynomial in the size of the graph.

Proof of the lemma

Here is the statement of the lemma again.

LEMMA A language is in NP if and only if it has a verifier that runs in time polynomial in $|\sigma|$. That is, $\mathcal{L} \in NP$ if and only if there is a polynomial p and a deterministic Turing machine $\mathcal{V}$ that halts on all inputs $\langle \sigma, \omega \rangle$ in $p(|\sigma|)$ time, and is such that $\sigma \in \mathcal{L}$ exactly when there is a witness ω where $\mathcal{V}$ accepts $\langle \sigma, \omega \rangle$.

PF Suppose first that the language $\mathcal{L}$ is accepted by a nondeterministic Turing machine $\mathcal{P}$ in polytime. We will construct a deterministic verifier $\mathcal{V}$ that runs in polytime. Let $p : \mathbb{N} \to \mathbb{N}$ be the polynomial such that for any input $\sigma \in \mathcal{L}$, the machine $\mathcal{P}$ has an accepting branch of length at most $p(|\sigma|)$. Use that branch to make a witness to σ's acceptance, for instance as in the sequence $\omega = \langle 3, 2 \ldots \rangle$ meaning, "At the first node take the third child, then take the second child of that, etc." Restated, $\mathcal{P}$ has a finite number of states k so we can represent the accepting branch of its computation tree with a sequence ω of at most $p(|\sigma|)$ many numbers, each less than k. In total, ω's length is polynomial in $|\sigma|$. With this ω, a deterministic machine $\mathcal{V}$ can verify $\mathcal{P}$'s acceptance of σ, in polynomial time.

For the converse suppose that the language $\mathcal{L}$ is accepted by a verifier $\hat{\mathcal{V}}$ that runs in time bounded by a polynomial q. We will construct a nondeterministic Turing machine $\hat{\mathcal{P}}$ that in polytime accepts an input bitstring τ if and only if $\tau \in \mathcal{L}$.

The key is that $\hat{\mathcal{P}}$ is nondeterministic. Given a candidate τ for membership in the language, (1) have $\hat{\mathcal{P}}$ nondeterminstically produce a witness $\hat{\omega}$ of length less than $q(|\tau|)$, (2) have $\hat{\mathcal{P}}$ then run $\langle \tau, \hat{\omega} \rangle$ through $\hat{\mathcal{V}}$, and (3) if the verifier accepts its input then $\hat{\mathcal{P}}$ accepts τ, while if $\mathcal{V}$ does not accept then $\hat{\mathcal{P}}$ rejects τ.

We must check that $\hat{\mathcal{P}}$ accepts τ if and only if $\tau \in \mathcal{L}$. A nondeterministic machine accepts a string if there exists a branch that accepts the string, and rejects the string if every branch rejects it. Suppose first that $\tau \in \mathcal{L}$. Because $\hat{\mathcal{V}}$ is a verifier, in this case there exists a witness $\hat{\omega}$ (of length less than $q(|\tau|)$) that will result in $\hat{\mathcal{V}}$ accepting $\langle \tau, \hat{\omega} \rangle$, so there is a way for the prior paragraph to result in acceptance of τ, and so $\hat{\mathcal{P}}$ accepts τ. Conversely, suppose that $\tau \notin \mathcal{L}$. By the definition of a verifier, no witness $\hat{\omega}$ will result in $\hat{\mathcal{V}}$ accepting $\langle \tau, \hat{\omega} \rangle$, and thus $\hat{\mathcal{P}}$ rejects τ. ∎

Problem reduction

Introduction

When we studied incomputability we considered a sense in which some problems are harder than others. Recall the Halts on Three problem to decide membership in the set $S = \{ e \in \mathbb{N} \mid \phi_e(3)\downarrow \}$. We showed that if we could solve this then we could solve the Halting problem and we denoted this situation with $K \leq_T S$. In general, a set B Turing reduces to A, written $B \leq_T A$, if there is a Turing machine that computes B from an A oracle, so that $\phi_e^A = \mathbb{1}_B$. Said another way, we can answer questions about membership in B by being given access to a routine that answers questions about membership in A.

Introduction

When we studied incomputability we considered a sense in which some problems are harder than others. Recall the Halts on Three problem to decide membership in the set $S = \{e \in \mathbb{N} \mid \phi_e(3)\downarrow\}$. We showed that if we could solve this then we could solve the Halting problem and we denoted this situation with $K \leq_T S$. In general, a set B Turing reduces to A, written $B \leq_T A$, if there is a Turing machine that computes B from an A oracle, so that $\phi_e^A = \mathbb{1}_B$. Said another way, we can answer questions about membership in B by being given access to a routine that answers questions about membership in A.

This is terminology that is used generally in Mathematics. We say that a problem B 'reduces to' A when in order to solve B, it suffices to solve A. Here are two examples.

Problem B	*Problem A*
Find the maximum of a list of numbers	Sort a list of numbers
Find the roots of a polynomial	Factor a polynomial into linear terms

The intuition is that A is at least as hard as B.

Reduction between language decision problems

Recall Vertex-to-Vertex Path: given a graph and two vertices, decide if there is a path between them. Recall also Shortest Path: given a weighted graph and two vertices, find the shortest path between them.

These two are related in that if you can find the shortest path then you can find whether there is a path at all.

Reduction between language decision problems

Recall Vertex-to-Vertex Path: given a graph and two vertices, decide if there is a path between them. Recall also Shortest Path: given a weighted graph and two vertices, find the shortest path between them.

These two are related in that if you can find the shortest path then you can find whether there is a path at all.

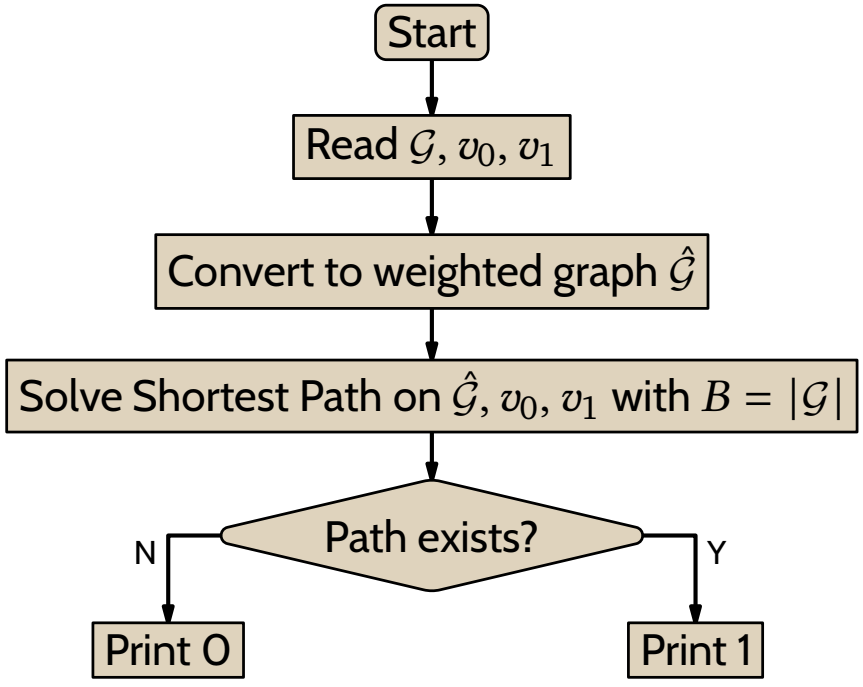
As a run-up toward the definition that makes this relationship precise, we first state these as language decision problems.

$$\mathcal{L} = \{\, \langle \mathcal{G}, v_0, v_1 \rangle \;\big|\; \text{there is a path from } v_0 \text{ to } v_1 \text{ in } \mathcal{G} \,\}$$

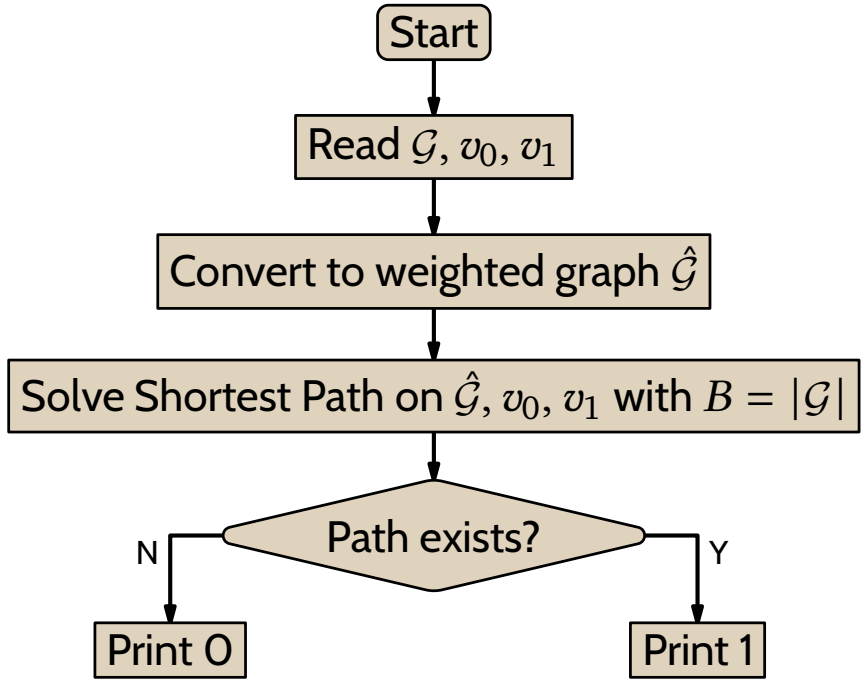
$$\mathcal{L}_B = \{\, \langle \hat{\mathcal{G}}, w_0, w_1 \rangle \;\big|\; \text{there is a path from } w_0 \text{ to } w_1 \text{ shorter than length } B \,\}$$

(With the second, we are applying the standard bound technique, using the parameter $B \in \mathbb{N}$. Also, there is a type issue because Shortest Path applies to weighted graphs. But we can patch up the type issue by taking an unweighted graph to be a weighted one where each edge has weight 1.)

Suppose that you could solve Shortest Path. Perhaps you have a library routine `SolveShortestPath`. We say that you have a 'Shortest Path oracle'. This sketches an algorithm leveraging that oracle to give an algorithm that solves Vertex-to-Vertex Path.



Suppose that you could solve Shortest Path. Perhaps you have a library routine `SolveShortestPath`. We say that you have a 'Shortest Path oracle'. This sketches an algorithm leveraging that oracle to give an algorithm that solves Vertex-to-Vertex Path.



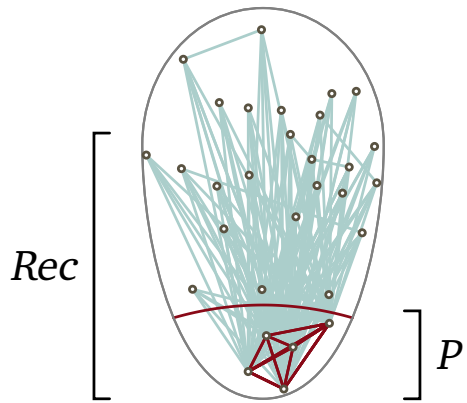
This computes a function which translates decisions about instances of Vertex-to-Vertex Path into decisions about instances of Shortest Path.

In particular, because the algorithm runs in polytime, if Shortest Path is in P then Vertex-to-Vertex Path is also in P.

Polytime reduction

DEFINITION Let $\mathcal{L}_0, \mathcal{L}_1$ be languages, subsets of $\mathbb{B}^*$. Then $\mathcal{L}_1$ is polynomial time reducible to $\mathcal{L}_0$, or Karp reducible, or polynomial time mapping reducible, or polynomial time many-one reducible, written $\mathcal{L}_1 \leq_p \mathcal{L}_0$ or $\mathcal{L}_1 \leq_m^p \mathcal{L}_0$, if there is a reduction function or transformation function $f : \mathbb{B}^* \to \mathbb{B}^*$ that is polynomial time computable and such that $\sigma \in \mathcal{L}_1$ if and only if $f(\sigma) \in \mathcal{L}_0$.

This is a sketch of polytime reduction among problems. The bean encloses all problems, which we take to be languages, subsets of $\mathbb{B}^*$ (we only show a few problems for graphical clarity). The easy ones, the ones with fast solution algorithm, are at the bottom. Problems are shown connected if there is a polynomial time reduction from one to the other. Highlighted are connections within the complexity class P.



Satisfiability $\leq_p$ Clique

The Clique problem is the decision problem for the language $\mathcal{L}_B = \{ \langle \mathcal{G}, B \rangle \mid \mathcal{G}$ has a clique with B vertices$\}$. We will sketch that Satisfiability $\leq_p$ Clique, so that intuitively Clique is at least as hard as Satisfiability.

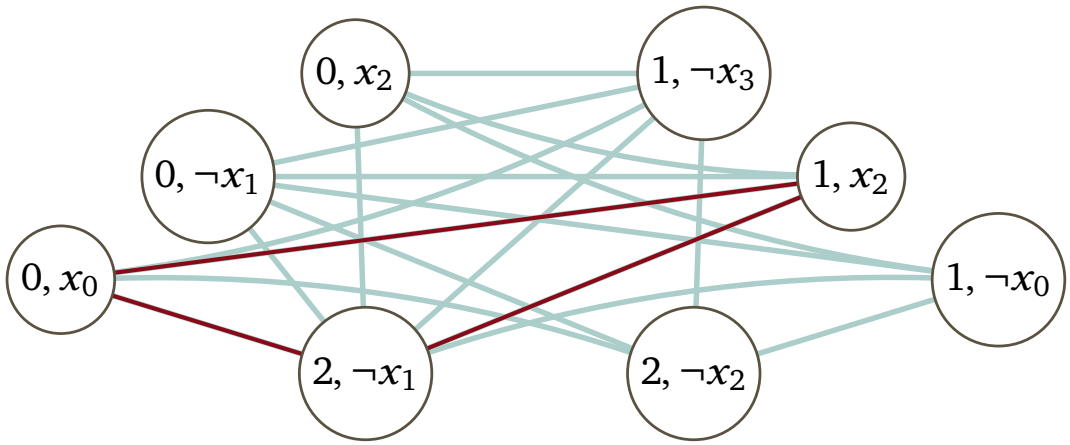
Consider how to satisfy this Boolean expression.

$$E = (x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$$

The $\wedge$'s make the statement as a whole T if and only if all of its clauses are T. The $\vee$'s mean that each clause is T if and only if any of its literals is T. So to satisfy the expression, select a literal from each clause and assign it the value T. For example, we can make E be T by selecting x_0 from the first clause, x_2 from the second, and $\neg x_1$ from the third, and making them T. Similarly, if $\neg x_1$ from the first and third clauses and x_3 from the second are T then E is T. What we cannot do is pick x_2 from the first and second and then $\neg x_2$ from the third, because we cannot set both of these literals to be T.

That is, we can think of Satisfiability as a combinatorial problem. The clauses are like buckets and we select one thing from each bucket, subject to the constraint that the things we select must be pairwise compatible.

This view of Satisfiability has a binary relation 'can be compatibly picked' between the literals. So, as below, let $\mathcal{G}_E$ be a graph whose vertices are pairs $\langle c, \ell \rangle$ where c is the number of a clause and ℓ is a literal in that clause. Two vertices $v_0 = \langle c_0, \ell_0 \rangle$ and $v_1 = \langle c_1, \ell_1 \rangle$ are connected by an edge if they come from different clauses so $c_0 \neq c_1$, and if the literals are not negations of each other so $\ell_0 \neq \neg\ell_1$.

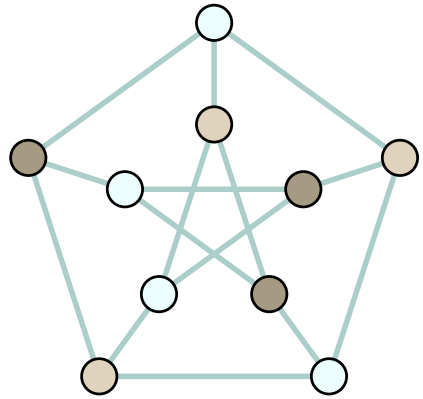


A choice of three mutually compatible vertices makes E evaluate to T. That is, the 3 clause expression E is satisfiable if and only if $\mathcal{G}_E$ has a 3-clique.

More formally, the reduction function f inputs a propositional logic expression E and outputs a pair $f(E) = \langle \mathcal{G}_E, B \rangle$ where $\mathcal{G}_E$ is the compatibility graph associated with E defined in the prior paragraph and where B is the number of clauses in E. Then $E \in$ SAT if and only if $f(E) \in \mathcal{L}_B$. Clearly this function can be computed in polytime.

Graph Colorability $\leq_p$ Satisfiability

Recall that a graph is k-colorable if we can partition the vertices into k many classes, called 'colors', so that two vertices can have the same color only when there is no edge between them. This is a 3-coloring of the graph.



We will illustrate that the Graph Colorability problem reduces to the Satisfiability problem, Graph Colorability $\leq_p$ Satisfiability, by focusing on the $k = 3$ construction. (Larger k's work much the same way, although the $k = 2$ case is different.)

To demonstrate that Graph Colorability $\leq_p$ Satisfiability, we must show that given an instance of Graph Colorability, we can translate it into an instance of Satisfiability (where the translation algorithm takes polytime).

Denote the set of satisfiable propositional logic statements as $\mathcal{L}_0$ and the set of 3-colorable graphs as $\mathcal{L}_1$. To show that $\mathcal{L}_1 \leq_p \mathcal{L}_0$ we must produce a reduction function f. It inputs a graph $\mathcal{G}$ and a outputs a propositional logic expression $E = f(\mathcal{G})$ such that the graph is 3-colorable if and only if the expression is satisfiable.

The function builds E by including clauses that state, in the language of propositional logic, the constraints to be met for the graph to be 3-colorable. Let $\mathcal{G}$ have n-many vertices $\{v_0, \ldots v_{n-1}\}$. Then E has $3n$-many Boolean variables, $a_0, \ldots a_{n-1}$, and $b_0, \ldots b_{n-1}$, and $c_0, \ldots c_{n-1}$. The idea is that if the i-th vertex v_i gets the first color then E will be satisfied when the associated variables have $a_i = T, b_i = F, c_i = F$, while if v_i gets the second color then E will be satisfied when $a_i = F, b_i = T, c_i = F$, and if v_i gets the third color then E will be satisfied when $a_i = F, b_i = F, c_i = T$.

Specifically, the expression includes two kinds of clauses. For every vertex v_i, there is a clause saying that the vertex gets at least one color.

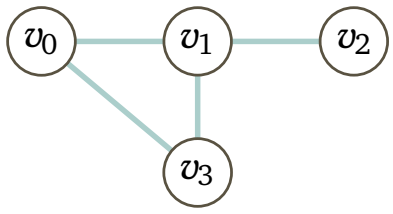
$$(a_i \vee b_i \vee c_i)$$

And for each edge $\{v_i, v_j\}$, there are three clauses which together ensure that the edge does not connect two same-color vertices.

$$(\neg a_i \vee \neg a_j) \qquad (\neg b_i \vee \neg b_j) \qquad (\neg c_i \vee \neg c_j)$$

The function's output E is the conjunction of all of these clauses.

This illustrates. The expression's top line has the clauses of the first kind while the remaining lines have the other kind.



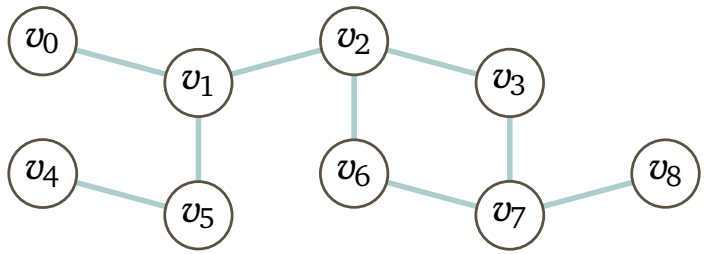
$$(a_0 \vee b_0 \vee c_0) \wedge (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge (a_3 \vee b_3 \vee c_3)$$
$$\wedge (\neg a_0 \vee \neg a_1) \wedge (\neg b_0 \vee \neg b_1) \wedge (\neg c_0 \vee \neg c_1)$$
$$\wedge (\neg a_0 \vee \neg a_3) \wedge (\neg b_0 \vee \neg b_3) \wedge (\neg c_0 \vee \neg c_3)$$
$$\wedge (\neg a_1 \vee \neg a_2) \wedge (\neg b_1 \vee \neg b_2) \wedge (\neg c_1 \vee \neg c_2)$$
$$\wedge (\neg a_1 \vee \neg a_3) \wedge (\neg b_1 \vee \neg b_3) \wedge (\neg c_1 \vee \neg c_3)$$

By construction, there is an assignments of truth values for the variables to satisfy the expression if and only if the graph has a 3-coloring. Completing the argument requires checking that the reduction function, which inputs a bitstring representation of the graph and outputs a bitstring representation of the expression, is polynomial. That's clear so we omit the details.

Vertex Cover $\leq_p$ Set Cover

Vertex Cover inputs a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ and a number $B \in \mathbb{N}^+$. It asks if there are vertices $v_{i_0}, v_{i_1}, \ldots v_{i_{B-1}}$ so that each edge contains at least one such vertex.

EXAMPLE Your museum has nine corridors and you must have a guard in a corner of each corridor.

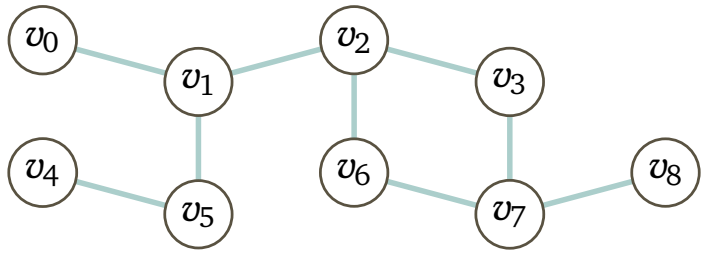


You can get away with $B = 4$ guards, posted at v_1, v_2, v_5, and v_7.

Vertex Cover $\leq_p$ Set Cover

Vertex Cover inputs a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ and a number $B \in \mathbb{N}^+$. It asks if there are vertices $v_{i_0}, v_{i_1}, \ldots\ v_{i_{B-1}}$ so that each edge contains at least one such vertex.

EXAMPLE Your museum has nine corridors and you must have a guard in a corner of each corridor.



You can get away with $B = 4$ guards, posted at v_1, v_2, v_5, and v_7.

Set Cover inputs a set U, a number $B \in \mathbb{N}^+$, and a collection $S_0, S_1, \ldots\ S_m$ of subsets of U. It asks if there is a subcollection whose union is the entire set $S_{i_0} \cup S_{i_1} \cup \cdots \cup S_{i_{B-1}} = U$.

EXAMPLE Imagine you are hiring people and have ten skills that you need to cover, $U = \{0, 1, \ldots\ 9\}$. The four applicants list these skills.

$$S_0 = \{1, 4\} \quad S_1 = \{2, 3, 4, 5\} \quad S_2 = \{0, 7, 9\} \quad S_3 = \{1, 6, 8\}$$

You can get away with hiring only $B = 3$ applicants because $S_1 \cup S_2 \cup S_3 = U$.

We will show that Vertex Cover $\leq_p$ Set Cover by showing how to translate instances of Vertex Cover into instances of Set Cover.

First write each as a language decision problem.

$$\mathcal{L}_0 = \{ \langle U, \{ S_0, \ldots S_m \}, B \rangle \mid \text{some } S_{i_0}, S_{i_1}, \ldots S_{i_{B-1}} \text{ covers } U \}$$

$$\mathcal{L}_1 = \{ \langle \mathcal{G}, B \rangle \mid \text{some set } v_{i_0}, v_{i_1}, \ldots v_{i_{B-1}} \text{ covers all edges} \}$$

We must produce a polytime computable function f that takes in $\sigma = \langle \mathcal{G}, B \rangle$ and outputs $f(\sigma) = \langle U, \{ S_0, \ldots S_m \}, B \rangle$, such that $\sigma \in \mathcal{L}_1$ if and only if $f(\sigma) \in \mathcal{L}_0$.

We will show that **Vertex Cover** $\leq_p$ **Set Cover** by showing how to translate instances of **Vertex Cover** into instances of **Set Cover**.

First write each as a language decision problem.

$$\mathcal{L}_0 = \{ \langle U, \{S_0, \ldots S_m\}, B\rangle \mid \text{some } S_{i_0}, S_{i_1}, \ldots S_{i_{B-1}} \text{ covers } U \}$$

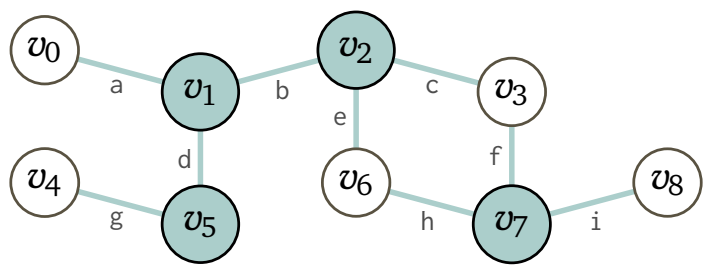
$$\mathcal{L}_1 = \{ \langle \mathcal{G}, B\rangle \mid \text{some set } v_{i_0}, v_{i_1}, \ldots v_{i_{B-1}} \text{ covers all edges} \}$$

We must produce a polytime computable function f that takes in $\sigma = \langle \mathcal{G}, B\rangle$ and outputs $f(\sigma) = \langle U, \{S_0, \ldots S_m\}, B\rangle$, such that $\sigma \in \mathcal{L}_1$ if and only if $f(\sigma) \in \mathcal{L}_0$.

Given $\sigma = \langle \mathcal{G}, B\rangle$, take U to be the set of $\mathcal{G}$'s edges. For each of $\mathcal{G}$'s vertices $v_i \in \mathcal{V}$, let $S_{v_i} = \{ e \in \mathcal{E} \mid v_i \text{ is a vertex in } e \}$. Define $f(\sigma) = \langle U, \{S_0, \ldots S_m\}, B\rangle$.

Clearly we can compute f in polytime. Further, for the bound B there is a collection of sets $S_{v_0} \cup S_{v_1} \cup \cdots S_{v_{B-1}}$ if and only if the collection $v_0, v_1, \ldots v_{B-1}$ is a vertex cover of $\mathcal{G}$.

EXAMPLE For the corridor guards, $\mathcal{G}$, the associated sets are $U = \{a, b, \ldots i\}$, along with $S_{v_1} = \{a, b, d\}$, and $S_{v_2} = \{b, c, e\}$, ... and $S_{v_7} = \{f, h, i\}$.



The set cover $\{S_{v_1}, S_{v_2}, S_{v_5}, S_{v_7}\}$ gives a vertex cover $\{v_1, v_2, v_5, v_7\}$.

Properties of polytime reduction

LEMMA Polytime reduction is reflexive: $\mathcal{L} \leq_p \mathcal{L}$ for all languages. It is also transitive: $\mathcal{L}_2 \leq_p \mathcal{L}_1$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ imply that $\mathcal{L}_2 \leq_p \mathcal{L}_0$. Every nontrivial computable language is *P* hard: for $\mathcal{L}_1 \in P$, every language $\mathcal{L}_0$ with $\mathcal{L}_0 \neq \emptyset$ and $\mathcal{L}_0 \neq \mathbb{N}$ satisfies that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. The class *P* is closed downward: if $\mathcal{L}_0 \in P$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ then $\mathcal{L}_1 \in P$. So also is the class *NP*.

PF The first two sentences and the final sentence are Exercise 6.35.

For the third sentence fix a $\mathcal{L}_0$ that is nontrivial, so there is a member $\sigma \in \mathcal{L}_0$ and a nonmember $\tau \notin \mathcal{L}_0$. Let $\mathcal{L}_1$ be an element of *P*. We will specify a polytime reduction function f for $\mathcal{L}_1 \leq_p \mathcal{L}_0$. For $\alpha \in \mathbb{B}^*$, computing whether $\alpha \in \mathcal{L}_1$ can be done in polytime. If it is a member then let $f(\alpha) = \sigma$ while if not then let $f(\alpha) = \tau$.

For the downward closure of *P*, suppose that $\mathcal{L}_1 \leq_p \mathcal{L}_0$ via the polytime function g and also suppose that there is a polytime algorithm for determining membership in $\mathcal{L}_0$. Determine membership in $\mathcal{L}_1$ by starting with an input σ, finding $g(\sigma)$, and applying the $\mathcal{L}_0$ algorithm to settle whether $g(\sigma) \in \mathcal{L}_0$. Where the $\mathcal{L}_0$ algorithm runs in time that is $\mathcal{O}(n^i)$ and where g runs in time that is $\mathcal{O}(n^j)$, determining $\mathcal{L}_1$ membership in this way runs in time that is $\mathcal{O}(n^{\max(i,j)})$, which is polynomial. ∎

NP completeness

The Cook-Levin theorem

THEOREM (COOK-LEVIN THEOREM) The Satisfiability problem is in NP and has the property any problem in NP reduces to it: $\mathcal{L} \leq_p \mathsf{SAT}$ for any $\mathcal{L} \in NP$.

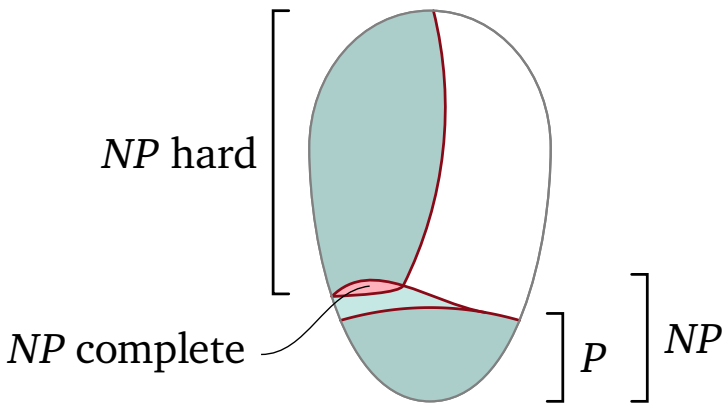
First, we have already observed that $\mathsf{SAT} \in NP$ because, given a Boolean expression, we can use as a witness ω an assignment of truth values that satisfies the expression.

Here is an outline of the proof's other half. Given $\mathcal{L} \in NP$, we must show that $\mathcal{L} \leq_p \mathsf{SAT}$. We produce a function $f_{\mathcal{L}}$ that translates membership questions for $\mathcal{L}$ into Boolean expressions, such that the membership answer is 'yes' if and only if the expression is satisfiable. What we know about $\mathcal{L}$ is that its member σ's are accepted by a nondeterministic machine $\mathcal{P}$ in time given by a polynomial q. With that, from $\langle \mathcal{P}, \sigma, q \rangle$ the proof constructs a Boolean expression that yields T if and only if $\mathcal{P}$ accepts σ. The Boolean expression encodes the constraints under which a Turing machine operates, such as that the only tape symbol that can be changed in the current step is the symbol under the machine's head.

NP complete, *NP* hard

DEFINITION A problem is *NP* hard if every problem in *NP* reduces to it, that is, $\mathcal{L}$ is *NP* hard if $\hat{\mathcal{L}} \in NP$ implies that $\hat{\mathcal{L}} \leq_p \mathcal{L}$. A problem is *NP* complete if, in addition to being *NP* hard, it is also a member of *NP*.

This shows all problems, the collection of all $\mathcal{L} \in \mathbb{B}^*$. In the bottom is *NP*, drawn with *P* as a proper subset (although we don't know this is true, most experts believe that it is). In the top are the *NP*-hard problems. The highlighted intersection of the two is the set of *NP* complete problems.



LEMMA If $\mathcal{L}_0$ is *NP* complete, and $\mathcal{L}_0 \leq_p \mathcal{L}_1$, and $\mathcal{L}_1 \in NP$ then $\mathcal{L}_1$ is *NP* complete.

Garey & Johnson's key *NP* complete problems

THEOREM (BASIC *NP* COMPLETE PROBLEMS) Each of these problems is *NP* complete.

3-Satisfiability, 3-SAT Given a propositional logic formula in conjunctive normal form in which each clause has at most 3 variables, decide if it is satisfiable.

3 Dimensional Matching Given as input a set $M \subseteq X \times Y \times Z$, where the sets X, Y, Z all have the same number of elements, n, decide if there is a matching, a set $\hat{M} \subseteq M$ containing n elements such that no two of the triples in $\hat{M}$ agree on any of their coordinates.

Vertex cover Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a size B set of vertices C such that for any edge $v_i v_j$, at least one of its ends is a member of C.

Clique Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a set of B-many vertices where any two are connected.

Hamiltonian Circuit Given a graph, decide if it contains a cyclic path that includes each vertex.

Partition Given a finite multiset S of natural numbers, decide if there is a division of the set into the two parts $\hat{S}$ and $S - \hat{S}$ so the total of their elements is the same, $\sum_{s \in \hat{S}} s = \sum_{s \notin \hat{S}} s$.

Travelling Salesman is *NP*-complete

EXAMPLE We will show that the Traveling Salesman problem is *NP* complete. Recall that we have recast it as the decision problem for the language of pairs $\langle \mathcal{G}, B \rangle$, where B is a parameter bound, and that this problem is a member of *NP*. We will show that it is *NP* hard by proving that the Hamiltonian Circuit problem reduces to it, Hamiltonian Circuit $\leq_p$ Traveling Salesman.

We need a reduction function f. It must input an instance of Hamiltonian Circuit, a graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ whose edges are unweighted. Define f to return the instance of Traveling Salesman that uses $\mathcal{N}$ as cities, that takes the distances between cities to be $d(v_i, v_j) = 1$ if $v_i v_j \in \mathcal{E}$ and $d(v_i, v_j) = 2$ if $v_i v_j \notin \mathcal{E}$, and such that the bound is the number of vertices, $B = |\mathcal{N}|$.

This bound means that there will be a Traveling Salesman solution if and only if there is a Hamiltonian Circuit solution; namely, the salesman uses the edges that appear in the Hamiltonian circuit. All that remains is to argue that the reduction function runs in polytime. The number of edges in a graph is no more than twice the number of vertices so polytime in the input graph size is the same as polytime in the number of vertices. The reduction function's algorithm examines all pairs of vertices, which takes time that is quadratic in the number of vertices.

Knapsack is *NP*-complete

EXAMPLE The Knapsack problem starts with a multiset of objects $S = \{s_0, \ldots s_{k-1}\}$, each with a natural number weight $w(s_i)$ and a value $v(s_i)$, along with a weight bound B and value target T. We then look for a knapsack $K \subseteq S$ whose elements have total weight less than or equal to the bound and total value greater than or equal to the target.

First we check that this problem is in *NP*. As the witness we can use the k-bit string ω such that $\omega[i] = 1$ if s_i is in the knapsack K, and $\omega[i] = 0$ if it is not. A deterministic machine can verify this witness in polynomial time since it only has to total the weights and values of the elements of K.

To finish we must show that Knapsack is *NP* hard. It is sufficient to show that a special case is *NP* hard. Consider the Knapsack instance where $w(s_i) = v(s_i)$ for all $s_i \in S$, and where the two criteria each equal half of the weight total, $B = T = 0.5 \cdot \sum_{0 \leq i < k} w(i)$. This shows that any instance of the Partition problem, which is in the above basic list, can be expressed as a Knapsack instance, so Partition $\leq_p$ Knapsack, and consequently the latter is *NP* hard.

EXP

DEFINITION A language decision problem is an element of the complexity class *EXP* if there is an algorithm for solving it that runs in time $\mathcal{O}(b^{p(n)})$ for some constant base b and polynomial p.

A first, informal, take is that *EXP* contains nearly every problem with which we concern ourselves in practice — it contains most problems that we seriously hope ever to attack. In contrast with polytime, where a rough summary is that its problems all have an algorithm that can conceivably be used, for the hardest problems in *EXP*, even the best algorithms are just too slow.

EXP

Definition A language decision problem is an element of the complexity class *EXP* if there is an algorithm for solving it that runs in time $\mathcal{O}(b^{p(n)})$ for some constant base b and polynomial p.

A first, informal, take is that *EXP* contains nearly every problem with which we concern ourselves in practice — it contains most problems that we seriously hope ever to attack. In contrast with polytime, where a rough summary is that its problems all have an algorithm that can conceivably be used, for the hardest problems in *EXP*, even the best algorithms are just too slow.

Lemma $P \subseteq NP \subseteq EXP$

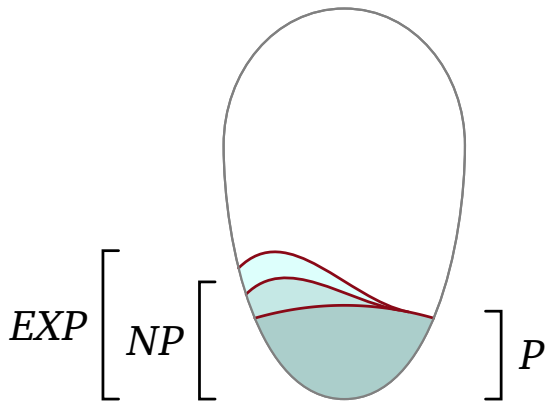
Pf. Fix $\mathcal{L} \in NP$. We can verify $\mathcal{L}$ on a deterministic Turing machine $\mathcal{P}$ in polynomial time using a witness whose length is bounded by the same polynomial. Let this problem's bound be n^c.

We will decide $\mathcal{L}$ in exponential time by brute-forcing it: we will use $\mathcal{P}$ to run every possible verification. Trying any single witness requires polynomial time, n^c. Witnesses are in binary so for length ℓ there are $\sum_{0 \leq i \leq \ell} 2^i = 2^{\ell+1} - 1$ many possible ones; In total then, brute force requires $\mathcal{O}(n^c 2^{n^c})$ operations. Finish by observing that $n^c 2^{n^c}$ is in $\mathcal{O}(2^{n^c})$. ∎

Dunno

We don't know whether there are any *NP* problems that absolutely require exponential time. Conceivably *NP* is contained in a smaller determinstic time complexity class — maybe Satisfiability, for instance, can be solved in less than exponential time. But we just don't know.

Here the blob encloses all problems. Shaded are the three classes *P*, *NP*, and *EXP*. They are drawn with strict containment, which most experts guess is the true arrangement, but no one knows for sure.

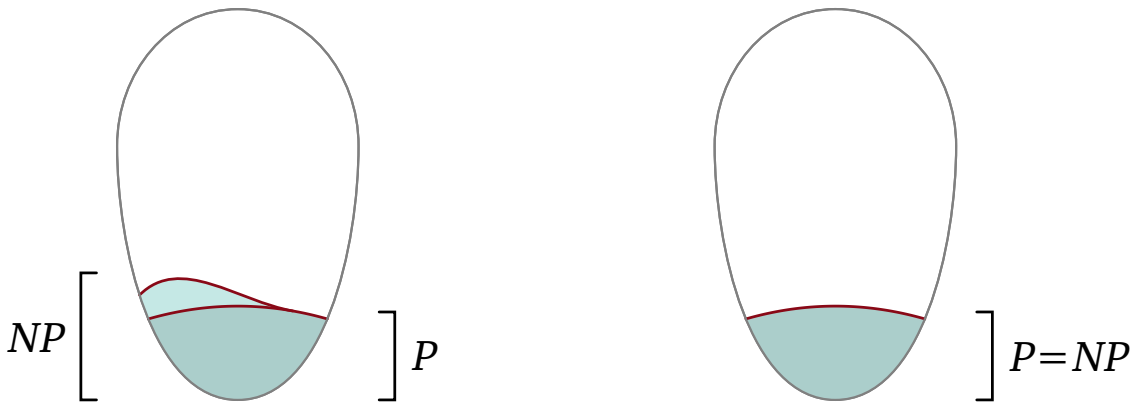


P versus NP

$?P = NP?$

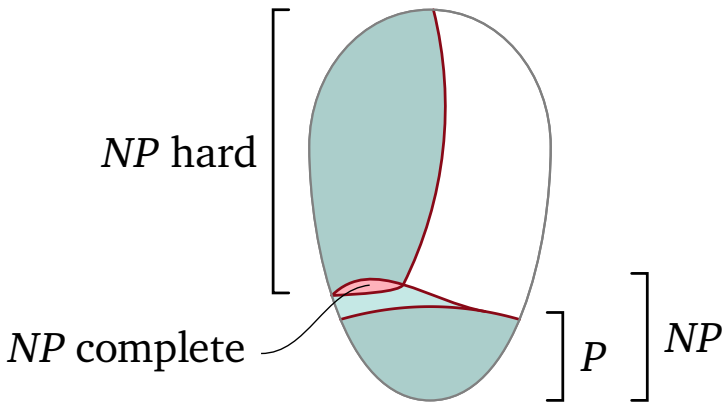
Every deterministic Turing machine is trivially a nondeterministic machine, and so $P \subseteq NP$. What about the other direction: can unbounded parallelism bring problems from super-polynomial to polynomial?

The short answer is that no one knows. One of these two pictures is correct but we do not know which one.



The place of *NP* complete problems

This bean holds all language decision problems, $\mathcal{L} \in \mathbb{B}^*$. In the bottom is *NP*, drawn with *P* as a strict subset (although we don't, strictly speaking, know that is true). In the top left are the *NP*-hard problems. The highlighted intersection of the two is the set of *NP* complete problems. These problems are elements of *NP* that are $\leq_p$-above all *NP* problems. They are the hardest of the *NP* problems. If there is a difference between *P* and *NP*, a gap containing the problems that are in $NP - P$, then they form a subset of it, right at its very top.



What's so interesting about *NP* complete problems?

Soon after Cook raised the question of *NP* completeness, R Karp brought it to widespread attention. Karp noted that there are clusters of problems: there is a collection of problems solvable in time $\mathcal{O}(\lg(n))$, problems of time $\mathcal{O}(n)$, those of time $\mathcal{O}(n \lg n)$, etc. There is also a cluster of problems that seem much tougher. He gave a list of twenty one of these, drawn from Computer Science, Mathematics, and the natural sciences, where lots of smart people had for years been unable find efficient algorithms. He showed that all of these problems are *NP* complete, so that if we could efficiently solve any then we could efficiently solve them all.

Karp demonstrated that many problems that people had been struggling with in practice — classic unsolved problems — fall in this category. Researchers who have been looking for an efficient solution to Vertex Cover and those who have been working on Clique find that they are working on the same problem, in that the two are inter-translatable. By now the list of *NP* complete problems includes determining the best layout of transistors on a chip, developing accurate financial-forecasting models, analyzing protein-folding behavior in a cell, or finding the most energy-efficient airplane wing. So the question of whether *P* equals *NP* is extremely practical, and extremely important.

If someone could solve just one of these problems in polytime then we could solve all of them in polytime.

In short: what to tell your boss

What if you have a problem that you cannot solve?

Researchers often take proving that a problem is *NP* complete to be an ending point; they may feel that continuing to look for an algorithm is a waste since many of the world's best minds have failed to find one. They may turn to finding approximations (see Extra B) or to probabilistic methods.

A researcher could imagine proving that the reason they cannot find an efficient algorithm is that no such algorithm exists. But perhaps more realistic is to show that your problem is *NP* complete and then argue that while you cannot find an efficient algorithm, neither can anyone else — that you are at the state of the art.

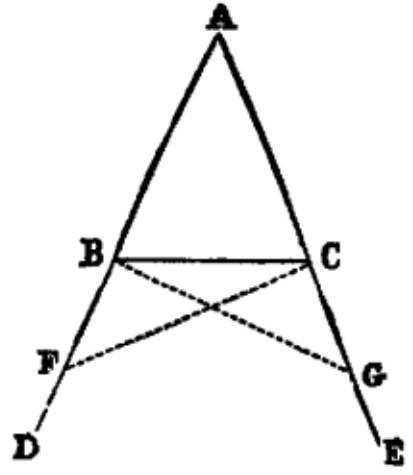
Status of the question

Certainly the P versus NP question is the sexiest one in the Theory of Computing today. It has attracted a great deal of gossip. In 2018, a poll of experts found that out of 152 respondents, 88% thought that $P \neq NP$ while only 12% thought that $P = NP$.

Reflection: why the question is central

In high school you had Euclid. This proves that the base angles of an isosceles triangle are equal.

Dem.—In BD take any point F, and from AE, the greater, cut off AG equal to AF [III]. Join BG, CF (Post. I.). Because AF is equal to AG (const.), and AC is equal to AB (hyp.), the two triangles FAC, GAB have the sides FA, AC in one respectively equal to the sides GA, AB in the other; and the included angle A is common to both triangles. Hence [IV.] the base FC is equal to GB, the angle AFC is equal to AGB, and the angle ACF is equal to the angle ABG.



Again, because AF is equal to AG (const.), and AB to AC (hyp.), the remainder, BF, is equal to CG (Axiom III); and we have proved that FC is equal to GB, and the angle BFC equal to the angle CGB. Hence the two triangles BFC, CGB have the two sides BF, FC in one equal to the two sides CG, GB in the other; and the angle BFC contained by the two sides of one equal to the angle CGB contained by the two sides of the other. Therefore [IV.] these triangles have the angle FBC equal to the angle GCB, *and these are the angles below the base.* Also the angle FCB equal to GBC; but the whole angle FCA has been proved equal to the whole angle GBA. Hence the remaining angle ACB is equal to the remaining angle ABC, *and these are the angles at the base.*

A proof is a sequence of statements, each justified by reference to some prior statement, to a law of logic, or to some axiom.

In a proof we require the all steps be typographic in the sense that we can for instance combine two prior statements using a law of logic to get a third statement, but a proof cannot take creative leaps. It must be mechanically checkable.

This is illustrated by Russell and Whitehead's *Principia Mathematica*, from the early 1900's, which tried to derive all of mathematics from first principles, in a way that is completely mechanically checkable. Each step describes the lines from which it is derived, so that for instance you can mechanically check that the second line follows from the first by applying the result 51.231.

This appears on p 362.

***54·43.** $\vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$\vdash . *54{\cdot}26 . \supset \vdash :. \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$

[*51·231] $\equiv . \iota`x \cap \iota`y = \Lambda .$

[*13·12] $\equiv . \alpha \cap \beta = \Lambda$ (1)

$\vdash . (1) . *11{\cdot}11{\cdot}35 . \supset$

$\vdash :. (\exists x, y) . \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda$ (2)

$\vdash . (2) . *11{\cdot}54 . *52{\cdot}1 . \supset \vdash .$ Prop

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

Recall the *Entscheidungsproblem* that was a motivation behind the definition of a Turing machine. It asks for an algorithm that inputs a mathematical statement and decides whether it is true. It is perhaps a caricature, but imagine that the job of mathematicians is to prove theorems. Then the *Entscheidungsproblem* asks if it is possible to replace mathematicians with mechanisms.

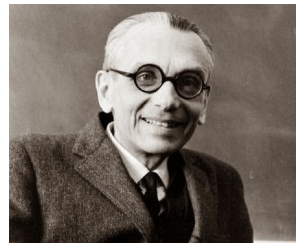
In the intervening century we have come to understand, through the work of Gödel and others, that there is a difference between a statement's being true and its being provable. Church and Turing expanded on this insight to show that the *Entscheidungsproblem* is unsolvable. Consequently, we change to asking for an algorithm that inputs statements and decides whether they are provable.

In principle this is simple. A proof is a sequence of statements, σ_0, σ_1, ... σ_k, where the final statement is the conclusion and where each statement either is an axiom or else follows from the statements before it by an application of a rule of deduction (a typical rule allows the simultaneous replacement of all x's with $y + 4$'s). A computer could brute-force the question of whether a given statement is provable by doing a dovetail, a breadth-first search of all derivations. If a proof exists then it will appear, eventually.

Restated: start with the axioms. Do a one step combination of them in all possible ways. Then do all two-step combinations, etc. This process generates all possible theorems, eventually.

The difficulty is the ‘eventually’. This algorithm is very slow. Is there a tractable way? In the terminology that we now have, the modified *Entscheidungsproblem* is a decision problem: given a statement σ and a bound, we ask if there is a sequence ω of statements witnessing a proof that ends in σ and that is shorter than the bound. A computer can quickly check whether a given proof is valid — this problem is in *NP*. With the current status of the *P* versus *NP* problem, the answer to the question in the prior paragraph is that no one knows of a fast algorithm but no one can show that there isn’t one either.


A Turing


K Gödel


J von Neumann

As far back as 1956, Gödel raised these issues in a letter to von Neumann (this letter did not become public until years later).

> *One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\Psi(F, n)$ be the number of steps the machine requires for this and let $\phi(n) = \max_F \Psi(F, n)$. The question is how fast $\phi(n)$ grows for an optimal machine. One can show that $\phi(n) \geq k \cdot n$. If there really were a machine with $\phi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\phi(n)$ grows that slowly. ... It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.*

This problem is in *NP*. If someone gives you a theorem's derivation ω then checking it is fast. But, that we know today, it appears that generating the theorems is slow. Certainly, the brute force, breadth-first, way is slow.

Gödel was extremely cautious. We can expand what he said into statements that are somewhat less careful but perhaps more memorable.

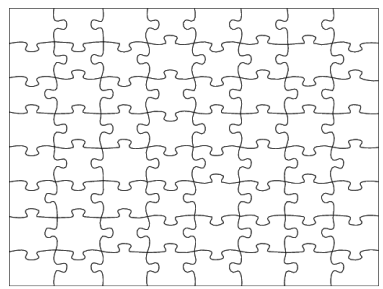
Gödel was extremely cautious. We can expand what he said into statements that are somewhat less careful but perhaps more memorable.

EXAMPLE Consider how the difficulty of an $n \times n$ Sudoku grid grows as n grows.

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

This problem is *NP* because as a witness ω we can use a filled-out grid, and check it quicky. But, we today know of no polytime solution.

EXAMPLE Jigsaw puzzles are similar.



This is *NP* because we can get as witness ω a finished puzzle. But like the SAT problem, researchers today do not know of any way to solve it that is essentially faster than trying all combinations.

S Cook:

> *. . . a practical algorithm for solving an NP-complete problem . . . would transform mathematics by allowing a computer to find a formal proof of any theorem that has a proof of reasonable length, since formal proofs can easily be recognized in polynomial time. . . . Although the formal proofs may not be initially intelligible to humans, the problem of finding intelligible proofs would be reduced to that of finding a recognition algorithm for intelligible proofs. Similar remarks apply to diverse creative human endeavors, such as designing airplane wings, creating physical theories, or even composing music. The question in each case is to what extent an efficient algorithm for recognizing a good result can be found.*

S Aaronson:

> *If $P = NP$, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett.*

A Widgerson:

> *The seemingly abstract, philosophical question: Can creativity be automated? in its concrete, mathematical form: Does $P = NP$?, emerges as a central challenge of science. And the basic notions of space and time, studied as resources of efficient computation, emerge as key objects of study to solving this mystery, just like their physical counterparts hold the key to understanding the laws of nature.*

Guessing whether $P = NP$: the dominant view

First we address the intuition around the conjecture that $P \neq NP$. One way to think about the question is that a problem is in P if finding a solution is fast, while a problem is in NP if verifying the correctness of a given witness is fast. Then the claim that $P \subseteq NP$ becomes the observation that if a problem is fast to solve then it must be fast to verify. But the other inclusion seems to most experts to be extremely unlikely. For example, speaking informally, S Aaronson has said, "I'd give it a 2 to 3 percent chance that P equals NP. Those are the betting odds that I'd take." Similarly, R Williams puts the chance that $P \neq NP$ at 80%.

Guessing whether $P = NP$: the dominant view

First we address the intuition around the conjecture that $P \neq NP$. One way to think about the question is that a problem is in P if finding a solution is fast, while a problem is in NP if verifying the correctness of a given witness is fast. Then the claim that $P \subseteq NP$ becomes the observation that if a problem is fast to solve then it must be fast to verify. But the other inclusion seems to most experts to be extremely unlikely. For example, speaking informally, S Aaronson has said, "I'd give it a 2 to 3 percent chance that P equals NP. Those are the betting odds that I'd take." Similarly, R Williams puts the chance that $P \neq NP$ at 80%.

Related reasoning is that there are thousands of NP complete problems spread across disciplines such as combinatorics, number theory, geometry, economics and finance, biology, operations research, etc. These problems differ wildely and bear no obvious relation to each other, besides the fact that they are all NP complete. Because they are NP complete, they are mutually inter-reducible, so a polytime solution to one would give a polytime solution to all. No one has been able to find a polytime algorithm to solve any of them. This is strong empirical evidence — not proof, to be sure, but still strong evidence — that $P \neq NP$.

Guessing whether $P = NP$: the contrarian view

Many observers have noted that there are cases where everyone "knew" that some algorithm was the fastest but in the end it proved not to be so. The section on Big-$\mathcal{O}$ begins with one, the grade school algorithm for multiplication. Another is the problem of solving systems of linear equations. The Gauss's Method algorithm, which runs in time $\mathcal{O}(n^3)$, is perfectly natural and had been known for centuries without anyone making improvements. However, while trying to prove that Gauss's Method is optimal, V Strassen found a $\mathcal{O}(n^{\lg 7})$ method ($\lg 7 \approx 2.81$).

A more dramatic speedup happens with the Matching problem. It starts with a graph whose vertices represent people and such that pairs of vertices are connected if the people are compatible. We want a set of edges that is maximal, and such that no two edges share a vertex. The naive algorithm tries all possible match sets, which takes 2^m checks where m is the number of edges. Even with only a hundred people there are more things to try than atoms in the universe. But since the 1960's we have an algorithm that runs in polytime.

Every day on the Theory of Computing blog feed there are examples of researchers producing algorithms faster than the ones previously known. A person can certainly have the sense that we are only just starting to explore what is possible with algorithms. R J Lipton captured this feeling.

> *Since we are constantly discovering new ways to program our "machines," why not a discovery that shows how to factor? or how to solve* SAT*? Why are we all so sure that there are no great new programming methods still to be discovered? . . . I am puzzled that so many are convinced that these problems could not fall to new programming tricks, yet that is what is done each and every day in their own research.*

Knuth has a related but somewhat different take.

> *Some of my reasoning is admittedly naive: It's hard to believe that $P \neq NP$ and that so many brilliant people have failed to discover why. On the other hand if you imagine a number M that's finite but incredibly large . . . then there's a humongous number of possible algorithms that do n^M bitwise addition or shift operations on n given bits, and it's really hard to believe that all of those algorithms fail.*
>
> *My main point, however, is that I don't believe that the equality $P = NP$ will turn out to be helpful even if it is proved, because such a proof will almost surely be nonconstructive. Although I think M probably exists, I also think human beings will never know such a value. I even suspect that nobody will even know an upper bound on M.*
>
> *Mathematics is full of examples where something is proved to exist, yet the proof tells us nothing about how to find it. Knowledge of the mere existence of an algorithm is completely different from the knowledge of an actual algorithm.*