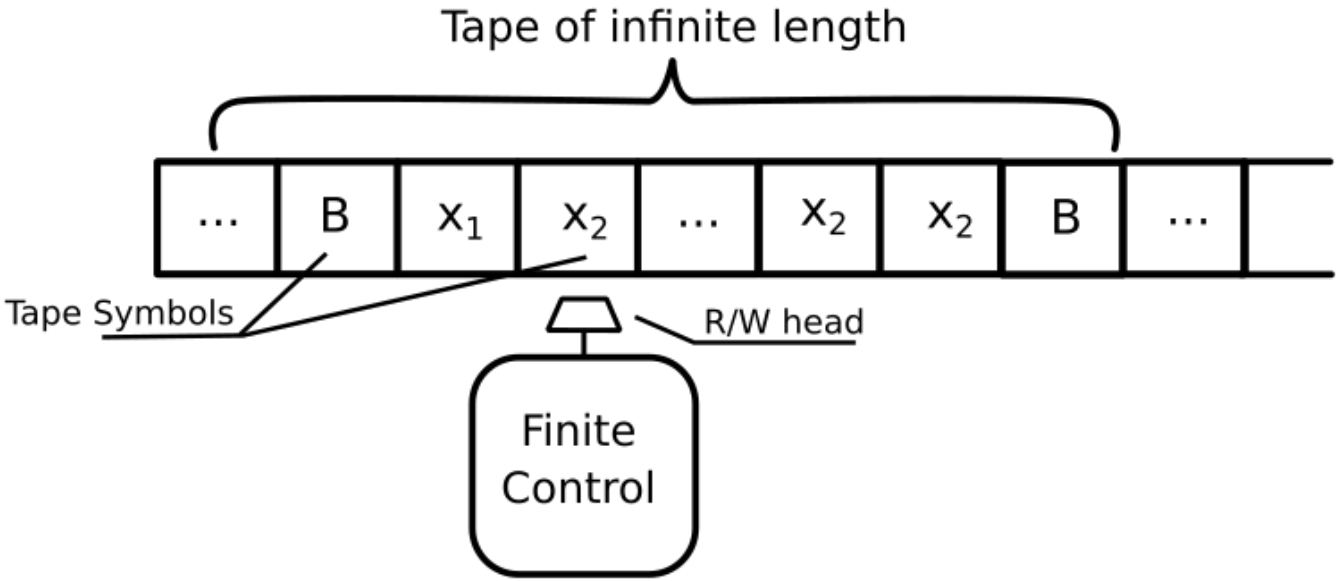
Quantum Algorithms
Lecture 8
Classical computation - summary

Zhejiang University

Turing machines

TM definition reminder

Turing machine has tape, where input is given and machine can write on it. Machine has finite control that can be described by finite set of states and transition function that describes the behavior of the machine.



TM computational step

In each computational step:

- TM reads the symbol s_p placed under its head.
- TM computes the value of the transition function: $\delta(q, s_p) = (q', s', \Delta p)$ that depends on symbol s_p and current inner state q; if step is not defined, TM stops.
- TM writes the symbol s' in cell p of the tape, moves the head by Δp, and passes to state q'.

TM that reverses the input

Task 1.1. Construct a Turing machine that reverses its input (e.g., produces "0010111" from "1110100").

TM that reverses the input

The idea is simple: the machine moves symbols alternately from left to right and from right to left until it reaches the center of the input string, at which point it stops.

Example:

11000

0100**1**

0**0**1**1**1

00**0**11

TM that reverses the input

We assume that the external alphabet A is {0,1}. The alphabet S ={ _,0,1,∗,0',1'} consists of the symbols of the external alphabet, the empty symbol _, and three auxiliary marks used to indicate the positions from which a symbol is taken and a new one should be dropped.

The set of states is

Q = {q0, qf, r0, r1, l0, l1, l0', l1'}.

The letters r and l indicate the direction of motion, and the subscripts at these letters refer to symbols being transferred.

TM that reverses the input

Beginning of work:

$(q0,0) \rightarrow (r0, *, +1)$, $(q0,1) \rightarrow (r1, *, +1)$,

$(q0, \_) \rightarrow (q0, \_, -1)$.

The first line indicates that the machine places a mark in the first position and moves the symbol that was there to the right. The second line indicates that the machine stops immediately at the empty symbol.

TM that reverses the input

Transfer to the right:

$(r0, 0) \rightarrow (r0, 0, +1)$, $(r1, 0) \rightarrow (r1, 0, +1)$,

$(r0, 1) \rightarrow (r0, 1, +1)$, $(r1, 1) \rightarrow (r1, 1, +1)$.

The machine moves to the right until it encounters the end of the input string or a mark.

TM that reverses the input

A change in the direction of motion from right to left consists of two actions: remove the mark (provided this is not the empty symbol)

$(r0, 0') \rightarrow (l0', 0, -1)$, $(r1, 0') \rightarrow (l1, 0, -1)$,
$(r0, 1') \rightarrow (l0', 1, -1)$, $(r1, 1') \rightarrow (l1', 1, -1)$,
$(r0, \_) \rightarrow (l0', \_, -1)$, $(r1, \_) \rightarrow (l1', \_, -1)$

and place it in the left adjacent position

$(l0', 0) \rightarrow (l0, 0', -1)$, $(l1', 0) \rightarrow (l0, 1', -1)$
$(l0', 1) \rightarrow (l1, 0', -1)$, $(l1', 1) \rightarrow (l1, 1', -1)$.

TM that reverses the input

Transfer to the left:

$(l0, 0) \to (l0, 0, -1)$, $(l1, 0) \to (l1, 0, -1)$,
$(l0, 1) \to (l0, 1, -1)$, $(l1, 1) \to (l1, 1, -1)$.

Change of direction from left to right:

$(l0, *) \to (q0, 0, +1)$, $(l1, *) \to (q0, 1, +1)$.

TM that reverses the input

The completion of work depends on the parity of the word length: for even length, the machine stops at the beginning of the motion to the right

$(q0, 0') \to (qf, 0, -1), (q0, 1') \to (qf, 1, -1),$

and for odd length, — at the beginning of the motion to the left

$(l0', *) \to (qf, 0, -1), (l1', *) \to (qf, 1, -1).$

TM that reverses the input

The transition function is undefined for the state qf; therefore, the machine stops after switching to this state.

Computational complexity

We say that a Turing machine works in time $T(n)$ if it performs at most $T(n)$ steps for any input of size n. Analogously, a Turing machine M works in space $s(n)$ if it visits at most $s(n)$ cells for any computation on inputs of size n.

Time – number of computational steps.

Space – number of cells visited by a TM.

Universal TM

TM is a finite object, and it can be encoded by a string. We can consider a universal TM U. Its input is a pair $([M], x)$, where $[M]$ is the encoding of a machine M with external alphabet A, and x is a string over A.

The output of U is $\phi_{\mathrm{M}}(x)$. Thus U computes the function u defined as follows:

$$u([M], x) = \phi_{\mathrm{M}}(x)$$

This approach gives important idea – the program can be considered as input data as well. This is the case of real computers – it is universal, and we give it software together with input.

Polynomial growth

A function $f(n)$ is of polynomial growth if $f(n) \leq cn^d$ for some constants c, d and for all sufficiently large n.

$$f(n) = poly(n)$$

Poly-time computation

A function F on B^* is computable in polynomial time if there exists a Turing machine that computes it in time $T(n) = poly(n)$, where n is the length of the input.

If F is a predicate, we say that it is decidable in polynomial time.

$B = \{0,1\}$; B^* - binary string.

The class of all functions computable in polynomial time, or all predicates decidable in polynomial time, is denoted by P.

Poly-time computation

A function F on B^* is computable in polynomial time if there exists a Turing machine that computes it in time $T(n) = poly(n)$, where n is the length of the input.

If F is a predicate, we say that it is decidable in polynomial time.

$B = \{0,1\}$; B^* - binary string.

The class of all functions computable in polynomial time, or all predicates decidable in polynomial time, is denoted by P.

Class PSPACE

A function (predicate) F on B^* is computable (decidable) in polynomial space if there exists a Turing machine that computes F and runs in space $s(n) = poly(n)$, where n is the length of the input.

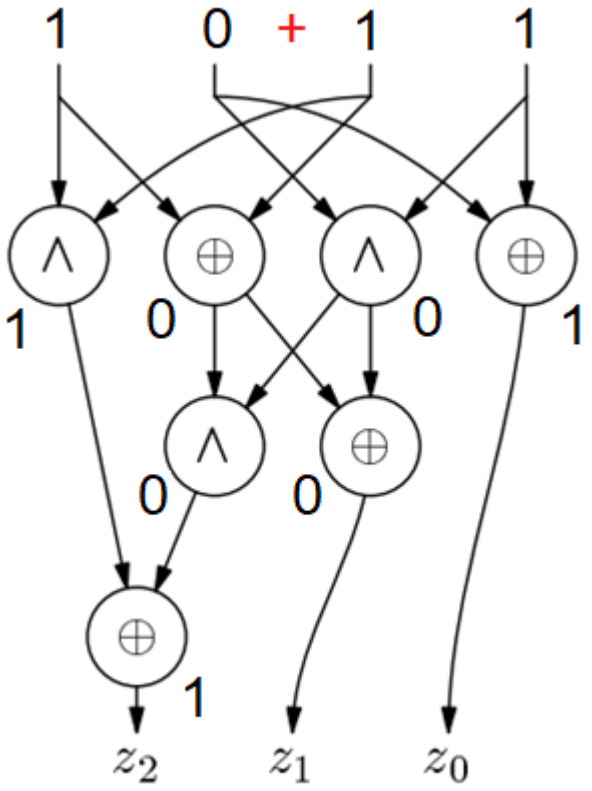
Boolean circuits

Boolean circuit example

$u_1 = 1 \wedge 1 = 1$
$u_2 = 1 \oplus 1 = 0$
$u_3 = 0 \wedge 1 = 0$
$z_0 = 0 \oplus 1 = 1$

$u_4 = 0 \wedge 0 = 0$
$z_1 = 0 \oplus 0 = 0$

$z_2 = 1 \oplus 0 = 1$



DNF

Disjunctive normal form. By definition, a DNF is a disjunction of conjunctions of literals.

$$f(x) = \bigvee_{\{i=1\ldots m\}} \mathrm{f}_i(x),$$

where each $f_i(x) = \bigwedge_{k=1\ldots l} u_k$, where each u_k is either some x or $\neg x$.

CNF

Conjunctive normal form — a conjunction of disjunctions of literals. Any Boolean function can be represented by a CNF. We can represent $\neg f$ by a DNF and then get a CNF for f by negation using De Morgan's identities:

$x \wedge y = \neg(\neg x \vee \neg y), x \vee y = \neg(\neg x \wedge \neg y).$

Class P/poly

P/poly = nonuniform P.

A predicate F belongs to the class P/poly if $c(F_n) = poly(n)$.

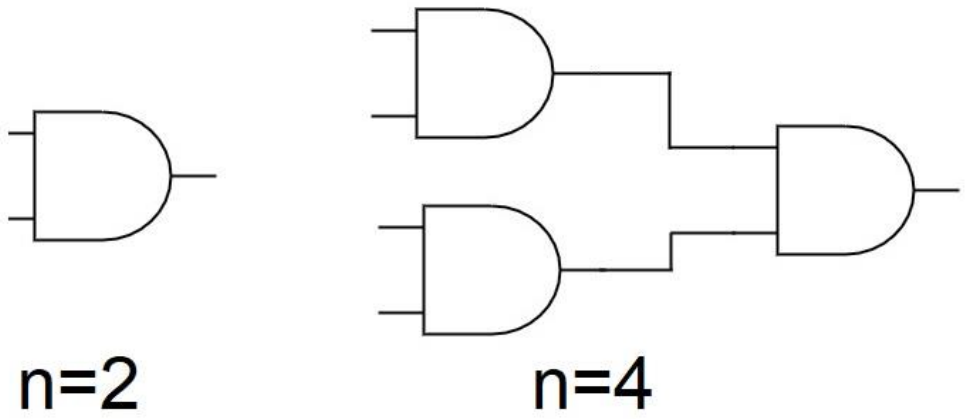
F_n - inputs of F are of length n.

$c(F_n)$ - maximum complexity (size) of F_n.

Nonuniform - definition

The term “nonuniform” indicates that a separate procedure, i.e., a Boolean circuit, is used to perform computation with input strings of each individual length.

This means that for each input length n different Boolean circuit is used.



Relation between P and P/poly

F belongs to P if and only if these conditions hold:

- $F \in$ P/poly;
- the functions F_n are computed by polynomial-size circuits C_n with the following property: there exists a TM that for each positive integer n runs in time $poly(n)$ and constructs the circuit C_n.

A sequence of circuits C_n with this property is called polynomial-time uniform.

P/poly small summary

P/poly can also be considered as complexity class where advice of polynomial size is provided together with input. Advice only depends on the size n. Why? Because additional information of polynomial size may be encoded into the circuit.

Consequence is the following – if input is unary (e.g., 11…1), then any problem can be solved in P/poly.

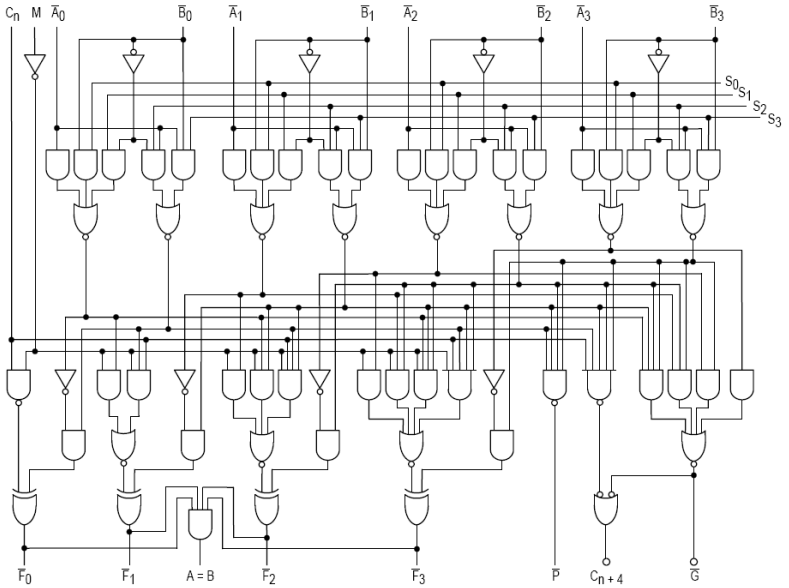
P/poly small summary

If our complexity class is EXPTIME, then we are able to encode solutions of all possible binary inputs of length n into the circuit, because the complexity of a circuit is $2^{poly(n)}$. Therefore, nonuniform version of EXPTIME contains all possible predicates, even noncomputable ones.

Classical Boolean circuits

Let us consider real life example.

Current classical CPUs have a lot of operations implemented as Boolean circuits, including basic Math operations like addition and multiplication. Therefore, CPU designers need to understand Boolean circuits. Here is (four-bit) arithmetic logic unit example:



Four bit addition circuit

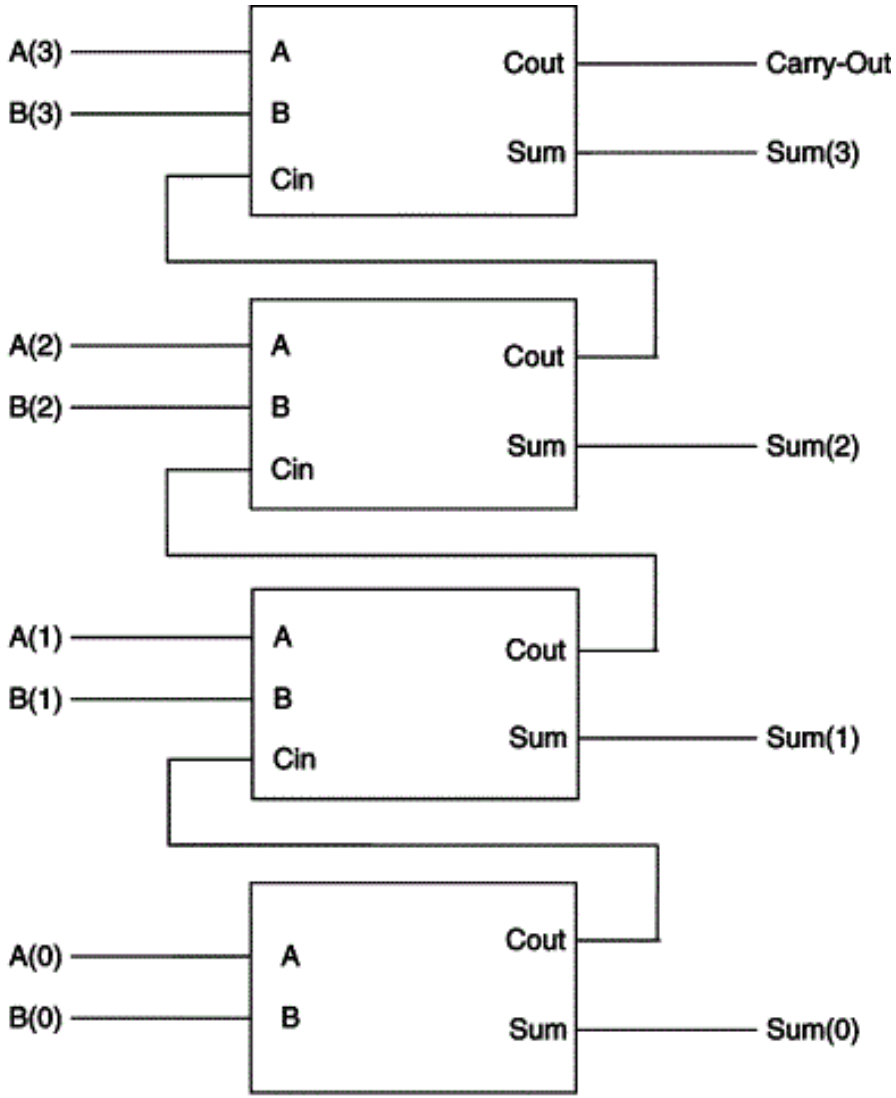
We will see the implementation of Boolean circuit for 4-bit binary addition. We should sum together according bits of two numbers and take care of carry bits (e.g., if A(0)=1 and B(0)=1, then we will have carry bit = 1 that will affect the value of the next bit.

On next page Cin – is carry bit that circuit receives and Cout – carry bit that circuit produces.

4-bit binary addition

	A(3)	A(2)	A(1)	A(0)	
	B(3)	B(2)	B(1)	B(0)	+
Carry-Out	Sum(3)	Sum(2)	Sum(1)	Sum(0)	

Four bit addition circuit

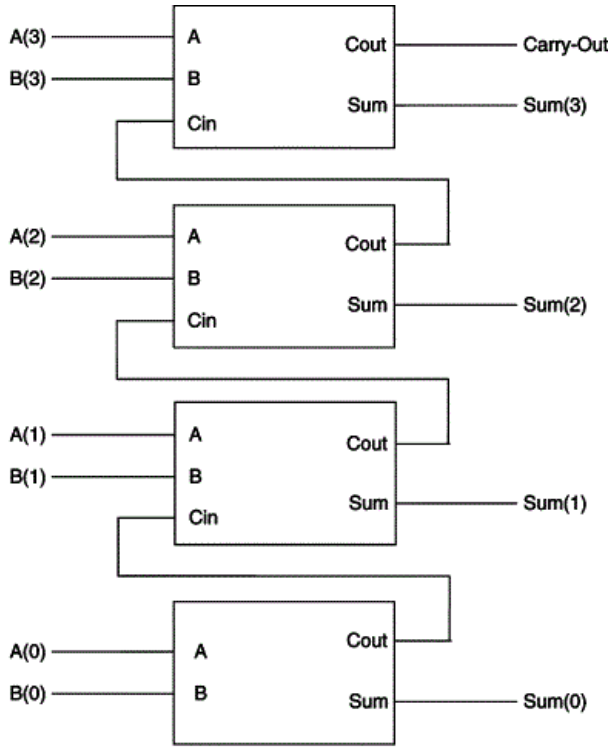


Four bit addition circuit

For each block we should calculate the Sum value and the value of Cout.

Sum=Cin⊕(A⊕B)

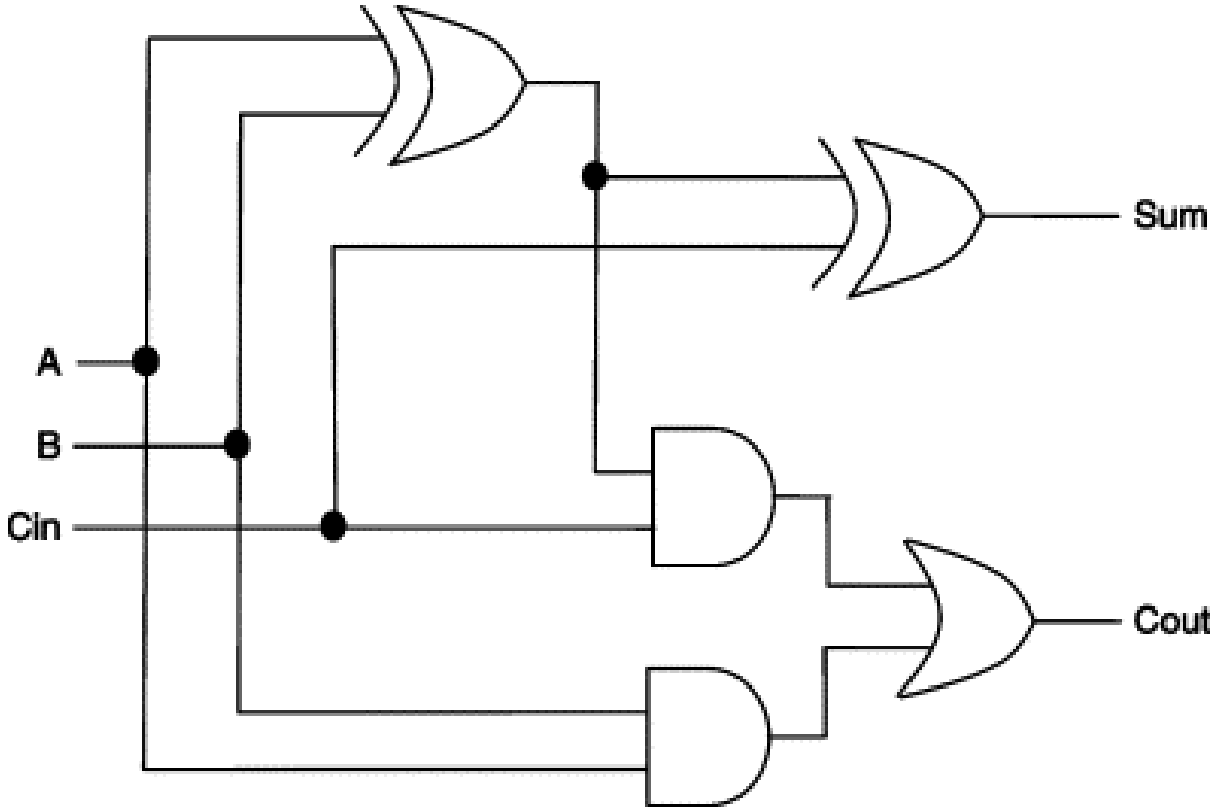
Cout=(A ∧ B) ∨ (Cin ∧ (A⊕B))



Four bit addition circuit

Sum=Cin⊕(A⊕B)

Cout=(A ∧ B) ∨ (Cin ∧ (A⊕B))



The class NP: Reducibility and completeness

Nondeterministic Turing machine

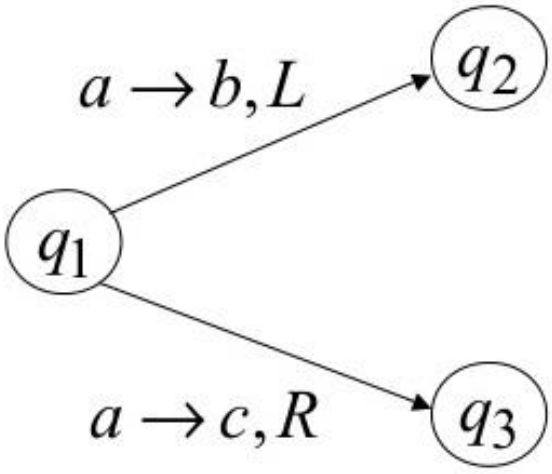
$\delta(q, s_p) = (q', s', \Delta p)$

For each pair (q, s_p) can exist more than one transition, as if machine would be able to do this transitions in parallel.

In this example:

$\delta(q1, a) = (q2, b, \mathrm{L})$

$\delta(q1, a) = (q3, c, \mathrm{R})$



NP - alternative definition

A predicate L belongs to the class NP if it can be represented as

$$L(x) = \exists y((|y| < q(|x|)) \wedge R(x, y)),$$

where q is a polynomial (with integer coefficients), and R is a predicate of two variables decidable in polynomial time.

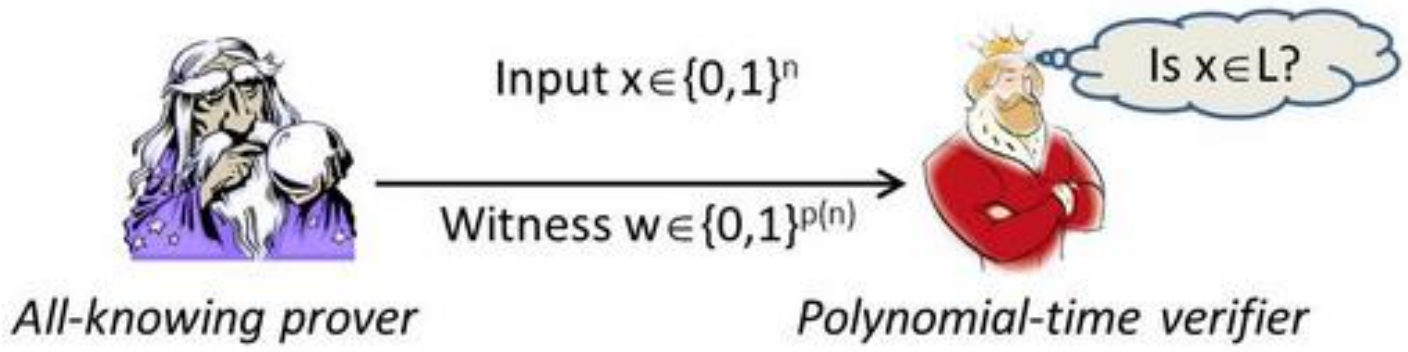
Problems, whose solutions can be checked efficiently, belong to class NP. Given possible solution can be verified in polynomial time.

$R(x, y)$ – here x is given input, and y is provided solution.

Another description of NP

- $L(x) = 1$ => M can convince A that $L(x)$ is true by presenting some proof y such that $R(x, y)$;
- $L(x) = 0$ => whatever M says, A is not convinced: $R(x, y)$ is false for any y.

Moreover, the proof y should have polynomial (in $|x|$) length, otherwise A cannot check $R(x, y)$ in polynomial (in $|x|$) time.



Karp reducibility

A predicate $L1$ is reducible to a predicate $L2$ if there exists a function $f \in P$ such that $L1(x) = L2(f(x))$ for any input string x.

We say that f reduces $L1$ to $L2$. Notation: $L1 \propto L2$.

Karp reducibility is also called "polynomial reducibility".

NP-completeness

A predicate $L \in NP$ is NP-complete if any predicate in NP is reducible to it.

If some NP-complete predicate can be computed in time $T(n)$, then any NP-predicate can be computed in time $poly(n) + T(poly(n))$. Therefore, if some NP-complete predicate belongs to P, then P = NP. Put is this way: if P ≠ NP (which is probably true), then no NP-complete predicate belongs to P.

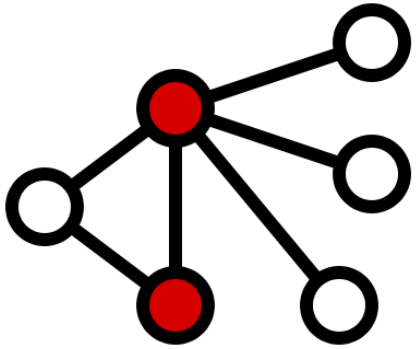
Reduction task example

We will consider an example – how to reduce the VERTEX-COVER problem to LIN-INEQ problem.

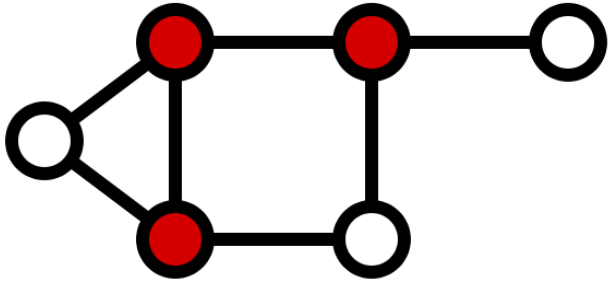
VERTEX-COVER definition

Given graph G and number k, verify, whether there is a set of k vertices that includes at least one endpoint of every edge of the graph (in other words, that for each edge uv at least one of vertices, u or v, is in that set of k vertices).

Examples:



$k = 2$



$k = 3$

LIN-INEQ definition

Whether linear inequalities have an integer solution.

Example:

$x1 + x2 + x3 \geqslant 1$

$(1 - x1) + x4 + (1 - x5) \geqslant 1$

$(1 - x2) + x5 + (1 - x6) \geqslant 1$

$(1 - x3) + x4 + (1 - x5) \geqslant 1$

$$0 \leq x_1, x_2, x_3, x_4, x_5, x_6 \leq 1$$

Reduction

We will reduce in general VERTEX-COVER problem to LIN-INEQ problem.

We have the following data of VERTEX-COVER problem: given graph G and number k.

Now we need to generate linear inequalities for LIN-INEQ.

Reduction

We use each vertex of the graph as a variable (x1,x2,…) so that the solution to our inequalities will correspond to picked k vertices.

Each variable will have one of the values {0,1}, where 1 means that the vertex has been picked. For example, $x1 = 1$ means that vertex x1 is picked for cover set, but $x2 = 0$ means, that vertex x2 is not picked.

Reduction

For each edge uv at least one vertex should be picked, so for each uv we write inequality:

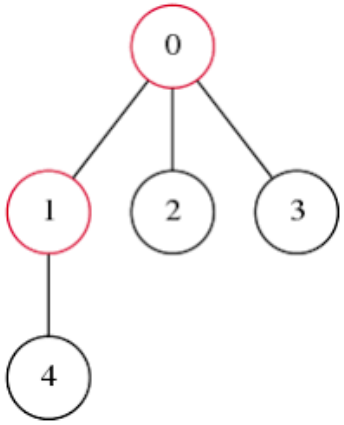
$$xu + xv \geq 1$$

At the same time, total number of picked edges should be equal to k:

$$\sum_{m=1}^{n} x_m = 1$$

Reduction example

We are given $k = 2$ and the following graph:



Reduction:

$$x_0, x_1, x_2, x_3, x_4 \in \{0,1\}$$

$$x_0 + x_1 \geq 1,\ x_0 + x_2 \geq 1,\ x_0 + x_3 \geq 1,\ x_1 + x_4 \geq 1$$

$$x_0 + x_1 + x_2 + x_3 + x_4 = 2$$

Probabilistic algorithms and the class BPP

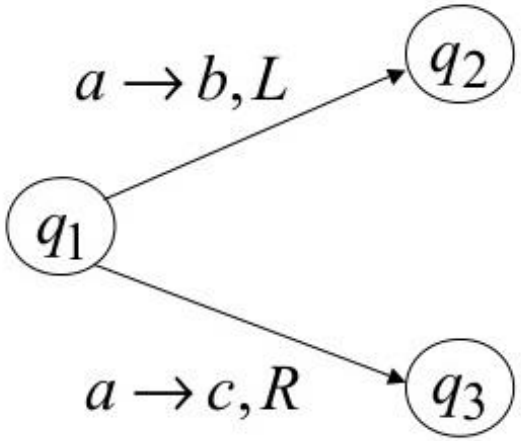
PTM

A probabilistic Turing machine (PTM) is somewhat similar to a nondeterministic one; the difference is that choice is produced by coin tossing, not by guessing.

For example:

$\delta(q1, a) = (q2, b, \mathrm{L})$ with probability 1/2

$\delta(q1, a) = (q3, c, \mathrm{R})$ with probability 1/2



BPP

Let ε be a constant such that $0 < \varepsilon < 1/2$. A predicate L belongs to the class BPP if there exist a PTM M and a polynomial $p(n)$ such that the machine M running on input string x always terminates after at most $p(|x|)$ steps, and

- $L(x) = 1 \Rightarrow M$ gives the answer “yes” with probability $\geq 1 - \varepsilon$;
- $L(x) = 0 \Rightarrow M$ gives the answer “yes” with probability $\leq \varepsilon$.

BPP - equivalent definition

A predicate L belongs to BPP if there exist a polynomial p and a predicate R, decidable in polynomial time, such that

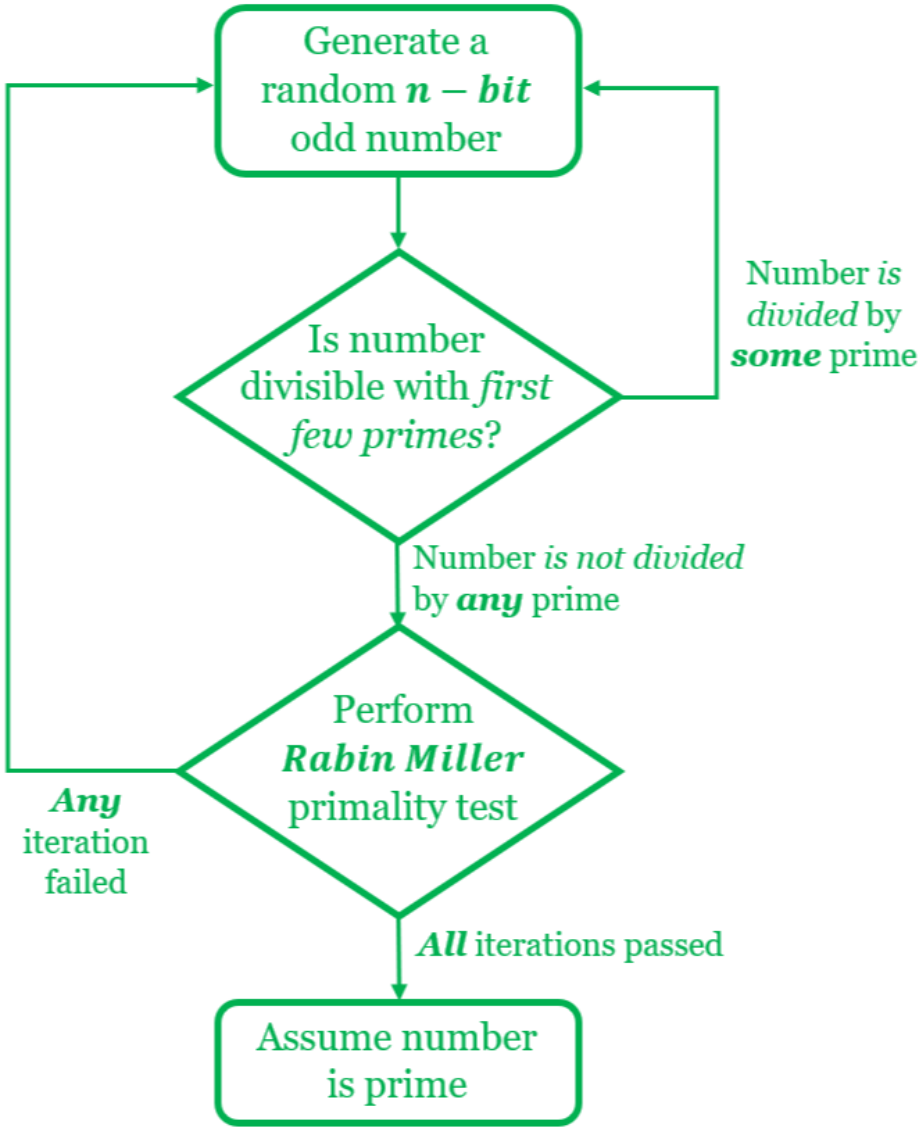
- $L(x) = 1 \Rightarrow$ the fraction of strings r of length $p(|x|)$ satisfying $R(x, r)$ is greater than $1 - \varepsilon$;
- $L(x) = 0 \Rightarrow$ the fraction of strings r of length $p(|x|)$ satisfying $R(x, r)$ is less than ε.

Here strings r denote strings of random bits – e.g., probabilistic choices of PTM. This reminds us proof/solution like in NP definition.

Probabilistic algorithm example

In previous lectures we checked the probabilistic procedure to check whether the given number is prime. Now we will consider an algorithm that generates random prime number. We will check an algorithm, that is described here: https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm/

Generating random prime



Random prime – step 1

Preselect a random number with the desired bit-size. This is probabilistic step.

For example, 1024-bit random number. This number will be more than 300 decimal digits long.

Random prime – step 2

Check that generated number is not divisible by some smallest prime numbers. For example, by: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349.

Random prime – step 3

Apply a certain number of Rabin Miller Primality Test iterations, based on acceptable error rate, to get a number which is probably a prime.

Usually, in commercial applications, we require error probabilities to be less than $1/2^{128}$.

Recall that the Miller-Rabin test uses the following witnesses for the compositeness: if $b^2 \equiv 1 (mod\ q)$, and $b \neq \pm 1\ (mod\ q)$ for some b, then q is composite.

To have probability of correctness $1/2^{128}$, we need to repeat Miller-Rabin test for 128 random values of b.

Random prime – analysis

Step 1 requires to generate n random bits, so number of operations is polynomial in size of number that we are generating.

Step 2 requires constant number of division operations, each is doable in polynomial number of steps.

Step 3 requires constant number of Miller-Rabin tests, each in polynomial time.

Therefore, one try to generate a prime number runs in polynomial time.

Random prime – analysis

The prime number theorem that among x first numbers there is approximately $x/logx$ prime numbers.

This means, that if we check random n bit number, then probability of it to be a prime number is roughly $1/n$. Which means, that we will have linear number of tries to generate a prime number with very high probability.

In conclusion, runtime is polynomial, so the algorithm belongs to BPP.

Random prime – result

Generated 1024 bit prime is:

178542003245811211274167228297361192303886321036074276889145691522634525820185614278499562592134188995169731066418203258297035264969457638591284906658912408319763156912951486020761069099132619194489006875108217247513715271974383296142805846405783845170862140174184507256128825312324419293575432423822703857091

This number is prime with very high probability.

The hierarchy of complexity classes

Two-player game

Consider a game with two players called White (W) and Black (B). A string x is shown to both players. After that, players alternately choose binary strings: W starts with some string $w1$, B replies with $b1$, then W says $w2$, etc. Each string has length polynomial in $|x|$. Each player is allowed to see the strings already chosen by his opponent.

Description of the game

There is a predicate $W(x, w1, b1, w2, b2, \ldots)$ that is true when W is the winner, and we assume that this predicate belongs to P. If this predicate is false, B is the winner (there are no ties). This predicate (together with polynomial bounds for the length of strings and the number of steps) determines the game.

The game is completed after some prescribed number of steps.

The referee, who knows x and all the strings and who acts according to a polynomial-time algorithm, declares the winner.

Defining classes

Since this game is finite and has no ties, for each x either B or W has a winning strategy.
Therefore, any game determines two complementary sets,
$Lw = \{x\colon\ W\ has\ a\ winning\ strategy\}$,
$Lb = \{x\colon\ B\ has\ a\ winning\ strategy\}$.
Many complexity classes can be defined as classes formed by the sets Lw (or Lb) for some classes of games.

Examples of classes

P: the sets Lw (or Lb) for games of zero length (the referee declares the winner after he sees the input)
NP: the sets Lw for games that are finished after W's first move. In other words, NP-sets are sets of the form $\{x\colon \exists w1\ W(x, w1)\}$.
co-NP: the sets Lb for games that are finished after W's first move. In other words, co-NP-sets are sets of the form $\{x\colon \forall w1\ B(x, w1)\}$.
(here B = ¬W means that B wins the game.)

Examples of classes

Σ_2: the sets Lw for games where W and B make one move each and then the referee declares the winner. In other words, Σ_2-sets are sets of the form
$\{x\colon \exists w1\ \forall b1\ W(x, w1, b1)\}$.
(W can make a winning move $w1$ after which any move $b1$ of B makes B lose).
Π_2: the sets Lb for the same class of games, i.e., the sets of the form
$\{x\colon \forall w1\ \exists b1\ B(x, w1, b1)\}$.

PSPACE – alternative definition

L ∈ PSPACE if and only if there exists a polynomial game such that

$L = \{x\colon W\ has\ a\ winning\ strategy\ for\ input\ x\}$.

By a polynomial game we mean a game where the number of moves is bounded by a polynomial (in the length of the input), players' moves are strings of polynomial length, and the referee's algorithm runs in polynomial time.

Polynomial hierarchy

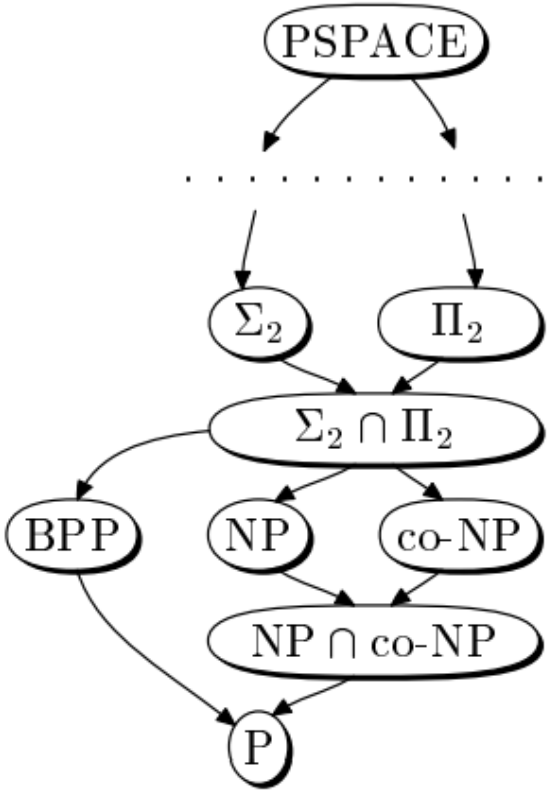
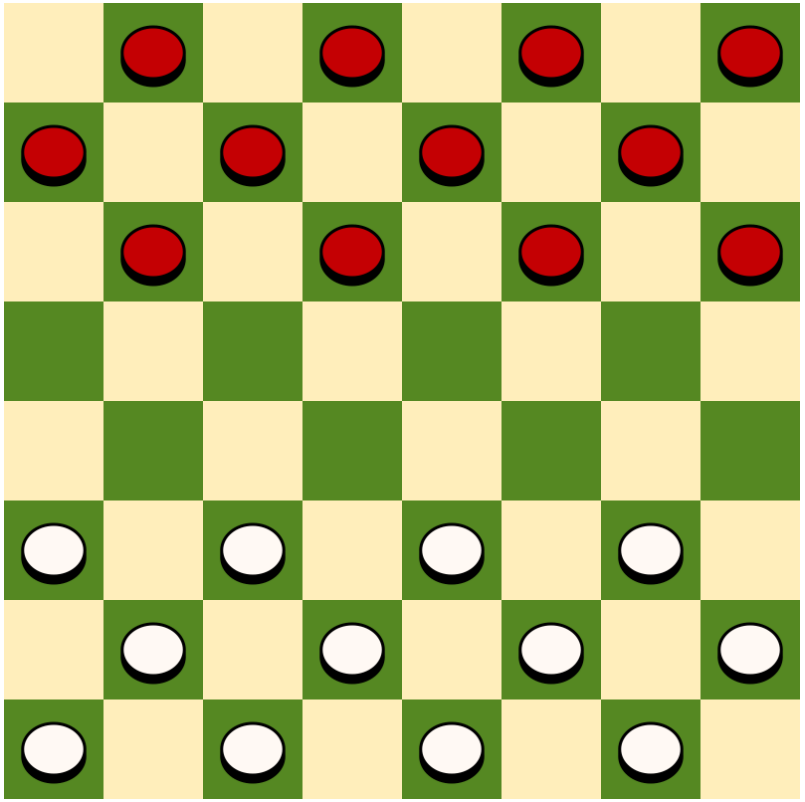


Fig. 5.1. Inclusion diagram for computational classes. An arrow from A to B means that B is a subset of A.

Generalized Checkers as PSPACE problem

Generalized Checkers is played on an $n \times n$ board. It is PSPACE-hard to determine whether a specified player has a winning strategy.



Generalized Checkers as PSPACE problem

The proof is based on the reduction from another problem known to be PSPACE. Let us consider just several thoughts on how we can show that this problem is PSPACE.

Suppose that like in regular checkers we limit the number of moves between jumps (when opponents figure is beaten) by a polynomial. Then it means that the game finishes in polynomial number of moves.

Generalized Checkers as PSPACE problem

If we have a polynomial number of moves, then we arrive at our definition of PSPACE, that shows how we can describe this game in terms of complexity games as few slides ago.

L ∈ PSPACE if and only if there exists a polynomial game such that

$L = \{x\colon W\ has\ a\ winning\ strategy\ for\ input\ x\}$.

Each move can be checked in polynomial space and there are total polynomial number of moves.

Thank you for your attention!