Computational complexity

Freely using various textbooks, mainly the one by Arora-Barak

Péter Gács

Computer Science Department
Boston University

Fall 2019

It is best not to print these slides, but rather to download them frequently, since they will probably evolve during the semester.

The class structure

See the course homepage.
In the notes, section numbers and titles generally refer to the book:
Arora-Barak: Computational Complexity.

Description complexity

See the lecture linked to the schedule in the course homepage.

- The paradox of randomness: what does it mean that the sequence

  0101010101010101010101010101010101010101010101

  is unlikely to come from a sequence of coin tosses?
- The definition of complexity

$$K_M(x) = \min_{M(p)=x} |p|.$$

Weak dependence on the machine M.

Theorem (Invariance) There is an "optimal" machine U with the following property. For every machine M there is a c_M such that for all x we have $K_U(x) \leq K_M(x) + c_M$.

Proof sketch. Let U be a optimal universal machine, that can simulate the work of any machine M, given a description q_M of M. If $M(p) = x$ then $U(q_M p) = x$. □

We will fix such an optimal universal machine U, and write $K(x) = K_U(x)$.

Upper and lower bounds

Theorem There is a constant c such that for any binary string x, we have $K(x) \leq |x| + c$.

Proof. Let p be a program saying that the data string following it must be printed: then $U(px) = x$. □

Theorem For all $n > k > 0$, the number of binary strings x with $K(x) < n - k$ is less than 2^{n-k}.

Proof. There are at most i different programs of length i, hence the number of strings with complexity i is at most 2^i. We can then estimate the number in question as

$$1 + 2 + 4 + \cdots + 2^{n-k-1} = 2^{n-k} - 1.$$

□

The function $n - K(x)$ seems a useful measure of non-randomness of a binary string x of length n. Unfortunately:

Theorem No machine can compute $K(x)$.

Proof. Suppose there was such a machine M, then U could simulate it. Then there is a program p such that for any binary string $\langle m \rangle$ denoting a number m, $U(p\langle m \rangle)$ is the first string x_m with $K(x) > m$. We have

$$m < K(x) \leq |p| + \log m,$$

a contradiction for large m. □

This proof is a formalization of a famous paradox: the sentence "The smallest number not definable with fewer than 200 characters." defines a number in fewer than 200 characters.

TeX notation (say in email)

- "Control sequences" begin with `\`.
- `a \le b` for $a \le b$, `a_i` for a_i, `a^{25}` for a^{25}, `x \in A` for $x \in A$, `X \cup Y` for $X \cup Y$, `X \cap Y` for $X \cap Y$, `X \subseteq Y` for $X \subseteq Y$, `X \setminus Y` for $X \setminus Y$, `\alpha` for α, and so on.
- If you want a math formula to be compiled, then in a TeX file put it between $ signs.
  In a Piazza question/answer, put it between $$ signs.

Notation for languages

Alphabet, string, length $|\cdot|$, binary alphabet.
Empty string e.
Set Σ^* of all strings in alphabet Σ.
Lexicographical enumeration.

Encodings

Machines can only handle strings. Other objects (numbers, tuples) will be encoded into strings in some standard way. Let us use codes that show their own end, and can therefore concatenated freely.

Example Let $'[','],',' \notin \Sigma$, then we can encode elements $u = s_1 \dots s_n$ of Σ^* by themselves as $\langle u \rangle = u$, and pairs (u, v) as

$$\langle u, v \rangle = [u, v].$$

For example, $\langle 0110, 10 \rangle = [0110, 10]$.

For a natural number x, we may denote by $\langle x \rangle$ the code of its binary representation, and again $\langle x, y \rangle = [\langle x \rangle, \langle y \rangle]$.
Triples, quadruples, or arbitrary finite sequences of natural numbers are handled similarly.

A relation can be viewed as a set of pairs and encoded as a language.

Example Encoding the relation

$$\{ (x, y) \in \mathbb{N}^2 : x \text{ divides } y \}$$

as a language

$$\{ \langle x, y \rangle \in \{0, 1, ', [',], ', ', \}^* : x, y \in \mathbb{N},\ x \text{ divides } y \}.$$

Encoding a function over strings or natural numbers: first, its graph as a relation, then this relation as a language.
The cardinality of the set of all languages: see later.

Turing machines

(Definition partly taken from the Lovász notes.)

- **a** k doubly infinite tapes, tape symbol alphabet Σ.
- **b** Read-write heads.
- **c** Control unit with state space Γ.

Configuration: (control state, the tape contents, head position).
Transition functions

$$\alpha : \Gamma \times \Sigma^k \to \Gamma,$$
$$\beta : \Gamma \times \Sigma^k \to \Sigma^k,$$
$$\gamma : \Gamma \times \Sigma^k \to \{-1, 0, 1\}^k$$

define the machine $M(\Gamma, \Sigma, \alpha, \beta, \gamma)$. They are applied repeatedly to transform a configuration into a next one.

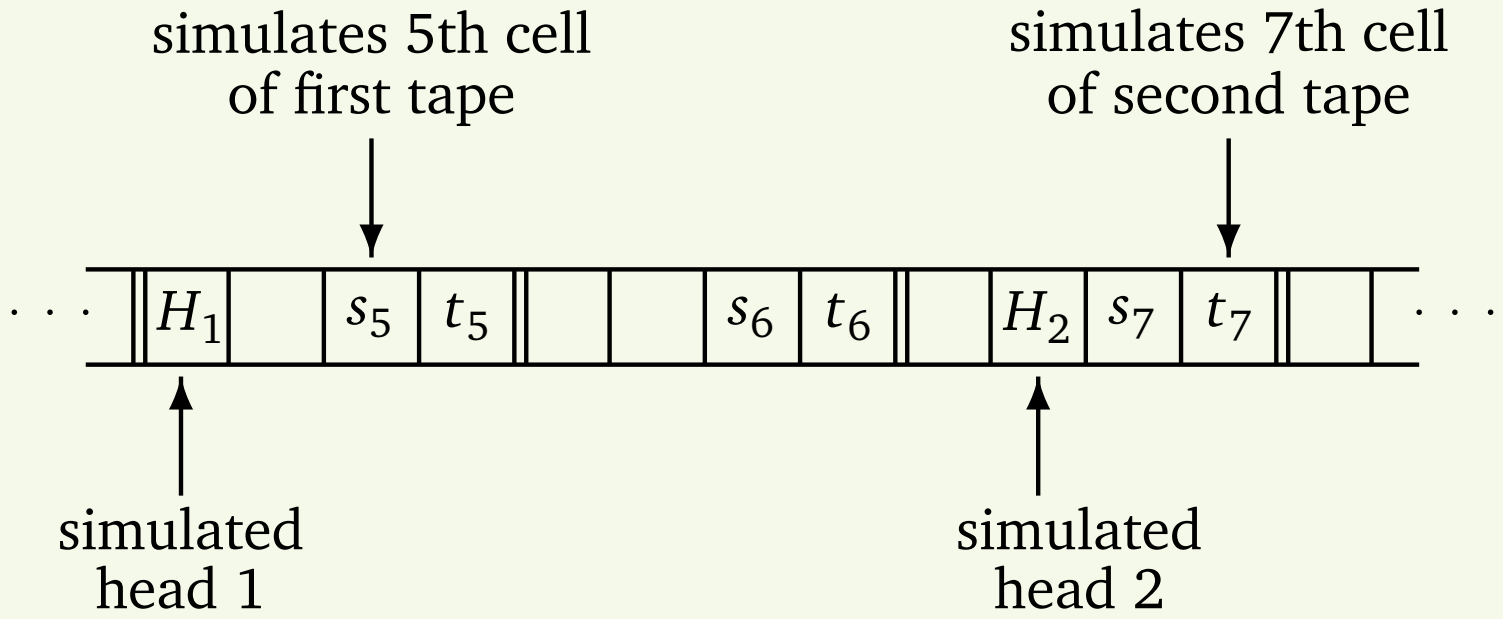
Conventions

- A part of the alphabet $\Sigma_0 \subseteq \Sigma$ will be called the input-output alphabet. Assume it always contains $0, 1$, while $\Sigma \setminus \Sigma_0$ always contains a blank symbol $\sqcup$. Normally, only finitely many positions on the tape will not be blank.
  Sometimes, some tapes are designated for input, and some for output.
- The set of states Γ contains two distinguished states: starting state q_{start}, and halting state q_{halt}.
- Computing a function $f : \Sigma_0^a \to \Sigma_0^b$: input and output conventions. The input strings are the maximal strings of input-output symbols found at the beginning of input tapes. Similarly for output strings, after halting.
- Examples of Turing machines computing some simple functions.
- A set of strings $L \subseteq \Sigma^*$ is sometimes called a language (a decision problem).

Variants on the definition: simulations

- One-sided tapes, only left-right moves (no staying in place), etc.
- The notion of simulation of machine M_2 by machine M_1: only the input-output behavior is reproduced.
  In practice: representing the data structure of M_2 in that of M_1, and "programming" the update. Each step of M_2 will be simulated by several steps of M_1.

Two tapes



If $s(x)$ is the memory requirement and $t(x)$ is the time requirement of the 2-tape machine on input x, then the time requirement of the 1-tape simulation is $O(s(x)t(x)) = O(t^2(x))$.

Simulating more complex machines

- 2-dimensional tape: several possibilities.
- The solution using (address, content) pairs is generalizable: say, to a machine whose storage structure is a tree.

The random access machine

Memory a (potentially) infinite sequence $x[0], x[1], x[2], \ldots$ of memory registers each containing an integer.

Program store a (potentially) infinite sequence of registers containing instructions.

$$x[i] := 0; \qquad x[i] := x[i] + 1; \qquad x[i] := x[i] - 1;$$
$$x[i] := x[i] + x[j]; \qquad x[i] := x[i] - x[j];$$
$$x[i] := x[x[j]]; \qquad x[x[i]] := x[j];$$
$$\textbf{if } x[i] \leq 0 \textbf{ then goto } p.$$

Input-output conventions.

How to define running time?

Simulations between the RAM and Turing machines. There is at most a $t \mapsto t^2$ slowdown.

The concept of an algorithm: Church thesis

Church-Turing Thesis This says that any algorithm defined in any "reasonable" formalism is implementable by some Turing machine.

Not a theorem, since it refers to unspecified formalisms. The above examples are part of its justification.

History Different formal definitions by Church (lambda calculus), Gödel (general recursive functions), Turing (you know what), Post (formal systems), Markov (a different kind of formal system), Kolmogorov (spider machine on a graph) all turned out all to be equivalent.

Algorithm any procedure that can be translated into a Turing machine. (or, equivalently, into a program on a universal Turing machine, see later)).

Uses of Church's Thesis

Justified Proving that something is not computable by Turing machines, we conclude that it is also not computable by any algorithm.

Unjustified Giving an informal algorithm for the solution of a problem, and referring to Church's thesis to imply that it can be translated into a Turing machine. It is your responsibility to make sure the algorithm is implementable: otherwise, it is not really an algorithm. Informality can be justified by common experience between writer and reader, but not by Church's Thesis.

More encodings

Consider some Turing machine $M(\Gamma, \Sigma, \alpha, \beta, \gamma)$.

- Elements s of alphabet Σ are encoded into strings of $\{0, 1, \#\}$ as $\langle s \rangle$ (binary string followed by $\#$). For any one alphabet Σ, we make all code strings $\langle s \rangle$ the same length.
- States q are encoded similarly into strings $\langle q \rangle$.
- The numbers $-1, 0, 1$ are encoded into, say, $\langle -1 \rangle = 10\#$, $\langle 0 \rangle = 00\#$, $\langle 1 \rangle = 01\#$.
- For each pair q, a, let $q' = \alpha(q, a)$, $a' = \beta q, a$, $\sigma = \gamma(q, a)$. The tuple (q, a, q', a', σ) describes the action of M on observing (q, a), and is encoded into $E(q, a) = \langle q \rangle \langle a \rangle \langle q' \rangle \langle a' \rangle \langle \sigma \rangle$.
- The whole transition function of M is encoded into

$$\langle M \rangle = E(q_1, a_1) E(q_2, a_2) \ldots E(q_n, a_n),$$

where (q_i, a_i) runs through all pairs (q, a).

Universal machine

- Machine U takes two input strings in the input-output alphabet Σ_0, and simulates all one-input machines M over this same input-output alphabet (the tape alphabets are not restricted). It is universal if $U(\langle M\rangle, x) = M(x)$ for all M, x. (When $M(x)$ does not halt, then $U(\langle M\rangle, x)$ should not halt either.)
- $U(p, x)$ is a partial function: undefined whenever U does not halt after producing output on input (p, x).
- Construction: simulating k tapes of one machine with $k+2$ tapes of a universal machine. (See an explicit program in the Lovász notes.)
  Tape $k+1$ represents the transition table, tape $k+2$ represents the current state.
- The number of tapes can be reduced further, of course.

The efficiency of the universal simulation

- The slowdown is only a constant factor, but this factor is huge. Indeed, simulating every step of M involves passing through its whole transition table. (Just imagine listing all possible control states of your laptop computer.)
- The simulation can be made much faster if, for example, the transition function is computed by a logic circuit (like in your laptop computer).

Matrix picture

Imagine $(M, w) \mapsto M(w) = U(\langle M \rangle, w)$ in a matrix with rows indexed by $\langle M \rangle$ and columns indexed by w: at position $(\langle M \rangle, w)$ sits the result $M(w)$, if it is defined, namely if the computation of M halts on input w. Let us put ∞ where it does not.

	$w_0 = e$	$w_1 = 0$	$w_2 = 1$	$w_3 = 00$	$\ldots$
$\langle M_1 \rangle$	e	∞	0001	e	$\ldots$
$\langle M_2 \rangle$					
$\langle M_3 \rangle$	$\langle M_3(e) \rangle = 111$	$\langle M_3(0) \rangle = 010$	$\langle M_3(1) \rangle = \infty$	$\langle M_3(00) \rangle = \infty$	$\ldots$
$\langle M_4 \rangle$					
$\vdots$					$\ddots$

The diagonal (partial) function $D(x) = U(x, x)$ is computable but its complement,

$$\overline{D}(x) = \begin{cases} 0 & \text{if } U(x, x) \neq 0 \\ 1 & \text{otherwise.} \end{cases}$$

is not: if it was, then there would be a k with $U(k, x) = \overline{D}(x)$. But $\overline{D}(k)$ was made different from $U(k, k)$!

Halting problem

Let Halting be the set of pairs (p, x) on which $U(p, x)$ halts.

Theorem (Halting) The set Halting is (algorithmically) undecidable.

Proof. If it was decidable then with its help, we could compute $\overline{D}(x)$. □

This proof is a typical (but simple) example of a reduction: we reduce the computation problem of $\overline{D}$ to that of Halting.

Diagonal method

The method used to show that $\overline{D}(x)$ is undecidable, is called the diagonal method.

- First used by Cantor to show that the real numbers (as represented, say, by infinite decimal fractions) cannot be listed in a sequence.
- Many uses in computer science, mainly to prove hierarchy theorems.

Sample hierarchy theorem

Theorem There is a function not computable in n^2 steps (as the function of the length of input) on any 2-tape Turing machine, but computable in $O(n^2 \log n \log\log n)$ steps on some 2-tape Turing machine.

Proof: (Idea: the diagonal construction yields a function that is computable, but in time longer than the bound used to define it.) Let M_p be the 2-tape machine with program p. Form the "diagonal" $E(x)$ (similar to $\overline{D}(x)$ before) as follows: Let $S(p)$ be a string of 0's of length $2^{2^{|p|}}$. For $x = pS(p)$ for some p we define

$$E(x) = \begin{cases} 0 & \text{if } x = pS(p) \text{ and } M_p(x) \neq 0 \\ & \text{is computed in time} \leq |x|^2 \text{ on } M_p, \\ 1 & \text{otherwise.} \end{cases}$$

- There is no 2-tape machine M_p computing $E(x)$ in time $|x|^2$. Indeed, if there were one then it would compute $M_p(x)$ in time $\leq |x|^2$, but $E(x)$ is defined for $|x| = pS(p)$ to be different from this value.
- Let us define a 4-tape machine computing $E(x)$ in time $O(|x|^2 \log\log x)$. A 4-tape universal Turing machine $U(p, x)$ simulates M_p with description p and running time t on input x in time $O(|p| \cdot t)$. Within such a time it can also keep track of the number of simulated steps and notice when it would exceed $|x|^2$. Modifying $U(p, x)$ we can compute $E(x)$ for $x = pS(p)$ and $n = |p|$ in time $O(n(n + 2^{2^n})^2) = O(|x|^2 \log\log|x|)$.
- By a non-trivial method any 4-tape machine can be simulated on a 2-tape machine in time $O(t \log t)$, this adds a factor of $O(\log|x|)$.

Other undecidable problems

- Many undecidable questions are purely mathematical, unrelated to Turing machines. The proof of undecidability is typically reduction to $D(x)$, but these reductions can be very complex.
- Consider equations of the form:

$$x^2 + 3xz^3 - 2y^4 - 6 = 0.$$

Every such polynomial equation can be viewed as a string of symbols E. Let $\mathcal{D}$ be the set of such strings E for which the corresponding equation has solution in integers $x, y, z, \ldots$ (these equations are called Diophantine equations). By a famous theorem, the set $\mathcal{D}$ is undecidable.

Some history

- The mathematician Hilbert asked in 1900 as one of his 23 famous challenges for mathematics in the coming century, to find a solution method for general Diophantine equations. There was no notion of undecidability at the time.
- The notion of computability was defined in the 1930's, in works of Gödel, Church and Turing. The theorem that the solvability of Diophantine equations is undecidable is due to Matiasevich, relying on earlier works of Davis, Putnam and Robinson.

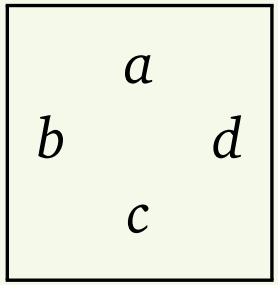
Let $\mathsf{Halting}'$ be the set of all (codes of) Turing machines that halt on empty input.

Theorem The set $\mathsf{Halting}'$ is undecidable.

Proof. Reducing $\mathsf{Halting}$ to $\mathsf{Halting}'$: for each pair (p, x) we construct a machine $M_{p,x}$ that on empty input does the same as U on input (p, x). If we could decide whether $M_{p,x}$ halts, we could decide $\mathsf{Halting}$. □

Tiling

- Prototile: a square shape, with a symbol on each side:



- Tile: an exact copy of some prototile.
- Kit: a finite set of prototiles.
- Initialized kit: a kit with a distinguished initial tile t: so it is given as a pair (K, t).
- Given a kit K, tiling the whole plane with K (if possible).
- For an initialized kit, we require that the tiling of the plane contains the initial tile.

- $\mathcal{L}_{\text{TLNG}}$ is the set of (encodings of) kits for which tiling the plane is possible, and $\mathcal{L}_{\text{NTLNG}}$ the set of those for which it is not.
- $\mathcal{L}'_{\text{TLNG}}$ is the set of initialized kits tiling the plane.

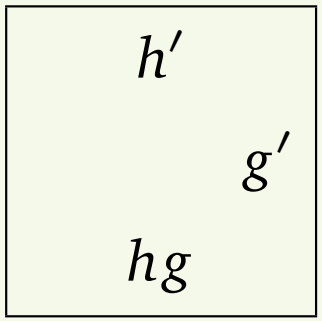
Theorem The set $\mathcal{L}'_{\text{TLNG}}$ is undecidable.

So it is undecidable about an initialized kit whether it tiles the plane. The set $\mathcal{L}_{\text{TLNG}}$ is also undecidable, but that theorem is harder to prove.

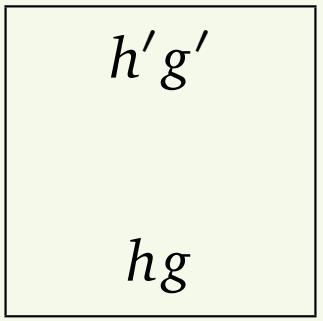
Proof idea. For each Turing machine M, we can construct a kit that tiles the plane if and only if M does not halt on empty input. As we put down these tiles, the rows will have to represent simulate subsequent configurations in a Turing machine computation. The next slides illustrate the kit used. □

$*$ $*$	1 1	2 2	1 g_1 $*g_2$	$*g_1$ g_1 $*$	
$*$ $*$	1 1	2 g_2 $*g_1$	$*g_2$ g_2 $*$	$*$ $*$	
$*$ $*$	1 g_1 $*$START	$*g_1$ g_1 $*$	$*$ $*$	$*$ $*$	
$*$ N N $*$	$*$START N P START$*$	$*$ P P $*$	$*$ P P $*$	$*$ P P $*$	

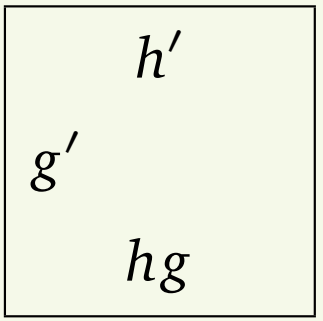
A tiling simulating a sample computation.



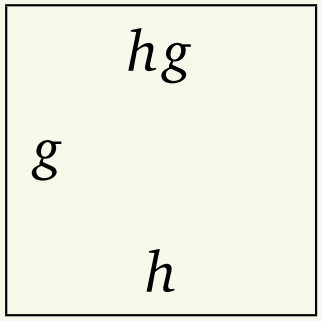
$\alpha(g,h) = g'$
$\beta(g,h) = h'$
$\gamma(g,h) = 1$



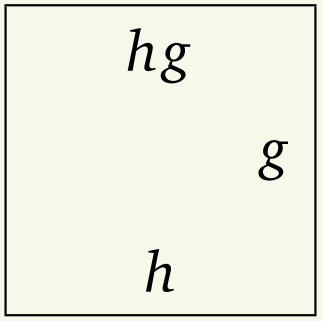
$\alpha(g,h) = g'$
$\beta(g,h) = h'$
$\gamma(g,h) = 0$



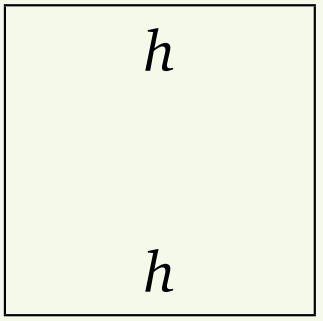
$\alpha(g,h) = g'$
$\beta(g,h) = h'$
$\gamma(g,h) = -1$



a)

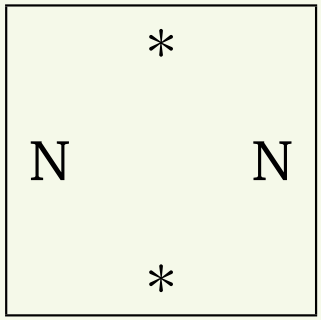


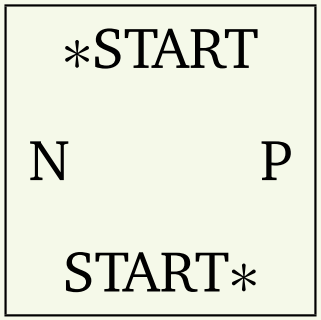
b)

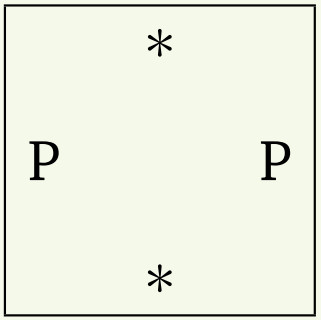


c)

Tiles around the simulated head position.







Tiles in the starting row.
For tiling the lower half-plane, add new tiles obtained by reflecting above tiles along the horizontal axis, and reversing the order of symbols in the labels of horizontal edges.

Some context for the undecidability of tiling.

Theorem If a kit K can tile every finite square then it can tile the whole plane.

Proof. Let $S_1 \subset S_2 \subset \cdots$ be a sequence of squares where $\bigcup_i S_i$ is the whole plane. For $i < j$ let T be a tiling of S_i and U a tiling of S_j. We will write $U > T$ if U is an extension of T.
Since each S_i can be tiled, there is a tiling T_1 of S_1 with infinitely many tilings $U > T_1$. Similarly, there is a tiling $T_2 > T_1$ with infinitely many tilings $U > T_2$, and so on. The union of the tilings $T_1 < T_2 < \cdots$ will tile the whole plain. □

This proof is not constructive: each step requires to check infinitely many conditions. And indeed, there is a kit K that tiles the whole plain but each of its tilings is uncomputable.

Corollary If K cannot tile the whole plane then there is some finite square that it cannot tile.

A tiling of the whole plane is called periodic if there is a tiling of a square (let us call it a repeating base) such that the whole tiling is obtained by just using disjoint copies of this tiled square. The undecidability result and the last corollary imply the following.

Theorem There is a set of tiles that tiles the whole plane but none of its tilings is periodic.

Proof. Given a kit K we can start a computation A checking for sizes $n = 1, 2, 3, \ldots$, whether a square of size n has a tiling. By the corollary above, if K cannot tile the plane then B terminates telling this.
It is also possible to start a computation B that searches for all possible repeating bases, and eventually finds one if one exists. If K can tile the plane periodically then B terminates telling this.
If every kit that tiles the plane tiles it also periodically then this algorithm would always terminate, and the tiling problem would be decidable. □

Undecidability in logic

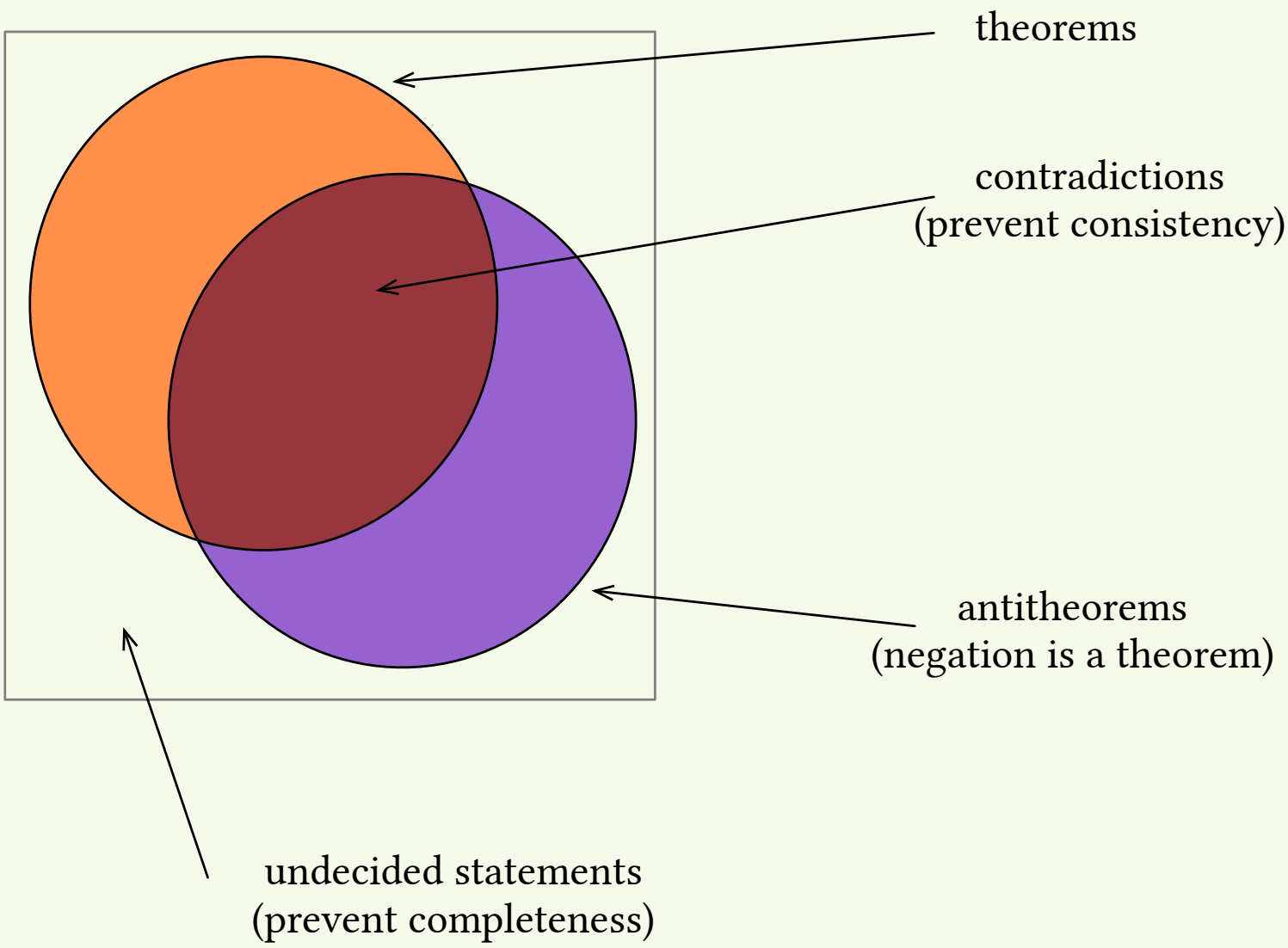
Sentences Strings in some alphabet that are statements about some mathematical objects.

Negation Assume that from each sentence s, another sentence s' called its negation can be computed.

Proof of some sentence T is a finite string P (in possibly another alphabet) that is proposed as an argument that T is true.

Formal system or theory $\mathcal{F}$ is an algorithm to decide, for any pair (P, T) of strings whether P is an acceptable proof for T. A sentence T for which there is a proof in $\mathcal{F}$ is called a theorem of the theory $\mathcal{F}$.

- A theory is called consistent if for no sentence can both it and its negation be a theorem.
  Inconsistent theories are uninteresting, but sometimes we do not know whether a theory is consistent.
- A sentence S is (logically) undecidable in a theory $\mathcal{T}$ if neither S nor its negation is a theorem in $\mathcal{T}$.
- A consistent theory is complete if it decides all sentences.


theorems
contradictions
(prevent consistency)
antitheorems
(negation is a theorem)
undecided statements
(prevent completeness)

Theorem If a theory $\mathcal{T}$ is complete then there is an algorithm that for each sentence S finds in $\mathcal{T}$ a proof either for S or for the negation of S.

Proof. Start listing all proof candidates of S and of its negation, and keep checking them. Eventually either a proof of S or a proof of its negation will be found. This is a program to actually find out whether S is provable in $\mathcal{T}$. □

- A theory $\mathcal{T}$ dealing with natural numbers is called rich if there is an algorithm computing for each (p, x), a sentence $\phi(p, x)$ that is a theorem of $\mathcal{T}$ if and only if $U(p, x)$ halts.
- There are well-known theories that are rich: one only needs a language with simple relations between (number-codes of) strings that help express how one TM configuraton gives rise to another. Then a proof of $U(p, x)$ halting can just trace the sequence of configurations starting from (p, x).

Theorem (Gödel's incompleteness theorem)

Every rich theory is incomplete.

Proof. If the theory were complete then, as shown, it would give a procedure to decide algorithmically the halting problem. □

- If the last proof is expanded by making diagonalizaton explicit then for any rich theory $\mathcal{T}$ a pair (p, x) is found such that
  - "$U(p, x)$ does not halt" is expressible in $\mathcal{T}$ and true.
  - it is not provable in $\mathcal{T}$.
- There are other, more interesting, sentences that are not provable, if only the theory $\mathcal{T}$ is even richer: Gödel proved that assertions expressing the consistency of $\mathcal{T}$ are among these. This is the Second Incompleteness Theorem of Gödel.
- Historically, Gödel's incompleteness theorems preceded the notion of computability by 3-4 years.

Complexity classes

Complexity of a problem (informally): the complexity of the "best" algorithm solving it.

Problems

- Compute a function
- Decide a language
- Given a relation $R(x, y)$, for input string x find an output string y for which $R(x, y)$ is true. Example: $R(x, y)$ means that the integer y is a proper divisor of the integer x.

What is the running time as a function of the length of input?

Given an algorithm, this is often a nontrivial question.

Example running time estimate

Gaussian elimination

- Solving a set of linear equations by Gaussian elimination: $O(n^3)$ algebraic operations $(\pm, \cdot, \div)$.
- Turing machine complexity=bit complexity. The problem of round-off errors.
- Rational inputs, exact rational solution: How large can the numerators and denominators grow? In principle, adding fractions: $\frac{a}{b} + \frac{a'}{b'} = \frac{ab'+a'b}{bb'} =: \frac{A}{B}$ may double the number of bits in the denumerator: potential of exponential increase of the lenght of numbers.
- Reducing $\frac{A}{B}$: divide by $\gcd(A, B)$ (you know this is computable in polynomial time from A, B).
- New algorithm: Gaussian elimination combined with the above reduction after each step.

Theorem In the new algorithm, the length of numbers remains polynomial in the length of the input.

For a proof, look in the Lovász notes.
Hence Gaussian elimination can be done in polynomial time: that is, in a polynomial number of bit operations, not only algebraic operations.

Upper and lower bounds

DTIME$(f(n))$: a class of languages.

Upper bound given a language L and a time-measuring function $g(n)$, showing $L \in$ **DTIME**$(g(n))$.

Lower bound given a language L and a time-measuring function $g(n)$, showing $L \notin$ **DTIME**$(g(n))$.

Example Let **DTIME**$(\cdot)$ be defined using 1-tape Turing machines, and let $L_1 = \{uu : u \in \Sigma^*\}$. Then it can be proved that

$$L_1 \notin \textbf{DTIME}(n^{1.5}).$$

The difficulty of proving a lower bound: this is a statement about all possible algorithms.

Why complexity classes?

Why we are just speaking about complexity classes, rather than the complexity of a particular problem.
Sometimes there is no “best” algorithm for a given problem. See the so-called speedup theorems.

Why language classes?

Sometimes, there are trivial lower bounds for functions: namely, $|f(x)|$ (the length of $f(x)$) is a lower bound.

Example $f(x, y) = x^y$ where the binary strings x, y are treated as numbers.

Naive algorithm $x \cdot x \cdot \cdots \cdot x$ (y times). This takes y multiplications, so it is clearly exponential in the length of y.

Repeated squaring now the number of multiplications is polynomial in $|y|$.

But no matter what we do, the output length is $|x^y| \approx y \cdot |x|$, exponential in $|y|$.
(Still, repeated squaring gives a polynomial algorithm for computing $(a, b, m) \mapsto a^b \bmod m$).

If the function values are restricted to $\{0, 1\}$ (deciding a language) then there are no such trivial lower bounds.

Polynomial time

Invariance with respect to machine model

$$\mathbf{P} = \bigcup_c \mathbf{DTIME}(n^c), \quad \mathbf{EXP} = \bigcup_c \mathbf{DTIME}(2^{n^c}).$$

2-tape Turing machines and even 2-dimensional and random-access machines can be simulated by 1-Tape Turing machines, with a slowdown similar to $t \mapsto t^2$. Therefore to some questions ("is there a polynomial-time algorithm to compute function f?") the answer is the same on all "reasonable" machine models. (Caveat about quantum.)

Some examples

- PATH: find the shortest path between points s and t in a graph. Breadth-first search.
- The same problem, when the edges have positive integer lengths. Reducing it to PATH in the obvious way (each edge turned into a path consisting of unit-length edges) may result in an exponential algorithm (if edge lengths are large).
- Dijkstra's algorithm works in polynomial time also with large edge lengths.

The greatest common divisor of two numbers can be computed in polynomial time, using:

Theorem $\gcd(a, b) = \gcd(b, a \bmod b)$.

This gives rise to Euclid's algorithm.
Why polynomial-time?

Does it capture “practical”?

- We may extend the class by allowing randomization—see later.
- It may miss the point. On small data, an $0.001 \cdot 2^{0.1n}$ algorithm is better than a $1000n^3$ algorithm.

Still, in typical situations, the lack of a polynomial-time algorithm means that we have no better idea for solving our problem than “brute force”: a run through “all possibilities”.

NP problems

Examples

- Shortest vs. longest simple paths
- Euler tour vs. Hamiltonian cycle
- Ultrasound test of sex of fetus.

Decision problems vs. optimization problems vs. search problems.

Example Given a graph G.

Decision Given k, does G have an independent subset of size $\geq k$?

Optimization What is the size of the largest independent set?

Search Given k, give an independent set of size k (if there is one).

Optimization+search Give a maximum size independent set.

Polynomial-time verification

Example Hamiltonian cycles.

- For simplicity, we assume that all our problem instances are encoded into binary strings.
- An NP problem given by a polynomial $p(n)$, and a relation

$$V(x, w),$$

of binary strings with values in $\{0, 1\}$ that for a given input x and a candidate witness (certificate) w is computable in time $p(|x|)$ and verifies whether w is indeed witness for x. (Then necessarily $|w| \leq p(|x|)$).

- The language defined by the problem is the set of strings

$$L = \{\, x \in \{0,1\}^* : \exists\, w\; V(x, w) = 1 \,\}.$$

The class **NP** is the set of languages L definable this way.

The same decision problem may belong to very different verification functions (search problems).

Example (Compositeness) Let the decision problem be the question whether a number x is composite (nonprime). The obvious verifiable property is

$$V_1(x, w) \Leftrightarrow (1 < w < x) \wedge (w|x).$$

There is also a very different verifiable property $V_2(x, w)$ for compositeness such that, for a certain polynomial-time computable $b(x)$, if x is composite then at least half of the numbers $1 \leq w \leq b(x)$ are witnesses. This can be used for probabilistic prime number tests.

Nondeterminism

Nondeterministic machines are not modeling any real machines: you can view this concept rather as a fancy way of talking about the existential quantifier. But this concept went into the name of the class **NP**, so let us define them.

- Nondeterministic Turing machines: possibly more than allowed transition from any given configuration: instead of transition functions α, β, γ, we have transition relations.
- Such a machine accepts an input x if it has a sequence of transitions from x leading to "accept". A language L is accepted by machine M if L is the set of those inputs accepted by M.

Theorem A language L is in **NP** if and only if it is accepted by some polynomial-bounded machine.

Satisfiability

- Let us use Boolean variables $x_i \in \{0, 1\}$, where 0 stands for false, 1 for true. A logic expression is formed using the connectives $\wedge, \vee, \neg$: for example

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee x_4).$$

  Other connectives: say $x \Rightarrow y = \neg x \vee y$.
- An assignment (say $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 0$) allows to compute a value (in our example, $F(0, 0, 1, 0) = 0$).
- An assignment (a_1, a_2, a_3, a_4) satisfies F, if $F(a_1, a_2, a_3, a_4) = 1$. The formula is satisfiable if it has some satisfying assignment.
- Satisfiability problem: given a formula $F(x_1, \ldots, x_n)$ decide whether it is satisfiable.

Special cases:

- A conjunctive normal form (CNF) $F(x_1, \ldots, x_n) = C_1 \wedge \cdots \wedge C_k$ where each C_i is a clause, with the form $C_i = \tilde{x}_{j_1} \vee \cdots \vee \tilde{x}_{j_r}$. Here each $\tilde{x}_j$ is either x_j or $\neg x_j$, and is called a literal.
  SAT: the satisfiability problem for conjunctive normal forms.
- A 3-CNF is a conjunctive normal form in which each clause contains at most 3 literals—gives rise to 3SAT.
- 2SAT: as will be seen, solvable in polynomial time.

The algorithm explores the consequences of $\neg x \vee y \iff x \leq y$.

- Graph with directed edges between literals. For $\neg x \vee y$ for any literals x, y, add directed edge $x \to y$ and $\neg y \to \neg x$.
- Repeat: find directed cycle C. If it contains a literal and its negation we call it a contradiction. If there is one, the formula is nonsatisfiable: otherwise collapse C.
- Otherwise we end up with a directed acyclic graph, and now we know the formula is satisfiable.
- To satisfy, repeat: For each literal x, if there is no path from x to $\neg x$ then add $\neg x \to x$ and set $x = 1$, otherwise add $x \to \neg x$ and set $x = 0$.

Circuits

Logic formulas can be generalized to logic circuits (Boolean circuits if using $\wedge, \vee, \neg$).

- Acyclic directed graph, where some nodes and edges have labels. Nodes with no incoming edges are input nodes, each labeled by some logic variable $x_1, \ldots, x_n$.
  Nodes with no outgoing edges are output nodes.
- Some edges have labels $\neg$. Non-input nodes are labeled $\vee$ or $\wedge$.
- If there is just one output node, the circuit C defines some Boolean function $f_C(x_1, \ldots, x_n)$. Circuit satisfiability is the question of satisfiability of this function.
- Assume also that every non-input node has exactly two incoming edges.

Theorem (Well-known) For every Boolean function $f : \{0,1\}^n \to \{0,1\}^k$ there is a Boolean circuit C_f with n inputs and k outputs computing f.

- The circuit satisfiability problem.
- Specialize: formula satisfiability problem.
- Specialize: CNF satisfiability, that is SAT.
- Specialize: 3SAT.

On the other hand:

Theorem Circuit satisfiability can be reduced to 3SAT.

Our first nontrivial example of reduction.

Proof. Introduce a new variable y_i for the output of each gate. The relation of each gate output to its inputs can be expressed by a formula of at most 3 variables: for example $y_i \iff x_j \wedge y_k$.
Transform this into a 3CNF G_i. The conjunction of all these gives a 3CNF

$$F(x_1, \ldots, x_n, y_1, \ldots, y_m) = G_1 \wedge \cdots \wedge G_m,$$

where y_m is the output. The satisfiability of the circuit is equivalent to the satisfiability of $F(x_1, \ldots, x_n, y_1, \ldots, y_m) \wedge y_m$. □

Reducibility, completeness

Reduction of problem A_1 to problem A_2 in terms of the verification functions V_1, V_2 and a reduction (translation) function τ:

$$\exists w\, V_1(x, w) \Leftrightarrow \exists u\, V_2(\tau(x), u).$$

Examples

- Reducing linear programming to linear programming in standard form.
- Reducing satisfiability for circuits to 3SAT.

Use of reduction in this course: proving hardness.

- NP-hardness.
- NP-completeness.

Theorem Circuit satisfiability is **NP**-complete.

Consider a verification function $V(x, w)$. For an x of length n, to a Turing machine T computing $V(x, w)$, in cost t, construct a circuit C_x of polynomial size in n, t (easier to see going through a cellular automaton simulating T) that computes $V(x, w)$ from any input string w. (We translated x to C_x.) Now there is a witness w if and only if C_x is satisfiable.

Theorem 3SAT is **NP**-complete.

Translating a circuit's local rules into a 3-CNF.

Theorem INDEPENDENT SET is **NP**-complete.

Reducing SAT to it.

Example Set cover $\geq$ vertex cover $\sim$ independent set.

Integer linear programming

- Special case: solving $\mathbf{Ax} = \mathbf{b}$, where the $m \times n$ matrix $\mathbf{A} \geq 0$ and the vector $\mathbf{b}$ consist of integers, and $x_j \in \{0, 1\}$.
- Case $m = 1$ is the subset sum problem.
- Reducing SAT to this. Translate the clause $x_1 \vee x_2 \vee \neg x_3$ to

$$x_1 + x_2 + x_3' + y_1 + y_2 = 3,$$
$$x_3 + x_3' = 1.$$

- Trick to put always 1 to the right-hand side: use the fact that $x \leq y$ iff $x + (1 - y) + z = 1$ is solvable.
- When always 1 on the right-hand side: equivalent to the set partition problem.

Reducing two equations to one:

$$a_{11}x_1 + \cdots + a_{1n}x_n = b_1,$$
$$a_{21}x_1 + \cdots + a_{2n}x_n = b_2.$$

Let $D > a_{ij}, b_i$, and form

$$(a_{11} + Da_{21})x_1 + \cdots + (a_{1n} + Da_{2n})x_n = b_1 + Db_2.$$

If $x_1, \ldots, x_n$ satisfies this it satisfies both above. Iterating this trick reduces $\mathbf{Ax} = \mathbf{b}$ to subset sum.

Hamilton paths

Reducing the SAT to dHAMPATH, the problem of directed Hamilton paths.

- Points $v_{\text{start}}, v_{\text{end}}$, and one point for each of the m clauses C_j.
- For each of the n variables x_i, a doubly linked chain $X_i = v_{i,0} \longleftrightarrow v_{i,1} \longleftrightarrow \cdots \longleftrightarrow v_{i,3m-1} \longleftrightarrow v_{i,3m}$.
- $v_{\text{start}} \to v_{1,0}, v_{1,3m}$; $v_{n,0}, v_{n,3m} \to v_{\text{end}}$.
- $v_{i,0}, v_{i,3m} \to v_{i+1,0}, v_{i+1,3m}$ if $i < n$.
- If x_i occurs in C_j then $v_{i,3j-2} \to C_j \to v_{i,3j-1}$.
  If $\neg x_i$ occurs in C_j then $v_{i,3j-1} \to C_j \to v_{i,3j-2}$.

Making x_i true corresponds to traversing X_i from left to right.

- Definition of the coNP class: L is in **coNP** if its complement is in **NP**. Example: logical tautologies.
- The class **NP** $\cap$ **coNP**. Examples: duality theorems.
- Example of a class that is in **NP** $\cap$ **coNP**, and not known to be in **P**: derived from the factorization problem.
  Let L be the set of those pairs of integers $x > y > 0$ for which there is an integer $1 < w < y$ with $w|x$. This is clearly in **NP**. But the complement is also in **NP**. A witness that there is no w with the given properties is a complete factorization

$$x = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$$

  of x, along with witnesses of the primality of $p_1, \ldots, p_k$. The latter are known to exist, by an old—nontrivial—theorem that primality is in **NP**.

Set of equations $\mathbf{Ax} = \mathbf{b}$ (rational coefficients), looking for integer solution.

- Is this in **NP**?
  Yes, but not trivially. It follows if whenever there is a solution there is a polynomial-length one .
- If some $\mathbf{y}$ makes $\mathbf{y}^T\mathbf{A}$ integer but $\mathbf{y}^T\mathbf{b}$ is not then clearly no integer solution $\mathbf{x}$—since $\mathbf{y}$ "derives a contradiction".

Theorem If there is no integer solution $\mathbf{x}$ then there is such a $\mathbf{y}$.

Does this prove that the problem $\mathbf{Ax} = \mathbf{b}$ is in **coNP**?
Not quite, but yes if there is also a polynomial-size $\mathbf{y}$ in the theorem (also true).

Knapsack problem

The knapsack problem is defined as follows.
Given: integers $b \geq a_1, \ldots, a_n$, and integer weights $w_1 \geq \cdots \geq w_n$.

$$\begin{aligned} \text{maximize} \quad & \mathbf{w}^T \mathbf{x} \\ \text{subject to} \quad & \mathbf{a}^T \mathbf{x} \leq \quad b, \\ & x_i = 0, 1, \ i = 1, \ldots, n. \end{aligned}$$

Dynamic programming: For $1 \le k \le n$,

$$A_k(p) = \min\{\, \mathbf{a}^T\mathbf{x} : \mathbf{w}^T\mathbf{x} \ge p,\ x_{k+1} = \cdots = x_n = 0 \,\}.$$

If the set is empty the minimum is ∞. Let $w = w_1 + \cdots + w_n$. The vector $(A_{k+1}(0), \ldots, A_{k+1}(w))$ can be computed by a simple recursion from $(A_k(0), \ldots, A_k(w))$. Namely, if $w_{k+1} > p$ then $A_{k+1}(p) = A_k(p)$. Otherwise,

$$A_{k+1}(p) = \min\{A_k(p), a_{k+1} + A_k(p - w_{k+1})\}.$$

The optimum is $\max\{\, p : A_n(p) \le b \,\}$.
Complexity: roughly $O(nw)$ steps.
Why is this not a polynomial algorithm?

Related pairs of tractable and NP-hard problems

- 2-SAT and 3-SAT (though max 2-SAT is **NP**-hard)
- 2-coloring and 3-coloring
- 2-matching and 3-matching
- maximum matching and maximum independent set
- Euler paths and Hamilton paths
- Chinese postman problem and the traveling salesman problem

In defining $\mathbf{DSPACE}(g(n))$, count only the amount of work tape used. This allows to have even logarithmic space complexity: $\log n$ space suffices to keep track of the position of the input being scanned. Let $\mathbf{L} = \mathbf{DSPACE}(\log n)$, $\mathbf{NL} = \mathbf{NSPACE}(\log n)$.
Obvious relations:

$$\mathbf{DTIME}(S(n)) \subseteq \mathbf{DSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))}),$$

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}.$$

None of these relations is known to differ from equality. Some equalities cannot hold simultaneously: for example $\mathbf{P} \neq \mathbf{EXP}$ is easy and $\mathbf{NP} \neq \mathbf{NEXP}$ is also known.

What can be computed in polynomial space?

- Obviously **NP** $\subseteq$ **PSPACE**.
- But we can do also more complicated things in polynomial space than just trying out all possible witnesses. For example, we can evaluate very general games.

Games

For problems (that seem) beyond **NP**, consider a (2-person) board game:

- Length parameter n: all configurations are described by binary strings of length n. (The configuration shows whose turn it is.)
- For configurations C, C', the predicate $C \to C'$ says whether C' is allowed by a move from C. Assume it decidable in polynomial time.
- Final(C) says whether C is final (no move possible), and $W(C)$ for final configurations returns win/lose/draw. These are polynomial time.

Example Chess, or Go (boards of any size).

Decide a game

Algorithm $W(x,t)$ (recursive) evaluates configuration x at time t for the player whose turn it is.

if Final(x) **then return** $W(x)$
if $t \geq 2^n$ **then return** draw // Some configuration repeats.
for all configurations y with $x \to y$ **do**
    if $W(y, t+1) =$ lose **then return** win // Move worth choosing
for all configurations y with $x \to y$ **do**
    if $W(y, t+1) =$ draw **then return** draw
return lose

- Space requirement defined: board-size × stack-length (number of moves). At most exponential. But only polynomial for polynomial number of moves (still exponential time).
- For an exponential time bound: memoization (dynamic programming): store each result $W(x,t)$ once computed, and check before the recursive call. May use exponential space even with polynomial number of moves.

Polynomial-time games are PSPACE-complete

- We have decided in polynomial space whether a configuration x is winning in $p(n)$ moves: that is $W(x, p(n))$.
- For an arbitrary n-space computation, we will find a game.

For configurations of some Turing machine M, board

$$(t_1, C_1 \mid C_0 \mid t_2, C_2)$$

means that M has configurations C_i at time t_i, and C_0 at time $t_0 = \lfloor \frac{t_1+t_2}{2} \rfloor$. C_0 can be "?". Start from $(0, C \mid ? \mid 2^n, \text{"accept"})$.
Players: Prover wants to prove the acceptance; Verifier keeps testing.

Prover's move Fill in the question sign.

Verifier's move If $t_0 = t_1 + 1$ or $t_2 - 1$ then check. Move to $(t_1, C_1 \mid ? \mid t_0, C_0)$ or $(t_0, C_0 \mid ? \mid t_2, C_2)$.

No limitation of computational power on either Prover or Verifier.

We can standardize any quantified Boolean formula into a form—called prenex form where all quantifiers are in front.

- Bring negations inside using the de Morgan rules:

$$\neg(u \vee v) = \neg u \wedge \neg v, \quad \neg\exists x F(x) = \forall x \neg F(x),$$
$$\neg(u \wedge v) = \neg u \vee \neg v, \quad \neg\forall x F(x) = \exists x \neg F(x).$$

- Change the bound variables to prevent clashes:

$$\forall x F(x) \vee \forall x G(x) = \forall x F(x) \vee \forall y G(y).$$

- Bring quantifiers outside:

$$A \wedge \exists x F(x) = \exists x (A \wedge F(x)).$$

Game to formula

$C \to C'$ means there is a move from C to C'.
$C \to C' \to C'' := C \to C' \wedge C' \to C''$.
$C_0 :=$ the starting configuration. Starting player wins in 4 moves:

$$\exists C_1 \forall C_2 \exists C_3 \forall C_4 (C_0 \to C_1 \to C_2 \to C_3 \to C_4 \wedge C_4 \text{ loses})$$

- SAT is the special case of 1 move: $\exists X \phi(X)$.
- Any game can be expressed by such a quantified Boolean formula (length depends on the bound on the number of moves).
  Indeed, $C \to C'$ can be represented by a polynomial-size circuit; this is expressible by $\exists Z \phi(C, C', Z)$ with 3-CNF ϕ, and auxiliary variables $Z = (z_1, \ldots, z_k)$.
- Any quantified Boolean formula in prenex form $\exists X_1 \forall X_2 \exists X_3 \cdots \phi(X_1, \ldots, X_k)$ where $X_i = (x_{i1}, \ldots, x_{in})$, corresponds to some game with k moves.

Other PSPACE-complete games

Many tricky reductions have been made showing various simply-defined games **PSPACE**-complete.
Go and Checkers (on $n \times n$ boards) are among them.

NPSPACE=PSPACE

- Nondeterministic machine, language computed by it, **NP**, **NPSPACE**.
- Proof of **PSPACE**-completeness of games applies without change also to nondeterministic machines. So if $L \in$ **NPSPACE** then $x \in L$ iff x is a winning configuration in a certain game with polynomial number of moves.
- Hence **NPSPACE** = **PSPACE**.
- Generalizing further (without any new idea) shows

Theorem (Savitch) **NSPACE**$(S(n)) \subseteq$ **DSPACE**$((S(n))^2)$.

In particular, **NSPACE**$(\log n) \subseteq$ **DSPACE**$(\log^2 n)$. Not known whether the inclusion is strict, that is whether **L** = **NL**.

- A typical (actually, complete) problem in **NL**: Given a directed graph G with two points s, t in it, decide whether there is a directed path from s to r. (The $\log^2 n$-space algorithm may not be polynomial time!)

Exponential-time games are EXP-complete

Players: again Prover and a Verifier. Convenient to use cellular automata, transition function $C(a,b,c)$. State at time t at position i is $\eta(i,t)$. Let $\eta(i,0) = X_i$, $i = 0,\ldots,n-1$.
To test $\eta(0,2^n) =$ "accept". Board configuration:

$$(X \mid t,i \mid b \mid a_{-1},a_0,a_1)$$

where we expect $b = \eta(i,t)$, and $a_j = \eta(i+j,t-1)$ for $j = -1,0,1$, Each a_j can be "?".
Start with $(X \mid 2^n,0 \mid$ "accept" $\mid ?,?,?)$.
(The board does not represent a whole—possibly exponential size—CA configuration.)

Prover's move Replace the question marks.

Verifier's move If $t = 1$ check $a_j = X_{i+j}$ for $j = -1,0,1$.
Else check $b = C(a_{-1},a_0,a_1)$.
Then move to $(X \mid t-1,i+j \mid a_j \mid ?,?,?)$ where $j \in \{-1,0,1\}$.

Randomization or average case

Algorithms can be analyzed probabilistically from several points of view. First distinction:

1. The algorithm is deterministic, but we analyze it on random inputs.
2. We introduce randomness during computation, but the input is fixed.
3. Randomize and also analyze on random inputs.

Approach 1 is less frequently used, since we rarely have reliable information about the distribution of our inputs. Levin's theory of problems that are hard on average adresses general questions of this type.
Most practical uses of randomness belong to category 2, randomization.

Examples

- Quicksort, median
- Prime number tests

Matrix product testing

Given $n \times n$ integer matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, there is no known deterministic algorithm to test the equality $\mathbf{AB} = \mathbf{C}$ in time $O(n^2)$. The following algorithm will accept equality and reject inequality with probability $\geq 1/2$. Repeating it k times will reduce the probability of false positive to 2^{-k}.

- Choose a random vector $\mathbf{x}$ with entries from $\{-1, 1\}$.
- Compute $\mathbf{c} = \mathbf{Cx}$, $\mathbf{b} = \mathbf{Bx}$, $\mathbf{c}' = \mathbf{Ab}$.
  If $\mathbf{c} = \mathbf{c}'$, accept, else reject.

This algorithm takes $O(n^2)$ operations, and if $\mathbf{C} = \mathbf{AB}$ then it always accepts.

Claim Else, it accepts with probability $\leq 1/2$.

The proof will be in a homework.

Polynomial identity

Given two functions $f(x), g(x)$, is it true that $f(x) = g(x)$ for all input values x? The functions may be given by a formula, or by a complicated program.

Example Matrix product testing is the same as testing $\mathbf{A}(\mathbf{Bx}) = \mathbf{Cx}$ for all $\mathbf{x}$.

We will concentrate on the case when f, g are polynomials. Crucial fact from elementary algebra:

Proposition A degree d polynomial of one variable has at most d roots.

So, if we find $P(r) = 0$ on a random r, this can only happen if r hits one of the d roots.

What is this good for? Checking $f(x) = g(x)$ is trivial: compare all coefficients.
In the interesting applications, the polynomial has many variables, and is given only by computing instructions: for example as the output of an arithmetic circuit.

Example $\det(\mathbf{A}_1 x_1 + \cdots + \mathbf{A}_k x_k + \mathbf{A}_{k+1})$, where $\mathbf{A}_i$ are $n \times n$ integer matrices.

- When expanded, potentially exponential size in n, but for each fixed value of $(x_1, \ldots, x_k)$ there is a polynomial algorithm: Gaussian elimination.
- Rounding of fractions is not allowed, but you could do it modulo some prime number larger than the largest possible value of $\det \mathbf{A}$. (Find one using a randomized prime test.)
  Or with exact fractions, as discussed earlier.
- An arithmetic circuit with logarithmic depth is shown in the Lovász notes in the parallel algorithms section.

Estimate the probability of hitting a root in a multivariate polynomial.

Lemma (Schwartz-Zippel) Let $p(x_1, \ldots, x_m)$ be a nonzero polynomial, with variable x_i having degree at most d_i. If $r_1, \ldots, r_m$ are selected randomly from $\{1, \ldots, f\}$ then the probability that $p(r_1, \ldots, r_m) = 0$ is at most $(d_1 + \cdots + d_m)/f$.

Proof. Induction on m. Let $p(x_1, \ldots, x_m) = p_0 + x_1 p_1 + \cdots + x_1^{d_1} p_{d_1}$, where $p_{d_1} \neq 0$. Let $q(x_1) = p(x_1, r_2, \ldots, r_m)$. Two cases:

$$\begin{cases} p_{d_1}(r_2, \ldots, r_m) = 0 & \text{with probability } \leq (d_2 + \cdots + d_m)/f, \\ q(r_1) = 0 & \text{with probability } \leq d_1/f. \end{cases}$$

Total is $\leq (d_1 + \cdots + d_m)/f$. □

Example Given a bipartite graph $G = (U \cup V, E)$ with $|U| = |V| = n$, define the $n \times n$ matrix $\mathbf{A}(x_{11}, x_{12}, \ldots, x_{nn})$ where

$$a_{ij} = \begin{cases} x_{ij} & \text{if } (u_i, v_j) \text{ is an edge,} \\ 0 & \text{otherwise.} \end{cases}$$

Then $\det \mathbf{A}(x_{11}, \ldots, x_{nn})$ is identically 0 if and only if G has no perfect matching.
From here: a fast randomized test of the existence of perfect matching.

- Similar matrix (a little more complex proof) for non-bipartite graph $G = (V, E)$. If (v_i, v_j) is an edge for $i < j$ then let $a_{ij} = x_{ij}$, $a_{ji} = -x_{ij}$. Otherwise $a_{ij} = 0$.
- Both of these are new examples (beyond the prime number tests) of an alternative verification function for an **NP** language, leading to a large number of witnesses if there is one.

Randomized classes

With randomization we give up (almost always) some certainty, but what sort?

Monte-Carlo algorithm: we have a bound on the time, and on the probability that the result is wrong.

Las Vegas algorithm (no particular reason): result is always correct, but we bound the execution time statistically (say by expected value).

One-sided error

Definition A language L is in $\mathbf{R}(t(n))$ if there is a randomized algorithm A working in time $O(t(n))$ such that for all $x \in \Sigma^*$

- if $x \notin L$ then $A(x)$ rejects.
- if $x \in L$ then $A(x)$ accepts with probability $\geq 1/2$.

Let $\mathbf{RP} = \bigcup_k \mathbf{R}(n^k)$.

If we want $1 - 2^{-k}$ in place of $1/2$, we can repeat k times; this does not change the definition of **RP**.

Examples Matrix product is not a good example, since it is also easily in **P**.

- Compositeness of integers.
- Polynomial non-identity.

Contrast RP with NP

$L \in \mathbf{RP}$ if there is a k and a (deterministic) algorithm $A(x, r)$ running in time n^k with $x \in \Sigma^n$, $r \in \{0,1\}^{n^k}$ such that

- if $x \notin L$ then $A(x, r)$ rejects for all r.
- if $x \in L$ then $A(x, r)$ accepts for at least half of all values of r.

On the other hand $L \in \mathbf{NP}$ if there is a k and a (deterministic) algorithm $A(x, r)$ running in time n^k with $x \in \Sigma^n$, $r \in \{0,1\}^{n^k}$ such that

- if $x \notin L$ then $A(x, r)$ rejects for all r.
- if $x \in L$ then $A(x, r)$ accepts for at least one value of r.

The algorithm $A(x, r)$ in the **NP** definition is called the verifier algorithm, the values of r for which it accepts are called witnesses, or certificates. Thus, $\mathbf{RP} \subseteq \mathbf{NP}$.

An **NP** language L is also in **RP** if it has some verifier algorithm with the property that if x has a witness then it has many (at least half of all potential ones).

Definition A language L is in $\mathbf{ZP}(t(n))$ if there is a Las Vegas algorithm working in time $t(n)$ deciding $x \in L$ in expected time $O(t(n))$.
$\mathbf{ZPP} = \bigcup_k \mathbf{ZP}(n^k)$.

Example? Quicksort is a Las Vegas algorithm, but sorting is also in $\mathbf{P}$.
It is not easy to find a nontrivial example of a $\mathbf{ZPP}$ language.
Adleman and Huang have shown that prime testing is in $\mathbf{ZPP}$, but by now, Agrawal, Kayal and Saxena showed that it is in $\mathbf{P}$, too.

Characterizing Las Vegas

The proof of the following theorem is an exercise: it is a good opportunity to practice our notions.

Theorem

a. $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$.

b. A language L is in **ZPP** if and only there is a randomized polynomial-time algorithm that either decides $x \in L$ correctly, or returns "I give up", but only with probability $\leq 1/2$.

Two-sided error

It is natural to consider a randomized complexity class with two-sided error.

Definition A language L is in $\mathbf{BP}(t(n))$ if there is a randomized polynomial-time algorithm A working within time $O(t(n))$ such that for all $x \in \Sigma^*$

- if $x \in L$ then $A(x)$ rejects with probability $\leq 1/3$.
- if $x \notin L$ then $A(x)$ accepts with probability $\leq 1/3$.

Let $\mathbf{BPP} = \bigcup_k \mathbf{BP}(n^k)$.

Similarly, we will say that $L \in \mathbf{PP}$ if there is a polynomial algorithm that fails only with probability $< 1/2$ and decides $x \in L$.

Example I do not recall a simple natural example of **BPP**.

But since **RP** is not closed under complementation, if $L_1, L_2 \in \mathbf{RP}$ then about $L_1 \setminus L_2$ we can only say that it is in **BPP**.

Theorem The definition of **BPP** does not change if we replace $2/3$ with $1/2 - \varepsilon$ for a fixed $\varepsilon > 0$, or even for n^{-k} with any fixed $k > 0$.

To get from error probability $1/2 - \varepsilon$ to error probability 2^{-n^k}, use repetition $O(n^k)$ times, majority voting and the Chernoff bound.

Why not $1/2$? The definition of **BPP** does not work with $1/2$ in place of $2/3$: in that case we get a (probably) much larger class closely related to #**P** (see later). But we could use any $1/2 + \varepsilon$ for some constant ε.

Placing BPP

- Is **BPP** $=$ **P**? Some prominent computer scientists believe so, since it is implied by the existence of certain "pseudo-random" generators.
- Recall that an **NP** language can be defined with one existential quantifier, formally **NP** $= \Sigma_1$. A language in **BPP** can be defined using 2 quantifiers: formally, **BPP** $\subseteq \Sigma_2 \cap \Pi_2$, so it is still in the polynomial hierarchy. The proof uses the technique of "universal hashing" .
- On the other hand, the whole polynomial hierarchy is contained in **PP**, so this class is (probably) much bigger than **BPP**.

Interactive proofs

- We have just seen that interaction in the proof process allows to verify much more computationally complex assertions than a single-step proof: in polynomial time, we can check any **PSPACE** predicate.
- But in the Prover/Verifier paradigm, it is more natural to restrict Verifier to only polynomial-time power.
- If Verifier is allowed only deterministic polynomial-time verification, then interaction brings nothing new compared to a single step. (This is proved by an easy argument.) To go beyond, Verifier will be allowed to randomize.

Examples

- Paul wants to prove to a color-blind friend Vera that his two socks have different color. Vera puts the socks behind her back and with probability $1/2$ switches them. Then she challenges Paul to tell whether she switched.
- Paul has two (labeled) graphs G_1, G_2, and wants to prove Vera that they are not isomorphic. Vera chooses a random $j \in \{1, 2\}$, shows Paul a randomly permuted version of G_j, and asks Paul to guess j.

If Paul is truthful, he answers Vera correctly in every attempt. If he is not, then he will miss with probability $1/2$.
(Though "Vera" translates into "Faith", our Vera requires at least some kind of proof. . . .)

Graph non-isomorphism is only known to be in **coNP**, so no plain proof is known for it. But the above example provides an interactive proof.

Formal definition of interactive proofs

Formalizing interactive computation with Verifier's coin-tossing, is straighforward. But some issues remain.

How many rounds? Polynomially many, but any constant number of rounds, for example 2, is also interesting.

One-sided version True assertion proved with probability 1. False one proved with probability $\leq 1/2$.

Two-sided version True assertion proved with probability $\geq 2/3$. False one with probability $\leq 1/3$.
We will find that the two versions have the same power.

Private-coin version Verifier does not show Prover her coin-tosses (essential in the socks/graph-nonisomorphism example).

Public-coin version Prover learns Verifier's coin-tosses. It is nontrivial that private-coin proofs can be transformed into public-coin proofs. We may have to skip that theorem here.

IP = PSPACE

Let **IP** be the set of languages provable using interactive proofs. By enumerating all possible interaction sequences and summing up probabilities, it is easy to show **IP** $\subseteq$ **PSPACE**. It was a major collaborative achievement to prove the following:

Theorem **IP = PSPACE.**

We have already given an interactive proof for graph non-isomorphism, which is in **coNP**.
We give now one for $\overline{\mathsf{3SAT}}$, also in **coNP**. The method will generalize.

Arithmetization

A Boolean formula $\phi(x_1, \ldots, x_n)$ can be seen as a polynomial $P_\phi(x_1, \ldots, x_n)$, using the following conversion:

$$a \wedge b = a \cdot b, \quad \neg a = 1 - a.$$

A 3-CNF formula with m clauses converts into a polynomial $g(x_1, \ldots, x_n)$ of degree

$$d \leq 3m.$$

We do not expand the products. It matters only that g can be evaluated fast, and has a degree bound.

$$\sum_{x_1 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} g(x_1, \ldots, x_n) = 0.$$

We will rather prove, for an arbitrary K:

$$\sum_{x_1 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} g(x_1, \ldots, x_n) = K.$$

What do we gain by arithmetization?

In parts of the proof, the prover may claim some identity of polynomials. We have seen that then cheating is caught fast if we can substitute large numbers, not just 0,1. Protocol: fix some $N \geq 2^{dn}$.

Verifier: If $n = 1$ just check $g(0) + g(1) = K$. Else:

Prover: Sends an explicit univariate degree d polynomial $s(x)$ that is equal to

$$h(x) = \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} g(x, x_2, \ldots, x_n).$$

Verifier: Reject if $s(0) + s(1) \neq K$; otherwise pick a random number $r \in \{1, \ldots, N\}$. Recursively use the same protocol to check

$$s(r) = \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} g(r, x_2, \ldots, x_n).$$

Claim The prover can get away with cheating, with probability at most nd/N.

Proof by induction on n: two cases, similarly to the case of polynomial identity check.

Quantified formulas in IP

The above idea seems generalizable to quantified formulas: $\forall x \phi(x)$ iff $P_\phi(0) \cdot P_\phi(1) \neq 0$. More generally,

$$\forall x_1 \exists x_2 \cdots \exists x_n \phi(x_1, \ldots, x_n) \Leftrightarrow$$
$$0 \neq \prod_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} P_\phi(x_1, \ldots, x_n).$$

Problem when products are used, the degree of the polynomial can grow exponentially, and the degree bound was essential.

Solution For $x^i \in \{0,1\}$, $x_i^k = x_i$. There is therefore a multilinear equivalent polynomial (linear in each variable).

For polynomial $g(x_1, \ldots, x_n)$ let

$$g' = Lx_i g = x_i g(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) + (1 - x_i) g(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n),$$

then $g'(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ for $x_1, \ldots, x_n \in \{0,1\}^n$, but g' is linear in x_i.
(Note: Lx_i does not bind x_i the way $\exists x_i$ or $\forall x_i$ do.)
In constructing the polynomial for $\forall x_1 \exists x_2 \cdots \exists x_n \phi(x_1, \ldots, x_n)$, we linearize the remaining free variables after every product operation:

$$\begin{aligned} &\forall x_1 \forall x_2 \exists x_3 \forall x_4 g(x_1, x_2, x_3, x_4) \\ \mapsto &\prod_{x_1 \in \{0,1\}} Lx_1 \prod_{x_2 \in \{0,1\}} \sum_{x_3 \in \{0,1\}} Lx_1 Lx_2 Lx_3 \prod_{x_4 \in \{0,1\}} g(x_1, x_2, x_3, x_4). \end{aligned}$$

This is the form whose value will be proved interactively.

- Operators $\mathcal{O}x \in \{Lx, \exists x, \forall x, [x = a]\}$. The substitution $[x_i = a]$ where a is some constant, replaces x_i with a. This operator does not increase the degree.
- Our formula is $g = \mathcal{O}_1 x_{i_1} \cdots \mathcal{O}_k x_{i_k} h(x_1, \ldots, x_n)$ where h is a polynomial of the variables $x_1, \ldots, x_n$, of degree d. It has no free variables.
- Push each substitution $[x_i = a]$ inside to the front of the outermost occurrence of Lx_i. If there is no such occurrence then for example in case of $i = 1$, just use $h(a, x_2, \ldots, x_n)$ in place of $h(x_1, x_2, \ldots, x_n)$.

- Prover can prove $g = b$ for any (polynomial-length) b
  with probability 1 if it is true,
  with probability $\leq \varepsilon$ if it is false.
- Let $i_1 = 1$, then we have
  $g = [x_1 = a]Lx_1 g'$ or $g = \exists x_1 g'$ or $g = \forall x_1 g'$
  for some (implicit) univariate polynomial $g' = g'(x_1)$ of degree $\leq d$. Prover provides an explicit degree d polynomial $s(x)$, claiming $s(x) = g'(x)$ for all x (not only binary).
- Assume for example $g = [x_1 = a]Lx_1 g'(x_1)$, the other cases are similar.
- Verifier checks directly $a_1 s(0) + (1 - a_1)s(1) = b$. Then picks random $r \in \{1, \ldots, N\}$ and asks (recursively) a proof for $s(r) = [x_1 = r]g'$.
  The recursion works since g' has fewer operators.

Failure probability if Prover lies: Verifier may pick a number r that swallows the lie, with probability $\leq d/N$. Otherwise, Prover may prove a false equality $s(r) = [x_1 = r]g'$ with probability $\leq \varepsilon$.

So we can handle every additional operator $\mathfrak{O}x_i$ at the expense of another d/N term in the failure probability.
Seen from inside out, the (per variable) degree d first decreases to 1 (Lx_i operators), and from there on it will never be more than 2.

Note The prover in this protocol uses public coins. But it is still not trivial to show that for any (constant, or even polynomial-size) k, a k-round interactive proof implies a $(k+2)$-round one with public coins.

Constraint satisfaction

Generalize CNFs as follows. Constraints on a vector of n of binary variables $\mathbf{u} = (u_1, \ldots, u_n)$. Instead of just disjunctive clauses, more general constraints, depending on any q-tuple $\sigma = (i_1, \ldots, i_q)$ of spots. Write

$$\mathbf{u}_\sigma = (u_{i_1}, \ldots, u_{i_q}).$$

A qCSP instance is a set of constraints $\phi = \{\phi_1, \ldots, \phi_m\}$, where $\phi_i = (\sigma_i, f_i)$, and each $f_i : \{0,1\}^q \to \{0,1\}$ is a Boolean function. An assignment $\mathbf{u} \in \{0,1\}^n$ satisfies the constraint ϕ_i if $f_i(u_{\sigma_i}) = 1$. Let

$$\mathsf{val}(\phi)$$

be the largest possible fraction of constraints that can be satisfied, over all possible assignments $\mathbf{u} \in \{0,1\}^n$.

The ρ-GAPqCSP is a problem to determine, for a given instance ϕ of qCSP, which of the following cases holds:

1. $\mathsf{val}(\phi) = 1$
2. $\mathsf{val}(\phi) < \rho$.

Examples MAX3SAT, MAXCUT, with appropriate ρ.

This is a promise problem: we are promised that the input falls into one of the two classes—at least are not obliged to answer correctly when it does not.

Gap-producing reduction

Theorem (PCP theorem, constraint satisfaction view) There are constants ρ, q such that ρ-GAPqCSP is **NP**-hard: for every **NP** language L there is a polynomial translation τ with:

Completeness: $x \in L$ implies $\mathsf{val}(\tau(x)) = 1$.

Soundness: $x \notin L$ implies $\mathsf{val}(\tau(x)) < \rho$.

This implies that even approximating $\mathsf{val}(\phi)$ within ρ is **NP**-hard. But is a little stronger, since it shows a particular form of reduction.

Spot-checkable proofs

We will recast **NP**-hard constraint satisfaction into another language, with a new version of interactive proof called probabilistically checkable proofs (I will call them spot-checkable proofs).

Definition An (r,q)-**PCP** verifier $V^{\pi}(x,i)$: The bit string π of length $2^r q$ that V checks is called a proof (for $x \in L$, corresponding to the assignment $\mathbf{u}$ in the CSP above), if the following holds. There is a set of constraints $\phi_i = (\sigma_i, f_i)$ of size $\leq q$ with $i \leq 2^r$. To some input x and random bit string i, of length r seen as an index, verifier V computes in polynomial time the constraint ϕ_i and returns

$$V^{\pi}(x,i) = f_i(\pi_{\sigma_i}).$$

Let L be a language. For verifying $x \in L$ we require:

Completeness: If $x \in L$ then there is a proof π with
$\mathsf{P}\{V^{\pi}(x,i) = 1\} = 1$.

Soundness: If $x \notin L$ then for all proofs π we have
$\mathsf{P}\{V^{\pi}(x,i) = 1\} \leq 1/2$.

If there is a $(c \cdot r(n), d \cdot q(n))$-**PCP**-verifier for L then we say $L \in \mathbf{PCP}(r(n), q(n))$.

Theorem (PCP theorem, proof verification view)

$\mathbf{NP} = \mathbf{PCP}(\log n, 1)$.

Equivalence of the two views.

MAX3SAT is hard to approximate

Theorem There is a constant $\rho < 1$ such that MAX3SAT cannot be approximated to within ρ unless $\mathbf{P} = \mathbf{NP}$.

Proof.

1. Turn each of the m constraints into $\leq 2^q$ clauses of a qCNF.
2. Use the following identity: $A \vee B \iff \exists z((\neg z \vee A) \wedge (z \vee B))$. Apply it repeatedly to disjunctions $A \vee B$ of size > 3 with A of size 2, turning any clause with $q > 3$ disjuncts to at most $q - 1$ clauses of ≤ 3 disjuncts each.
3. Now we have a 3-CNF with $\leq q2^q$ constraints. If ε fraction of the original constraints is unsatisfied then so is $\frac{\varepsilon}{q2^q}$ fraction of the new ones.

□

From MAX3SAT to independent sets

- Take the usual reduction from 3SAT to independent sets: Each occurrence of each literal is a point. Points are connected if they are in the same clause or are negations of each other. k satisfied clauses give k independent points and vice versa.
- We just showed that approximating the size of maximum independent set to some factor $1-\varepsilon$ is **NP**-hard.
  Let us increase the gap to any constant positive factor.
  For graph G of size n, let $G^k = (V^k, E_k)$, where $(u_1, \ldots, u_k)$ is adjacent to $(v_1, \ldots, v_k)$ iff there is an i where u_i is adjacent to v_i. If the maximum independent set size of G is r, then that of G^k is r^k. Relative sizes: $\frac{r}{n} \to \left(\frac{r}{n}\right)^k$.

Towards the proof of the PCP theorem

To illustrate the techniques, we will prove:

Theorem $\mathbf{NP} \in \mathbf{PCP}(\text{poly}(n), 1)$.

This allows poly(n) random bits— and with it, proofs of size $2^{\text{poly}(n)}$—but still only a constant number of spot checks. Still not trivial.

Difficulty: how to catch a prover when he leaves a gap only in one spot of a long proof?

Solution idea: Encode the proof using some error-correcting code, so that even one error of the original proof changes the codeword in a large fraction of places.

Spot checking: We need not only that the cheating changes a large fraction of the places, but that it is detectable by spot checks. Our error-correcting code achieving this is very redundant.

Walsh-Hadamard code

Denote the inner product of n-bit vectors $\mathbf{x}, \mathbf{y}$ by $\mathbf{x} \odot \mathbf{y} = \sum_{i=1}^{n} x_i y_i$.

Definition The Walsh-Hadamard code $\mathsf{WH}(\mathbf{u})$ of an n-bit word $\mathbf{u}$ is a 2^n-bit word representing a linear Boolean function:

$$f_{\mathbf{u}} = \mathsf{WH}(\mathbf{u}), \quad f_{\mathbf{u}}(\mathbf{x}) = \mathbf{u} \odot \mathbf{x}.$$

Example $\mathsf{WH}(0010) = 0011001100110011$, where we listed the values of $0011 \odot \mathbf{x}$ for $\mathbf{x} = 0000, 0001, \ldots, 1111$.

(I do not know why "Walsh-Hadamard": both Hadamard and Walsh lived before error-correcting codes.)
Clearly, every linear function f is a Walsh-Hadamard code $f_{\mathbf{u}}$ for some $\mathbf{u}$.

Linearity check

A linear function is locally decodable: If g is close to linear function f, then for all $\mathbf{x}$, $f(\mathbf{x})$ can be restored (with large probability) from a few spot checks of g.
This relies on the following theorem showing that if a function passes linearity check with high probability then it is almost linear.

Theorem Let $f : \{0,1\}^n \to \{0,1\}$ be such that

$$\mathsf{P}_{\mathbf{x},\mathbf{y}\in\{0,1\}^n}\{f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})\} \geq \rho > 1/2.$$

Then f is ρ-close to some linear function.

We accept this on faith, it is proved later in the book (using Fourier transforms).

Local decoding

Let us use the theorem for local decoding. Suppose that g is $(1-\delta)$-close to a linear function f. Method to find $f(\mathbf{x})$ for any $\mathbf{x}$:

- Choose random vector $\mathbf{u}$.
- Output $\mathbf{y} = g(\mathbf{x}+\mathbf{u}) - g(\mathbf{u})$.

With the help of the theorem, it is an exercise to show $\mathsf{P}\{\mathbf{y} = f(\mathbf{x})\} \geq 1-2\rho$.

Quadratic equations

For showing **NP** $\subseteq$ **PCP**(poly(n), 1), start from an **NP**-complete problem of algebraic nature: QUADEQ. This asks for the solvability of a set of quadratic equations modulo 2.

Proposition QUADEQ is **NP**-complete.

Proof. Recall the reduction of circuit satisfiability to 3SAT. We obtained first a set of constraints of the form $z = \neg x$, $z = x \vee y$, $z = x \wedge y$. Now, for example $z = x \vee y$ can be translated into

$$z \equiv 1 - (1 - x)(1 - y) \pmod 2.$$

□

Example (all operations understood modulo 2):

$$\begin{aligned} u_1u_2 + u_1u_5 + u_3u_4 &= 1 \\ u_1u_4 + u_2u_2 &= 0 \\ u_1u_4 + u_3u_4 + u_3u_5 &= 1 \end{aligned}$$

In general:

$$\sum_{j,k=1}^{n} a_{i,jk}u_ju_k = b_i, \quad i = 1, \ldots, m.$$

Let $\mathbf{A}$ be the $m \times n^2$ matrix $(a_{i,jk})$, and $\mathbf{u} \otimes \mathbf{u}$ the tensor product (a vector with elements u_ju_k). Then this can be written as $\mathbf{A}(\mathbf{u} \otimes \mathbf{u}) = \mathbf{b}$.
From a solution (witness) $\mathbf{u}$, the **PCP** proof is the pair (f, g) with

$$f = \mathsf{WH}(\mathbf{u}), \quad g = \mathsf{WH}(\mathbf{u} \otimes \mathbf{u}).$$

Checking the long proof

Each random check to be repeated some constant number of times.

1. Check the linearity of f, g. From now on, under f, g, we will actually mean $\tilde{f}(\mathbf{x}), \tilde{g}(\mathbf{x})$ obtained by local decoding.
2. Verify $g = \mathsf{WH}(\mathbf{u} \otimes \mathbf{u})$ assuming $f = \mathsf{WH}(\mathbf{u})$: check $g(\mathbf{r} \otimes \mathbf{r}') = f(\mathbf{r}) f(\mathbf{r}')$ for random vectors $\mathbf{r}, \mathbf{r}'$.
3. Check $\mathbf{A}(\mathbf{u} \otimes \mathbf{u}) = \mathbf{b}$ as follows.
   With $\mathbf{z}_i = (a_{i,jk})$, notice $(\mathbf{A}(\mathbf{u} \otimes \mathbf{u}))_i = g(\mathbf{z}_i)$. Checking $g(\mathbf{z}_i) = b_i$ for all i would be too many checks, so we combine them into one, with random coefficients.
   With columns $\mathbf{a}_{11}, \ldots, \mathbf{a}_{nn}$ of $\mathbf{A}$, let $\mathbf{v} \odot \mathbf{A}$ be the vector with coordinates $\mathbf{v} \odot \mathbf{a}_{jk}$, $j, k = 1, \ldots, n$.
   Check $g(\mathbf{r} \odot \mathbf{A}) = \mathbf{r} \odot \mathbf{b}$ for a random vector $\mathbf{r}$.

Simple estimates show that if the proof is bad then with probability at least $1/2$, one of these checks fails.

How about constraints of size just 2?
Given a graph $G = (V, E)$ and a partition $V = S \cup T$, $S \cap T = \emptyset$, the set of edges in the cut is denoted

$$E(S, T) = \{ (u, v) \in E : u \in S, v \in T \}.$$

Example The MAX-CUT problem: given graph $G = (V, E)$, find the cut (S, T) with the largest possible $|E(S, T)|$. This is a 2CSP problem: on each edge is a constraint saying that the ends should have different values.
MAX-CUT is known to be **NP**-complete. Inapproximability is harder.

Here, we will show that the constraints can be made binary, but we make it at the expense of increasing the alphabet size from 2 to some constant value W.

Definition $q\mathsf{CSP}_W$ is the q-ary constraint satisfaction problem: we have n variables $x_1, \ldots, x_n$ and m q-ary constraints as before, but the variables can take values from $\{1, \ldots, W\}$, for a constant W.

Example (Approximate 3-coloring) Given a graph G, assign 3 colors to its vertices in such a way that maximizes the number edges whose endpoints have different colors.
This problem is in $2\mathsf{CSP}_3$.

More generally, a $2\mathsf{CSP}_W$ has a constraint graph: its edges are the pairs that participate in some constraint.

Proposition For every q there is a polynomial algorithm τ translating every $q\mathsf{CSP}$ instance ϕ on n variables with m constraints to an instance $\psi = \tau(\phi)$ of $2\mathsf{CSP}_{2^q}$ on $m+n$ variables, with the following properties.

- If ϕ is satisfiable then ψ is also satisfiable.
- If $\mathsf{val}(\phi) \leq 1-\varepsilon$ then $\mathsf{val}(\psi) \leq 1-\varepsilon/q$.

Construction. If $\phi(u_1,\ldots,u_n) = \phi_1 \wedge \cdots \wedge \phi_m$, introduce for each clause ϕ_i an extra variable $y_i = (z_{i1},\ldots,z_{iq}) \in \{0,1\}^q$ (viewed as q binary variables).
ψ has qm constraints. Suppose u_4 occurs in ϕ_i: say $\phi_i = u_3 \vee \neg u_4 \vee u_7$ (case $q=3$). Then $\psi_{i,4}$ says

$$(z_{i1} \vee \neg z_{i2} \vee z_{i3}) \wedge (u_4 \Leftrightarrow z_{i2}).$$

This construction gives a constraint graph with unbounded degree (possibly m).

Expander graphs

Sometimes a constraint graph with bounded degree is desirable. The method to find hard problems with such a graph uses expanders, an important tool in theoretical computer science. An expander is a graph $G = (V, E)$ with a constant degree d that behaves in some important respects as a random graph. Let $S \subset V$. If the graph is random with degree d, then we expect $E(S, T) \approx d|S||T|/|V|$.

Definition For a $\lambda \leq 1$, we will call a graph a λ-expander if for all cuts (S, T):

$$|E(S, T)| \geq (1 - \lambda) d |S| \frac{|T|}{|V|}.$$

It is not hard to prove by counting argument that expander graphs of arbitrarily large size exist for all constants λ, and d.

What makes expanders interesting is that some of them can be efficiently constructed (even though they behave somewhat like random graphs).

Definition Let us fix degree d and $\lambda < 1$. We say that an algorithm defines an explicit expander family with parameters (d, λ) if for all n it computes in time polynomial in n a d-regular graph G_n that is a λ-expander.

Theorem There is a constant d and an explicit expander family with parameters $(d, 0.9)$.

We absolutely have to take this now on faith.

Let $\phi = \phi_1 \wedge \cdots \wedge \phi_m$ be a $2\mathsf{CSP}_W$ instance, on variables $u_1, \ldots, u_n$. We will construct from this another set of constraints, with a bounded degree constraint graph. Suppose variable u_j appears in k constraints.

- New variables $y_j^1, \ldots, y_j^k$: one for each occurrence of u_j in some constraint ϕ_i. In old constraint ϕ_i, change u_j to the corresponding new variable y_j^s.
- We need new constraints trying to enforce $y_j^1 = \cdots = y_j^k$. Adding just the constraints $y_j^1 = y_j^2$, …, $y_j^{k-1} = y_j^k$ is not sufficient, since when for example $y_j^1 \neq y_j^2 = \cdots = y_j^k$, only one constraint is violated (the reduction may not be gap-preserving).
- Idea: constraints $y_j^s = y_j^t$ along edges (s,t) of a "random-like" graph. Get expander graph G_k on $\{1, \ldots, k\}$.

This defines a new constraint set $\psi = \psi_1 \wedge \cdots \wedge \psi_{m'}$, with $m' \leq m(d+1)$. The graph of ψ has degree $\leq d+1$.

It is obvious that if ϕ is satisfiable then ψ is also.

Claim If $\mathsf{val}(\phi) \leq 1-\varepsilon$ then $\mathsf{val}(\psi) \leq 1-\varepsilon/c$ with $c \leq \max\{2(d+1), 20W\}$.

For the proof, consider an assignment to all variables y_j^s. We assign to u_j the plurality value of $y_j^1, \ldots, y_j^k$. Let t_j be the number of y_j^s that disagree with the plurality: $t_j/k \leq 1-1/W$.
Simple case: assume $\sum_j t_j < m\varepsilon/2$. Then at least $m\varepsilon/2$ violated constraints of ϕ have unchanged value in ψ, and thus are also violated.
Interesting case: $\sum_j t_j \geq m\varepsilon/2$. By the expansion property, in graph G_k these t_i non-plurality variables have at least $(1-\lambda)dt_j(1-t_j/k) \geq 0.1dt_j/W$ edges to the plurality variables (as $\lambda \leq 0.9$).
So $0.1\sum_j dt_j/W \geq \frac{md\varepsilon}{20W}$ equality constraints are violated.

Approximating Vertex Cover and Steiner Tree

(From Vijay Vazirani's book.)

Example (Vertex cover) Given an undirected connected graph $G = (V, E)$, and cost function on vertices $c : V \to \mathbb{Q}^+$, find a minimum cost vertex cover, that is a set $V' \subseteq V$ such that every edge has at least one endpoint incident at V'.
Special case, in which all vertices are of unit cost: cardinality vertex cover problem.

An NP-optimization problem Π consists of:

- A set of valid instances D_Π, decidable in polynomial time.
- Each instance $I \in \mathbf{D}_\Pi$ has a nonempty polynomial-time decidable set $S_\Pi(I)$ of feasible solutions, of length polynomially bounded in $|I|$.
- A polynomial time computable objective function, obj_Π assigning a nonnegative number to each pair (I, s), where I is an instance and s is a feasible solution for I.
- It is said whether this is a minimization or maximization problem.

$\text{OPT}_\Pi(I) =$ the optimum for instance I.

Gap-preserving reductions

There are several versions of this, depending on what is to minimize and what to maximize. It would be better to call "gap-translating", since the reduction may change the gap size.

Definition A gap-preserving reduction Γ from a minimization problem Π_1 to maximization problem Π_2 comes with four functions: f_1, α, f_2, and β.
For a given instance x of Π_1, it computes in poly-time an instance y of Π_2 such that

1. If $\mathrm{OPT}(x) \leq f_1(x)$, then $\mathrm{OPT}(y) \geq f_2(y)$.
2. If $\mathrm{OPT}(x) > \alpha(|x|) f_1(x)$, then $\mathrm{OPT}(y) < \beta(|y|) f_2(y)$.

Since Π_1 is minimization and Π_2 is maximization, $\alpha(|x|) \geq 1$ and $\beta(|y|) \leq 1$.

Hardness of Vertex cover problem

To establish hardness of Vertex cover, we will give a gap-preserving reduction from MAX3SAT.
For a fixed k, let MAX3SAT(k) be the restriction of MAX3SAT to the instances in which each variable occurs at most k times.
For $d \geq 1$, let $VC(d)$ denote the restriction of the cardinality vertex cover problem to instances in which each vertex has degree at most d.

Theorem There is a gap preserving reduction from MAX3SAT(29) to $VC(30)$ that transforms a Boolean formula φ to a graph $G = (V, E)$, such that:

- **a** if $\mathrm{OPT}(\varphi) = m$, then $\mathrm{OPT}(G) \leq \frac{2}{3}|V|$,
- **b** if $\mathrm{OPT}(\varphi) < (1 - \varepsilon_b)m$, then $\mathrm{OPT}(G) > \frac{2}{3}(1 + \varepsilon_v)|V|$,

where m is the number of clauses in φ, ε_b is the constant from the gap of MAX3SAT(29), and $\varepsilon_v = \varepsilon_b/2$.

Proof of the theorem

Can assume that each clause has exactly 3 literals: repeat as necessary.
$\varphi \mapsto$ graph G: each literal of each clause $\mapsto$ a node.
Edges:

- Connect literals within each clause.
- Connect each literal with its negations.

G is an instance of $VC(30)$, since each vertex has two edges of the first type and at most 28 edges of the second type.
Size of a maximum independent set is exactly $OPT(\varphi)$: proof as in the **NP**-completeness reduction.
Complement of a maximum independent set in G is a minimum vertex cover, so:

1. $\text{OPT}(\varphi) = m \Rightarrow \text{OPT}(G) = 2m = \frac{2}{3}|V|$.
2. $\text{OPT}(\varphi) < (1-\varepsilon_b)m \Rightarrow \text{OPT}(G) > \frac{2}{3}(2+\varepsilon_b)m$, thus $\text{OPT}(G) > \frac{2}{3}(1+\varepsilon_v)|V|$, for $\varepsilon_v = \epsilon_b/2$.

Example (Steiner tree problem) (R, S, cost). Here R, S are disjoint sets of required and Steiner nodes.
$\text{cost} : R \cup S \to \mathbb{Q}^+$ is a metric (triangle inequality). (The graph is the complete graph on $R \cup S$ with value $\text{cost}(u, v)$ on each edge (u, v).)
Find a minimum cost tree in G (cost is the sum of the metric value of edges) that contains R.

Theorem There is a gap preserving reduction from $VC(30)$ to the Steiner tree problem transforming an instance $G = (V, E)$ of $VC(30)$ to an instance $H = (R, S, c)$ of Steiner tree, and satisfies:

- **a** if $\mathrm{OPT}(G) \leq (2/3)|V|$, then $\mathrm{OPT}(H) \leq |R| + (2/3)|S| - 1$
- **b** if $\mathrm{OPT}(G) > (1 + \varepsilon_v)(2/3)|V|$, then $\mathrm{OPT}(H) > (1 + \varepsilon_s)(|R| + (2/3)|S| - 1)$,

where $\varepsilon = 4\varepsilon_v/97$, and ε_v corresponds to the gap of the $VC(30)$ established earlier.

Proof

From an instance $G = (V, E)$, construct an instance $H = (R, S, \text{cost})$ of the Steiner tree problem such that H has a Steiner tree of cost $|R| + c - 1$ iff G has a vertex cover of size c.
Edge $e \in E \mapsto$ a corresponding node $r_e \in R$.
Vertex $v \in V \mapsto$ corresponding node $s_v \in S$.

$$\begin{cases} \text{cost}(a, b) = 1 & \text{if } a, b \in S, \\ \text{cost}(a, b) = 2 & \text{if } a, b \in R, \\ \text{cost}(r_e, s_v) = 1 & \text{if } e \text{ is incident to } v, \\ \text{cost}(r_e, s_v) = 2 & \text{otherwise.} \end{cases}$$

For a vertex cover C of size c, let $S_C = \{ s_v : v \in C \}$. There is a tree spanning $R \cup S_C$ using edges of cost 1 only: cost $R + c - 1$.
For the other direction, take a a Steiner tree T of cost $R + c - 1$.

Claim (to be proved below) We can transform T to a Steiner tree T' of the same cost that uses only edges of cost 1.

Now, T' must have exactly c Steiner vertices, with all required nodes having an edge of cost one to these. So these Steiner vertices form a vertex cover.

Proof of the claim

- First we make sure that all edges (u, v) of cost 2 are between points of R, by repeating the following: If $u \in S$, remove (u, v) from T and obtain two components of T. Put an edge from v to some $x \in R$ to connect these components.
- Now repeat the following: consider an edge (r_e, r_f) in R. Removing (r_e, r_f) from T we obtain two components, with sets of required nodes R_1, R_2: $r_e \in R_1$ and $r_f \in R_2$.
  Since G is connected, there is a path π in G from an endpoint of e to an endpoint of f. There are edges (a, b) and (b, c) in π with $r_{(a,b)} \in R_1$ and $r_{(b,c)} \in R_2$. Add these edges, of cost 1, to replace (r_e, r_f).

Summing up:
If $\mathrm{OPT}(G) \leq (2/3)|V|$, then $\mathrm{OPT}(H) \leq |R| + (2/3)|S| - 1$.
If $\mathrm{OPT}(G) > (2/3)(1+\varepsilon_v)|V|$, then
$\mathrm{OPT}(H) > |R| + (2/3)(1+\varepsilon_v)|S| - 1$..

Historical remark

Gap-preserving reductions were introduced by Papdimitriou and Yannakakis: they have given such reductions between 17 problems, about two years before the PCP theorem.

Average-case complexity

Two distinct ways to introduce probability theory into algorithmic analysis:

Randomization

Average case analysis instead of worst-case analysis. Due to the difficulties of meaningful results about the average case, we postponed it—now we return.

Natural question: when to call an algorithm polynomial-time on average? The corresponding answers in case of randomization (resulting in classes **RP**, **ZPP**, **BPP**) were robust: did not depend much on machine models, or the constants like 1/2, 2/3 used. The reason was that an experiment could be repeated, using new random numbers. The result can be different, even with the same input.

Repetition is not an option in average-case complexity: if a bad input causes the algorithm to run long, repetition still gets the same input!

Examples

- Random graphs, with the usual distributions, are generally not good examples: many difficult problems are easily solved for them.
- But some manipulation helps: finding large clique in a random graph with a planted clique of size $n^{1/4}$, seems difficult.
- Closest solution of a random set of linear equations modulo 2. (Decoding a random linear code.)
  Note, for real numbers: l_2 (least squares) closest solution has formula. l_1 closest solution can be found by linear programming.

Distributions

There is no reason to consider just the uniform distribution. In general, in an algorithmic problem for which the average case can be discussed, we are given both a function $f(x)$ to compute, and a probability distribution $\mathcal{D}$ on the set of inputs: a distributional problem is given by the pair $(f, \mathcal{D})$. Simplification:

- Only decision problems: deciding some language L.
- Instead of one distribution over all strings, consider for each n a different distribution $\mathcal{D}_n$, on inputs of length n. We still write $(L, \mathcal{D})$.

Warning example

Let A be an algorithm on a 2-tape Turing machine whose running time $t_1(x)$ on strings x of length n behaves as follows:

$$t_A(x) = \begin{cases} 2^{-n} & \text{if } x = 1^n, \\ n & \text{otherwise.} \end{cases}$$

Let X be a random variable over $\{0,1\}^n$, with the uniform distribution, then

$$\mathsf{E}\, t_1(X) = 2^n \cdot 2^{-n} + n \cdot (1 - 2^{-n}) < n + 1,$$

so this seems a polynomial algorithm. But we may have to simulate this algorithm on a 1-tape Turing machine, resulting in a running time about which we only know that it is less than $(t_A(x))^2$. But then for the expected time we only know

$$\mathsf{E}\,(t_A(X))^2 \leq 2^{2n} \cdot 2^{-n} + n^2 \cdot (1 - 2^{-n}) = 2^n + n^2(1 - 2^{-n}),$$

not polynomial anymore.

Levin's tricky solution is to take roots before averaging:

Definition Running time t_A is polynomial on average, distribution $\mathcal{D}$, if there is an $\varepsilon > 0$ such that

$$\mathsf{E}\,\frac{(t_A(X))^\varepsilon}{n} = O(1).$$

A distributional problem $(L, \mathcal{D})$ is polynomial on average, is in **distP**, if it has a decision algorithm with running time polynomial on average.

You can check that this property is robust with respect to taking a power.

Which distributions?

- Need to restrict the class of distributions considered: otherwise we are back to worst-case analysis.
- A reasonable class of distributions $\mathcal{D}$: let

$$\mu_{\mathcal{D}_n}(x) = \sum_{y \in \{0,1\}^n : y \leq x} \mathsf{P}(x),$$

where $y \leq x$ is in the lexicographic sense, be the cumulative distribution function of $\mathcal{D}$.
We require $\mu_{\mathcal{D}_n}(x)$ to be computable in polynomial time (approximation within 2^{-k} in time polynomial in $n+k$). Such distributions will be called **P**-computable.

- Why not require just $\mathsf{P}(x)$ to be computable? This seems too weak for the results that come. It does not imply **P**-computability.
- A weaker property that works just as well: a distribution is called **P**-samplable, if it is the distribution of the output of a randomized polynomial-time computation.
  P-computable implies **P**-samplable (exercise), but the converse is not believed to be true.

Note Two distinct new definitions of polynomiality, both needed:

- What is an expected polynomial-time algorithm?
- What is a polynomial-time distribution?

The average-case version of **NP**:

Definition $(L, \mathcal{D})$ is in **distNP**, if $L \in \mathbf{NP}$ and $\mathcal{D}$ is **P**-computable.

Definition (Average-case reduction) $(L, \mathcal{D}) \leq_p (L', \mathcal{D}')$, if there is a polynomial-time computable τ, and polynomials p, q with

Completeness $x \in L \Leftrightarrow \tau(x) \in L'$

Length regularity $|\tau(x)| = p(|x|)$

Domination For all n, all $y \in \{0,1\}^{p(n)}$,

$$\mathsf{P}\{\, y = \tau(\mathcal{D}_n)\,\} \leq q(n)\,\mathsf{P}\left\{\, y = \mathcal{D}'_{p(n)}\right\}.$$

Here, $\mathcal{D}_n$ denotes also a random variable with distribution $\mathcal{D}_n$.

Length regularity makes sure polynomial-time has the same meaning in the input and output.
Domination makes sure that probable instances of $(L, \mathcal{D})$ do not go into improbable instances of $(L', \mathcal{D}')$: needed to assure that an expected polynomial algorithm for $(L', \mathcal{D}')$ implies one for $(L, \mathcal{D})$.

Theorem If $(L, \mathcal{D}) \leq_p (L', \mathcal{D}')$, and $(L', \mathcal{D}') \in \textsf{distP}$, then $(L, \mathcal{D}) \in \textsf{distP}$.

The proof is straightforward, but uses all the required properties of reduction.

A complete problem

Here is an **NP** language U that is simplest to prove complete:

$$(\langle V \rangle, x, 1^t) \in U$$

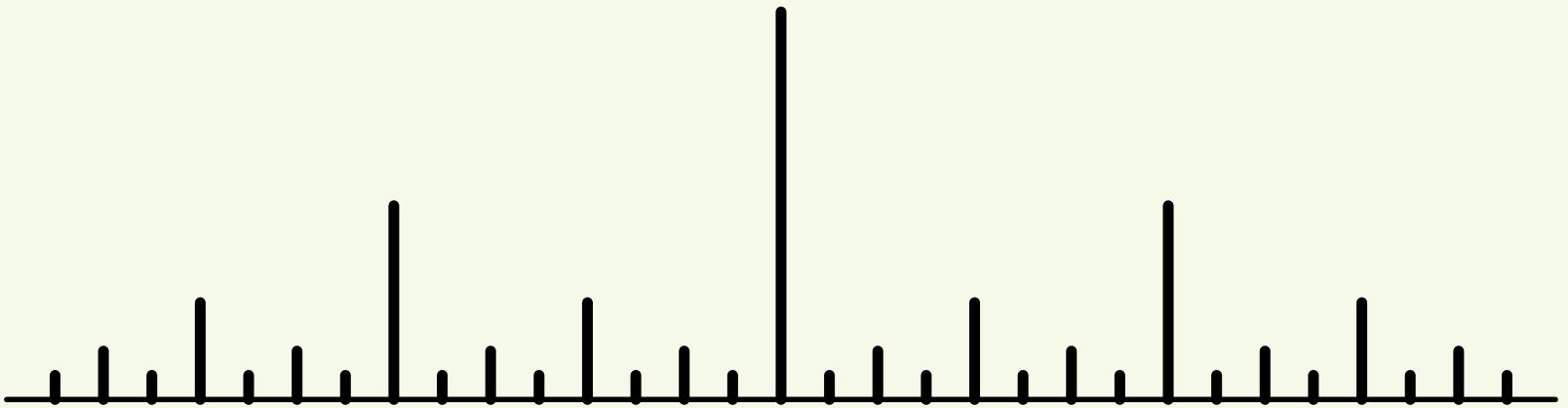
if $\langle V \rangle$ describes a verifying Turing machine, and there is a witness w such that $V(x, w)$ accepts in t steps. (From now on, we write V in place of $\langle V \rangle$.)
This is clearly complete: for every **NP** language L with verification function V and polynomial time bound $p(n)$, the translation $x \mapsto (V, x, 1^{p(|x|)})$ reduces L to U.

To define $\mathcal{U}_n$ for a **distNP**-complete distributional problem $(U, \mathcal{U})$, choose randomly an n-bit string defining $(V, x, 1^n)$ as follows:

- Choose V uniformly among the first n machine descriptions: $|V| = \lceil \log n \rceil$.
- Choose $|x|$ uniformly from $\{0, \ldots, n - |V| - 1\}$.
- Choose each bit of x uniformly, then pad it with 10^t to length n.

This distribution is not uniform, but is is separately uniform for each grain size. Ignoring multiplicative factors,

$$\mathsf{P}(Vx10^t) \sim 2^{-|x|}, \quad |x| \in \{0, 1, \ldots, n - |V| - 1\}.$$



Non-uniform distributions

Reduction to $(U, \mathcal{U})$ uses a probability theory idea. In randomization, how to produce an arbitrarily distributed real variable X, with cumulative distribution function $F(y) = \mathsf{P}\{X < y\}$? As shown, producing uniform distribution is sufficient:

Proposition Suppose that $F : (-\infty, \infty) \to [0, 1]$ is a monotonic continuous function.
The variable $F(X)$ is uniformly distributed over $[0, 1]$.
Conversely, if Z is uniformly distributed over $[0, 1]$ then $F^{-1}(Z)$ is distributed like X.

The proof is immediate.

Example Let X have the exponential distribution with parameter λ: $\mathsf{P}\{X < a\} = 1 - e^{-\lambda a}$ for $a \geq 0$, and 0 for $a \leq 0$.
Then $1 - e^{-\lambda X}$ is uniform. On the other hand, if Z is uniform, then $-\ln(1 - Z)/\lambda$ is exponential with parameter λ.

Proof of the completeness theorem

We want to reduce the **distNP** problem $(L, \mathcal{D})$, where L is given by the pair $(V, p(\cdot))$: here V is a verification function and $p(\cdot)$ a polynomial bound. For each $|x| = n$ we have $x \in L$ if there is a witness w such that $V(x, w)$ accepts in time $p(n)$. For a random variable X distributed according to $\mathcal{D}_n$, let $F(x) = \mathsf{P}\{X < x\}$. Consider an instance x of $(L, \mathcal{D})$, with $|x| = n$. We define an instance $(V', y, 0^t)$ of $(U, \mathcal{U})$.

Approximate idea Define V' to have $V'(y, w) = V(F^{-1}(y), w)$. Let $y = F(x)$, $t = p(n)$.

This is using the original reduction, plus translation of an arbitrary (continuous) distribution to the uniform one.

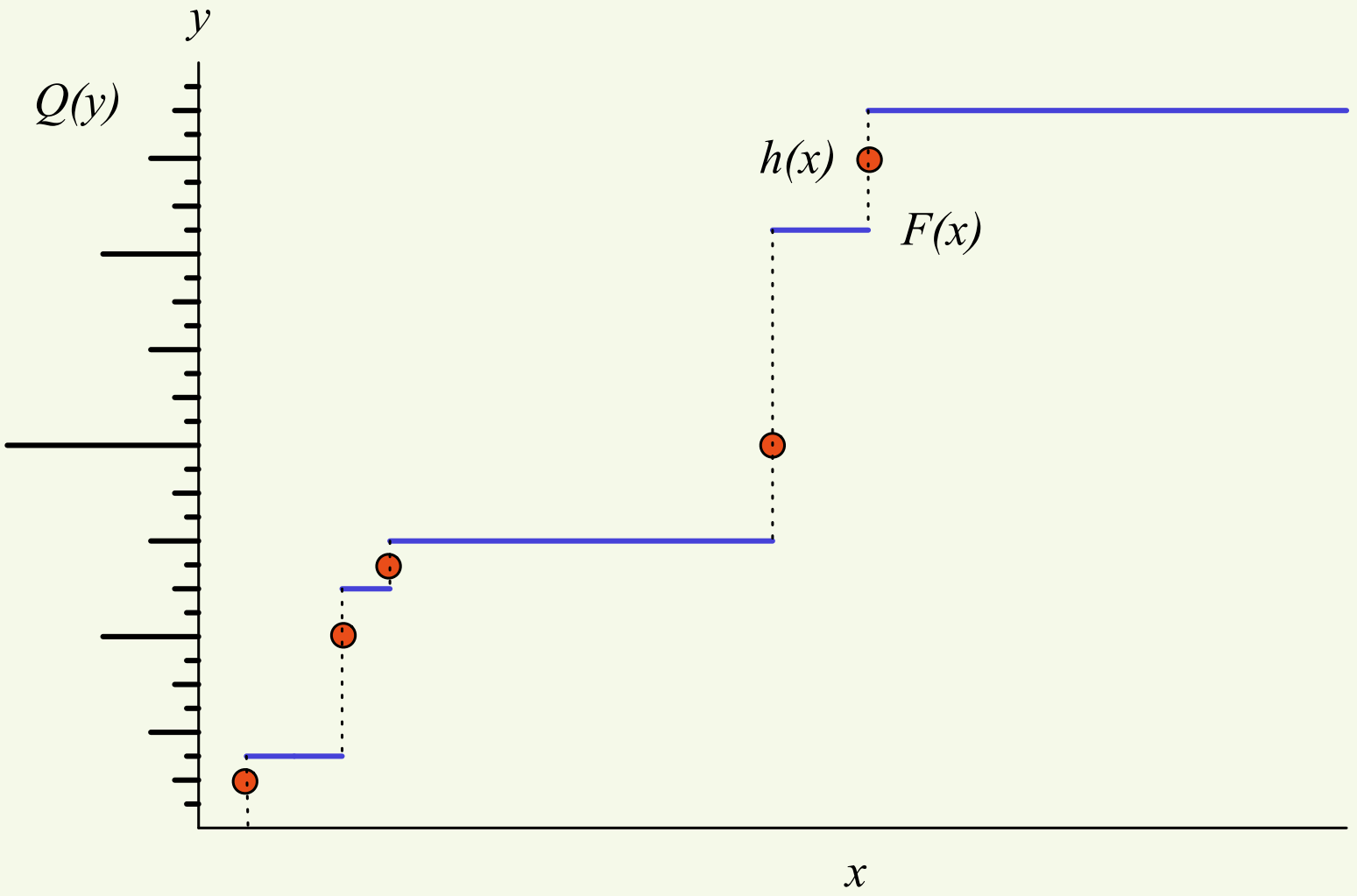
Difficulty: The distribution $\mathcal{D}_n$ is not continuous, and $\mathcal{U}_n$ is not uniform.

We will try with an approximate version, which assigns strings y of large probability in $\mathcal{U}$ to strings x of large probability in $\mathcal{D}$.

For $x \in \{0,1\}^n$, let

$$g(x) = \begin{cases} 0x1 & \text{if } F(x+1) - F(x) < 2^{-n}, \\ 1z10^{n-|z|} & \text{if } 0.z \text{ is the shortest binary fraction in } [F(x), F(x+1) \end{cases}$$

The function $g : \{0,1\}^n \to \{0,1\}^{n+2}$ is one-to-one.



The bars on the left show $Q(y) = \mathsf{P}_{\mathcal{U}}(y)$. The red dots show $h(x) = 0.z$ determining $g(x)$, in the cases where $\mathsf{P}_{\mathcal{D}}(x) = F(x+1) - F(x) \geq 2^{-n}$. As seen, $\mathsf{P}_{\mathcal{U}}(g(x))$ is roughly proportional to $\mathsf{P}_{\mathcal{D}}(x)$.

Second attempt: $V'(y,w) = V(g^{-1}(y),w)$.
Difficulty: is $g(\cdot)$ invertible in polynomial time? Maybe, but a trick takes away this worry. Defining V', just add another witness that does the inversion:

$$V'(y,u,w) = \begin{cases} 0 & \text{if } g(u) \neq y, \\ V(u,w) & \text{otherwise.} \end{cases}$$

The final translation is $\tau(y) = (V', g(y), 1^{p(n)})$.

Communication complexity

A new kind of computation problem and complexity. Sometimes, conclusions about other kinds of complexity, but the results are important in their own right, too.
We will rely more on the Lovász notes than on the Arora-Barak book.

Participants: Alice sees input x, Bob sees input y, both $\in \{0,1\}^n$.

Goal: find bit $f(x,y)$, so that at the end, both know it.
Example: $f(x,y)$ shows whether $x = y$.

Method: Communication protocol, sending bits to each other.
Bits already sent always determine whose turn it is.

Cost: The total number of bits sent (computation is free).

Complexity: $\kappa(C) =$ minimum over protocols of maximum cost.

Trivial solution: Alice sends x, Bob computes the bit $f(x,y)$ and sends it back to Bob. So, $n+1$ is an upper bound on the cost.

Combinatorial formulation

Communication matrix $C = (c_{xy})$ of type $2^n \times 2^n$, where $c_{xy} = f(x, y)$. Alice has row x, Bob has column y.
At each time of the protocol, both Alice and Bob know that (x, y) is in some submatrix M (selected rows and columns).

- When Alice sends a bit, this decreases M by choosing a subset of the rows.
- When Bob sends a bit, this decreases M by choosing a subset of the columns.

The algorithm stops when M has all 0's or all 1's.
Protocol is a decision tree: each node has a submatrix M, shows who splits it and how.
$\kappa(C) =$ is the smallest possible depth of a decision tree.

A lower bound tool

Theorem $\kappa(C) \geq 1 + \log \operatorname{rank}(C)$.

Proof. The number of leaves of the decision tree that have all-1 submatrices is $\leq 2^{\kappa(C)-1}$. Each such submatrix contributes at most 1 to the rank. □

Example Let $f(x, y) = 1$ iff $x = y$. The rank of the matrix is clearly 2^n, so the trivial upper bound $n + 1$ is exact.

- How many bits do Alice and Bob have to exchange, if they want to find the value $f(x, y)$ only with error probability bounded by $1/3$?
- We will treat only the example where $f(x, y) = 1$ iff $x = y$. We will use the idea already introduced in an assignment, checking $x = y$ in logarithmic space. Let $|x|, |y| \leq n$. We will treat them as numbers. Choose some $N > n$ to be determined later.
  Alice: choose a random prime $p < N$, and send p and $x \bmod p$. (We do not count the computational complexity of this now, even though it is not large.)
  Bob: accept iff $x \bmod p = y \bmod p$.

We will analyze the probability of failure.
The following two facts of number theory are used without proof. Let $\pi(n)$ be the number of primes less than n, and $\Pi(n)$ their product.

Theorem $\pi(n) \sim \frac{n}{\ln n}$, $\Pi(n) > 2^n$ for large n.

The protocol uses $2\log N$ bits. It clearly accepts if $x = y$. What is the probability that it accepts when $x \neq y$?
Let $d = |x - y|$, with prime divisors $q_1, \ldots, q_k$, then, with p_k the kth prime:

$$2^n > d \geq q_1 \cdots q_k \geq 2 \cdot 3 \cdots p_k > 2^{p_k},$$

hence $k \leq \pi(n)$. So the probability that our random prime p divides d is

$$\leq \frac{k}{\pi(N)} \leq \frac{\pi(n)}{\pi(N)} \sim \frac{n}{N}.$$

Choose $N = 3n$, then this is $\sim 1/3$. We proved

Theorem The randomized communication complexity of $x = y$ is only $2\log n + O(1)$.