Quantum Algorithms
Lecture 2
Boolean circuits I

Zhejiang University

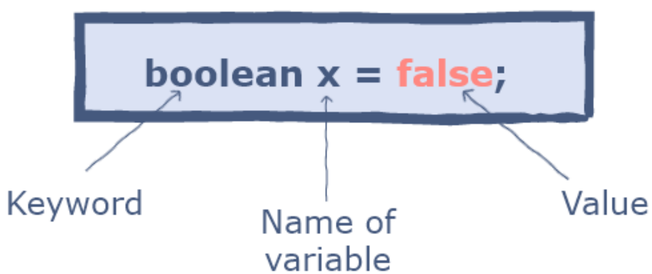
Definitions. Complete bases.

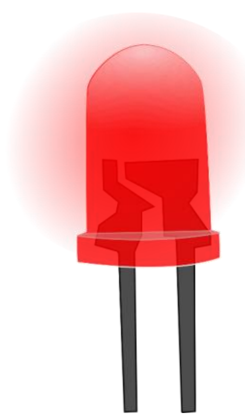
Boolean values

Truth values of logic and Boolean algebra. Can be true or false.

Correlates quite well with classical computing in general, because it can be represented with a single bit.

Example from programming:





Boolean circuit

A Boolean circuit is a representation of a given Boolean function as a composition of other Boolean functions.

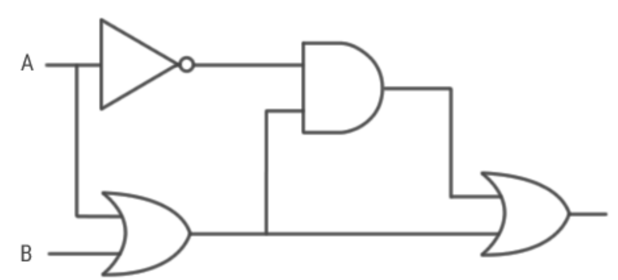
It is like putting blocks together – you compose complicated functions from simple ones.

Boolean function

A function of type $B^n \to B$. (For $n = 0$ we get two constants 0 and 1.) Assume that some set A of Boolean functions (basis) is fixed. It may contain functions with different number of arguments (arity).

Examples:

- 1-ary (unary) NOT NOTx
- 2-ary (binary) OR, XOR, AND x AND y

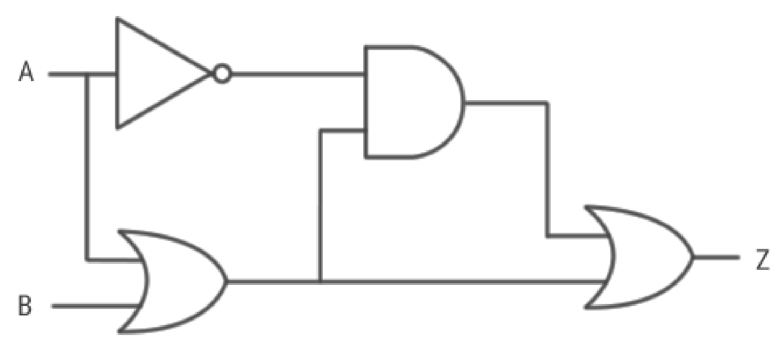


Circuit components

A circuit C over A is a sequence of assignments:

- n input variables $x_1, \dots, x_n$;
- several auxiliary variables $y_1, \dots, y_m$.

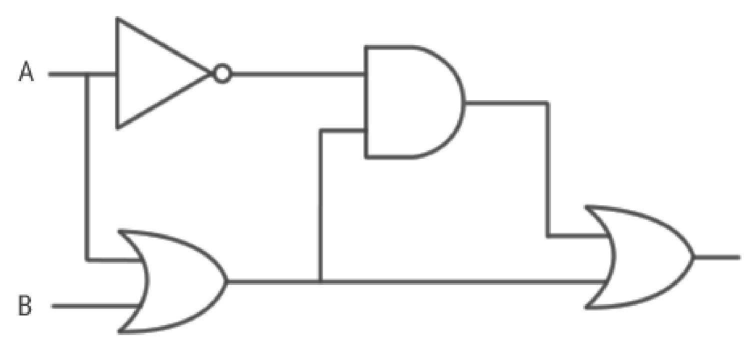
The j-th assignment has the form $y_j = f_j(u_1, \dots, u_r)$. f_j is some function from A, and each of the variables $u_1, \dots, u_r$ is either an input variable or an auxiliary variable that precedes y_j.



Circuit result

Result = the value of the last auxiliary variable.

A circuit with n input variables $x_1, \ldots, x_n$ computes a Boolean function $\mathrm{F}: B^n \to B$ if the result of computation is equal to $F(x_1, \ldots, x_n)$ for any values of $x_1, \ldots, x_n$.

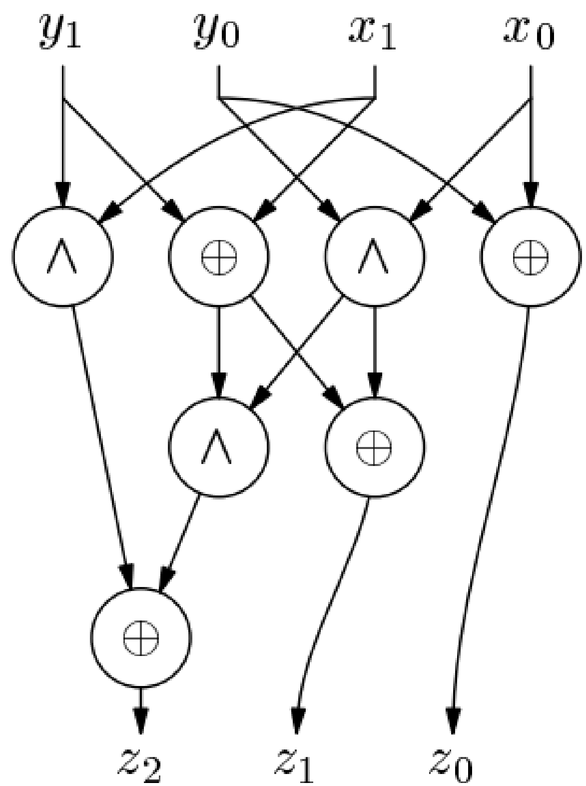


Multiple output bits

m auxiliary variables (instead of one) to be the output.

Circuit computes a function $\mathrm{F}\colon B^n \to B^m$.

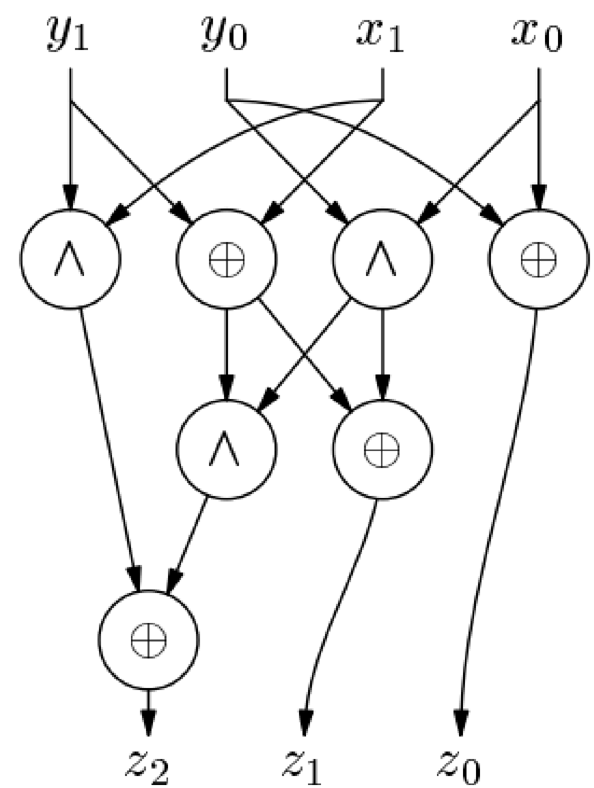
Acyclic directed graph



Circuit over the basis $\{\wedge, \oplus\}$ for the addition of two 2-digit numbers: $\overline{z_2 z_1 z_0} = \overline{x_1 x_0} + \overline{y_1 y_0}$

Graph = assignments

$u_1 = y_1 \wedge x_1$
$u_2 = y_1 \oplus x_1$
$u_3 = y_0 \wedge x_0$
$z_0 = y_0 \oplus x_0$
$u_4 = u_2 \wedge u_3$
$z_1 = u_2 \oplus u_3$
$z_2 = u_1 \oplus u_4$

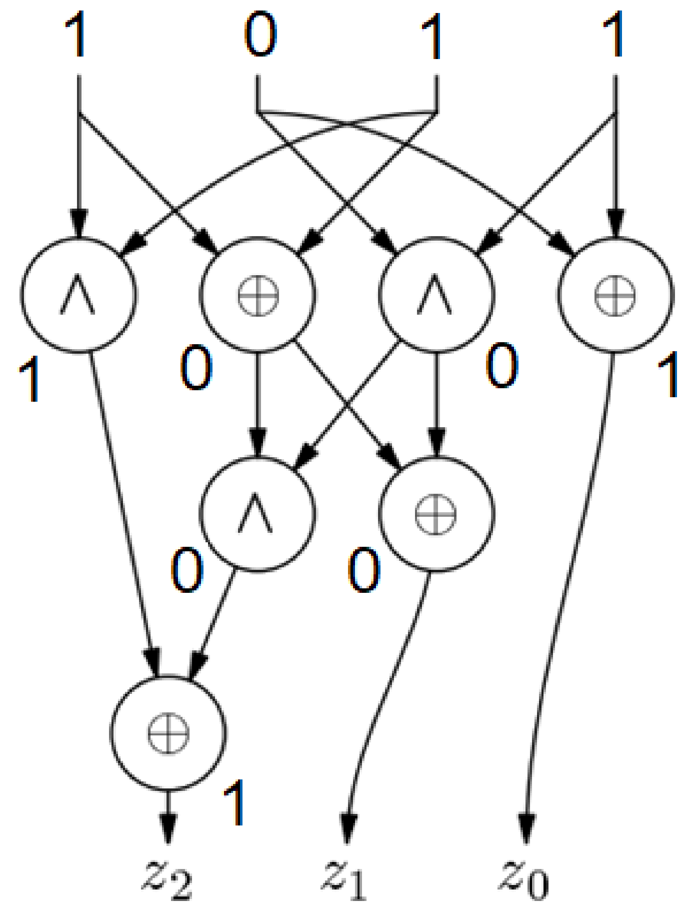


Graph = assignments

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

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

$$z_2 = 1 \oplus 0 = 1$$

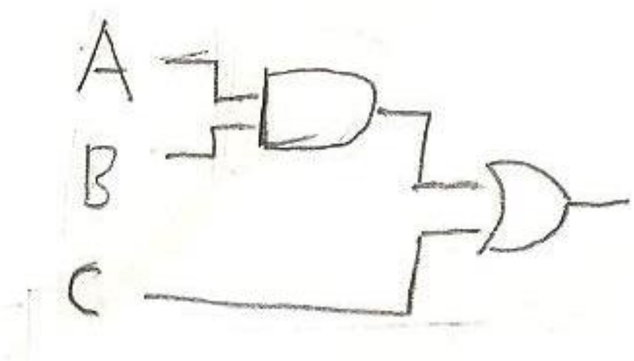


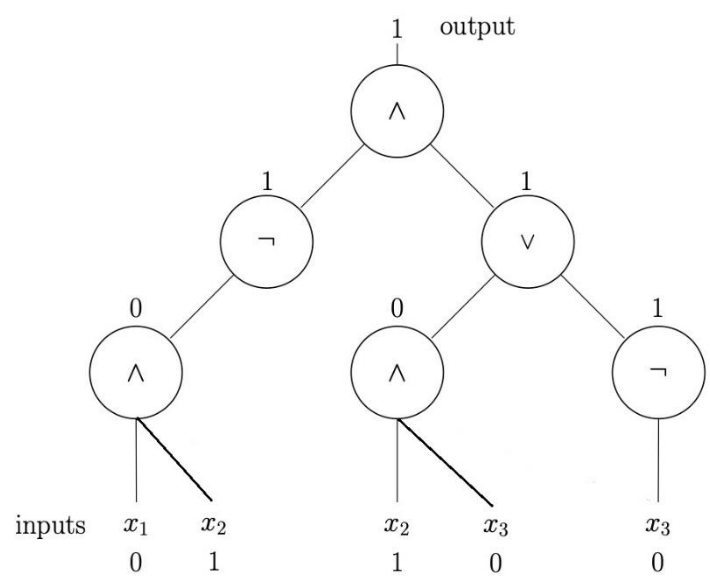
In this example we have calculate binary 10+11

Formula

Each auxiliary variable, except the last one, is used (i.e., appears on the right-hand side of an assignment) exactly once.

The graph of a formula is a tree whose leaves are labeled by input variables; each label may appear any number of times.





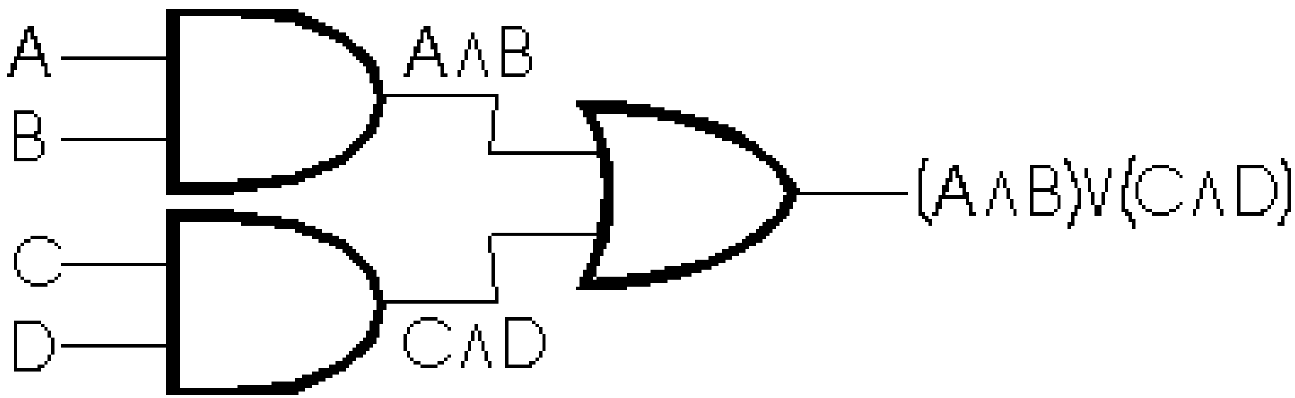
Formula

If each auxiliary variable is used only once, we can replace it by its definition. Performing all these "inline substitutions", we get an expression for f that contains only input variables, functions from the basis, and parentheses. The size of this expression approximately equals the total length of all assignments.

Otherwise, size can grow exponentially.

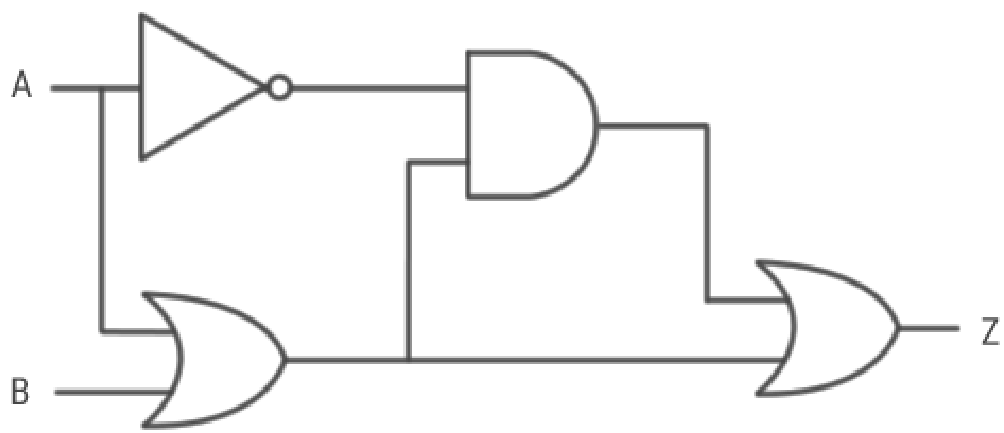
Formula example

$$y_1 = x_1 \wedge x_2,\ y_2 = x_3 \wedge x_4$$
$$f = y_1 \vee y_2 = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$$



Not-formula example

$y_1 = \neg a,\ y_2 = a \vee b,$
$y_3 = y_1 \wedge y_2,$
$f = y_2 \vee y_3 = y_2 \vee (y_1 \wedge y_2) =$
$= (a \vee b) \vee (\neg a \wedge (a \vee b))$



Complete basis

For any Boolean function f, there is a circuit over A that computes f. (It is easy to see that in this case any function of type $B^n \to B^m$ can be computed by an appropriate circuit.)

The most common basis

NOT(x) = ¬x, OR(x1, x2) = x1 ∨ x2, AND(x1, x2) = x1 ∧ x2.

x	$\neg x$
0	1
1	0

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

NOT, AND, OR = complete basis

Any Boolean function of n arguments is determined by its value table, which contains 2^n rows. Each row contains the values of the arguments and the corresponding value of the function.

NOT, AND, OR = complete basis

If the function takes value 1 only once, it can be computed by a conjunction of literals; each literal is either a variable or the negation of a variable.

For example, if $\mathrm{f}(x_1, x_2, x_3)$ is true (equals 1) only for $x_1 = 1$, $x_2 = 0$, $x_3 = 1$,
then

$$\mathrm{f}(x_1, x_2, x_3) = x_1 \wedge \neg x_2 \wedge x_3$$

(the conjunction is associative, so we omit parentheses; the order of literals is also unimportant).

NOT, AND, OR = complete basis

In the general case, a function f can be represented in the form

$$f(x) = \bigvee_{\{u:f(u)=1\}} \chi_u(x),$$

where $u = (u_1, \dots, u_n)$, and χ_u is the function such that $\chi_u(x) = 1$ if $x = u$, and $\chi_u(x) = 0$ otherwise.

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 $f_i(x) = \wedge_{k=1\ldots l} u_k$, 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:

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

DNF to CNF example

	$\equiv \neg((\neg P \lor Q) \land R \land (\neg P \lor Q))$	
	$\equiv \neg(\neg P \lor Q) \lor \neg R \lor \neg(\neg P \lor Q)$	deM.
	$\equiv (\neg\neg P \land \neg Q) \lor \neg R \lor (\neg\neg P \land \neg Q)$	deM.
(DNF)	$\equiv (P \land \neg Q) \lor \neg R \lor (P \land \neg Q)$	double neg.
	$\equiv ((P \lor \neg R) \land (\neg Q \lor \neg R)) \lor (P \land \neg Q)$	distr.
	$\equiv ((P \lor \neg R) \lor (P \land \neg Q)) \land$	distr.
	$((\neg Q \lor \neg R) \lor (P \land \neg Q))$	
	$\equiv (((P \lor \neg R) \lor P) \land ((P \lor \neg R) \lor \neg Q)) \land$	distr.
	$(((\neg Q \lor \neg R) \lor P) \land ((\neg Q \lor \neg R) \lor \neg Q))$	
	$\equiv (P \lor \neg R) \land (P \lor \neg R \lor \neg Q) \land (\neg Q \lor \neg R)$	assoc. comm. idemp.

Other complete bases

The basis {¬, ∨, ∧} is redundant: the subsets {¬, ∨} and {¬, ∧} also constitute complete bases. Another useful example of a complete basis is {∧, ⊕}.

Circuit complexity

The number of assignments in a circuit is called its size. The minimal size of a circuit over A that computes a given function f is called the circuit complexity of f (with respect to the basis A) and is denoted by $c_A(f)$. The value of $c_A(f)$ depends on A.

Circuit complexity

The transition from one finite complete basis to another changes the circuit complexity by at most a constant factor:

if A_1 and A_2 are two finite complete bases, then $c_{A_1}(f) = O(c_{A_2}(f))$ and vice versa. Indeed, each A_2-assignment can be replaced by $O(1)$ A_1-assignments since A_1 is a complete basis.

Is set a complete basis?

Construct an algorithm that determines whether a given set of Boolean functions A constitutes a complete basis. (Functions are represented by tables.)

Maximum complexity

Let c_n be the maximum complexity $c(f)$ for Boolean functions f in n variables. For sufficiently large n: $1.99^n < c_n < 2.01^n$.

An upper bound $O(n2^n) < 2.01^n$ (for large n) follows immediately from the representation of the function in disjunctive normal form.

$O(n2^n)$ – we have n variables and 2^n possible bracket combinations of up to n variables each.

Maximum complexity

Let c_n be the maximum complexity $c(f)$ for Boolean functions f in n variables. For sufficiently large n: $1.99^n < c_n < 2.01^n$.

To obtain a lower bound, we compare the number of Boolean functions in n variables (i.e., 2^{2^n}) and the number of all circuits of a given size. Conclusions are made by analyzing combinatorics.

Circuits versus Turing machines

Fixing length n

Any predicate F on B^* can be restricted to strings of fixed length n, giving rise to the Boolean function

$$F_n(x_1, \ldots, x_n) = F(x_1 x_2 \cdots x_n)$$

Thus F may be regarded as the sequence of Boolean functions F_0, F_1, F_2, ….

Function as a sequence of functions

A function of type $F: B^* \to B^*$ can be represented by a sequence of functions $F_n: B^n \to B^{p(n)}$, where $p(n)$ is a polynomial with integer coefficients.

Functions may be partial.

Class P/poly

P/poly = nonuniform P.

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

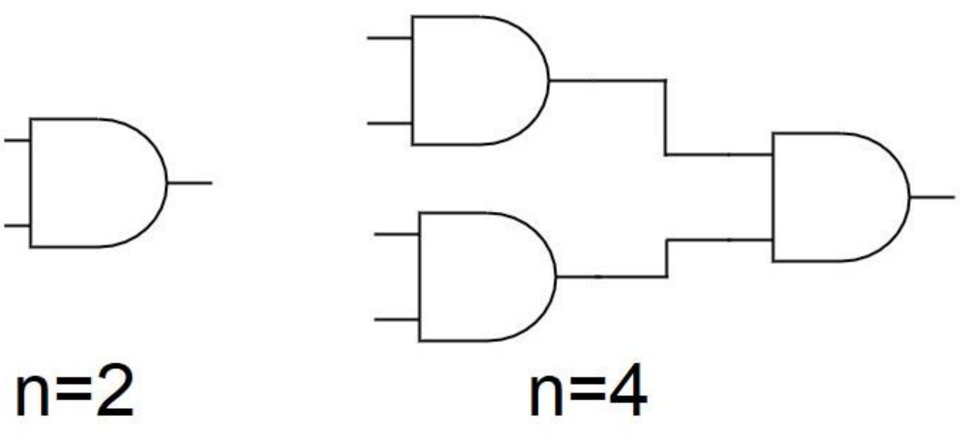
F_n - inputs of F are of length n.

$c(F_n)$ - 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.



P ⊂ P/poly

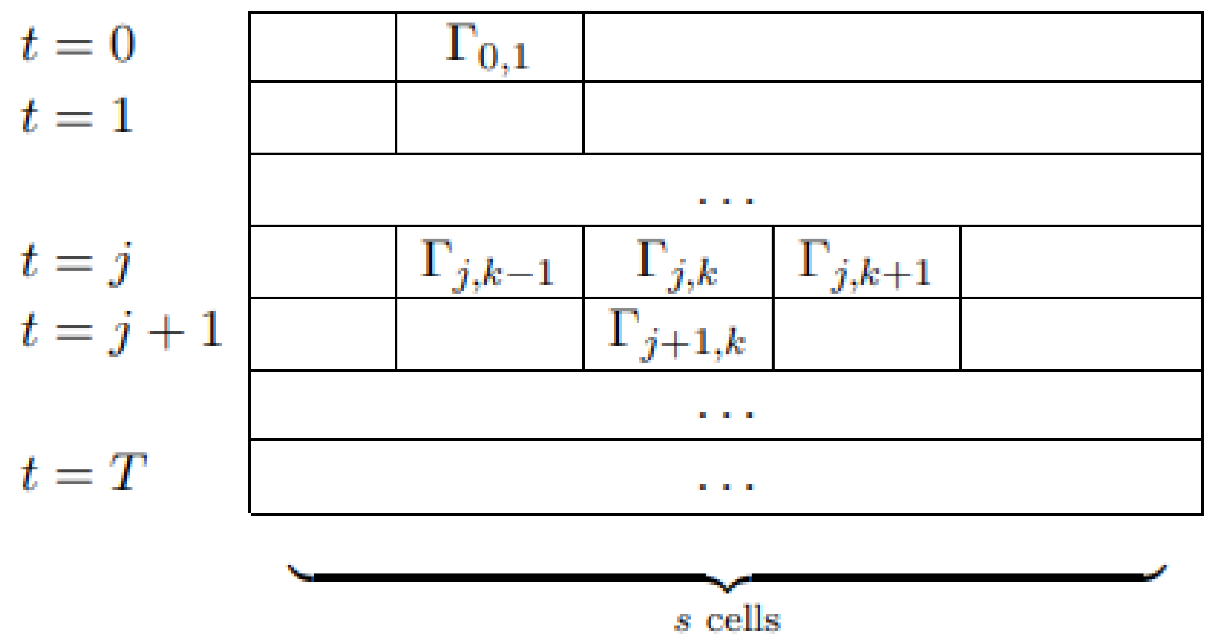
We have to prove that $F \in$ P/poly.

F – a predicate decidable in polynomial time.

M – a TM that computes F and runs in polynomial time. This also means polynomial space.

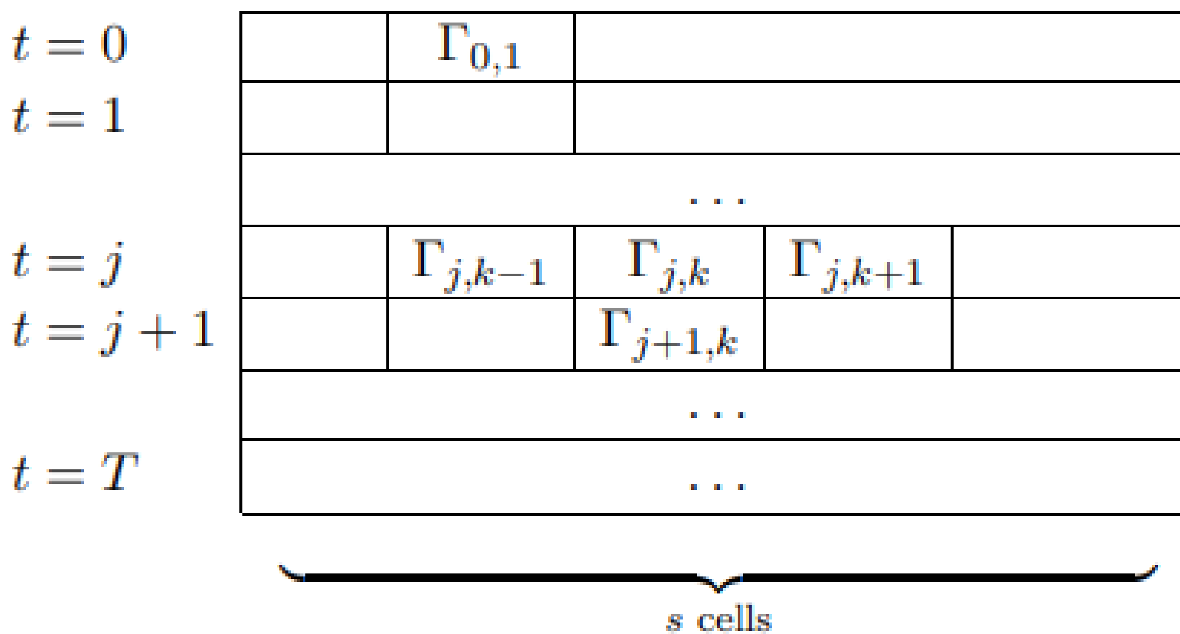
$\mathbf{P} \subset \mathbf{P/poly}$

The computation by M on some input of length n can be represented as a space-time diagram Γ that is a rectangular table of size $T \times s$, where $T = poly(n)$ and $s = poly(n)$.



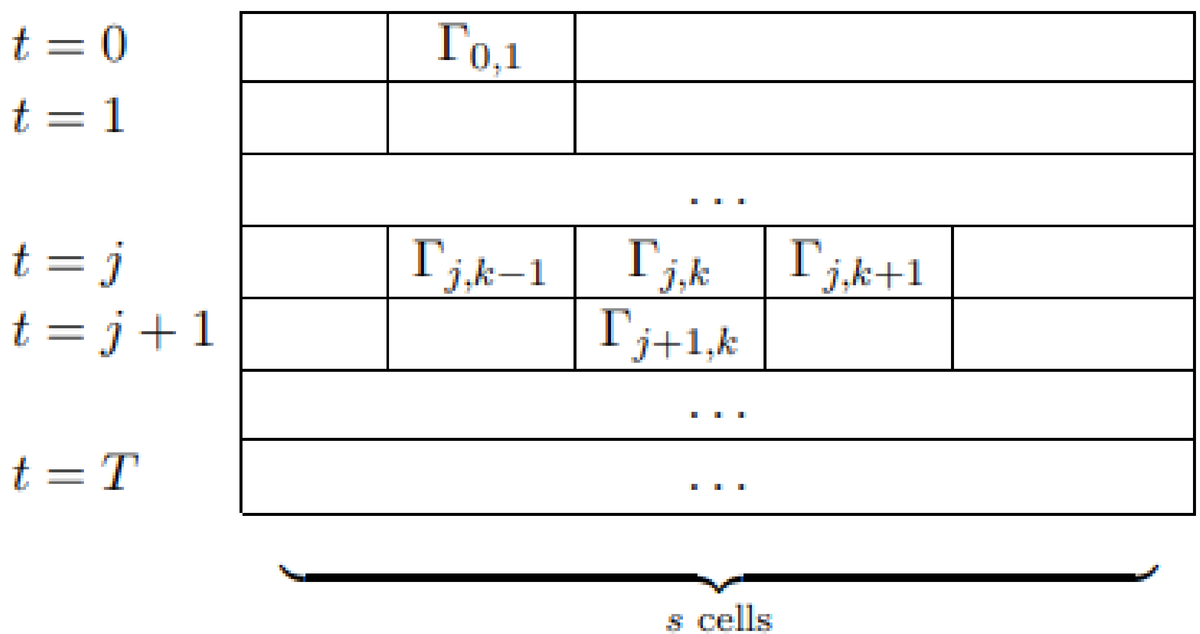
P ⊂ P/poly

$\Gamma_{j,k}$ corresponds to cell k at time j and consists of two parts: the symbol on the tape and the state of the TM if its head is in k-th cell (or a special symbol Λ if it is not). In other words, all $\Gamma_{j,k}$ belong to $S \times (\{\Lambda\} \cup Q)$.



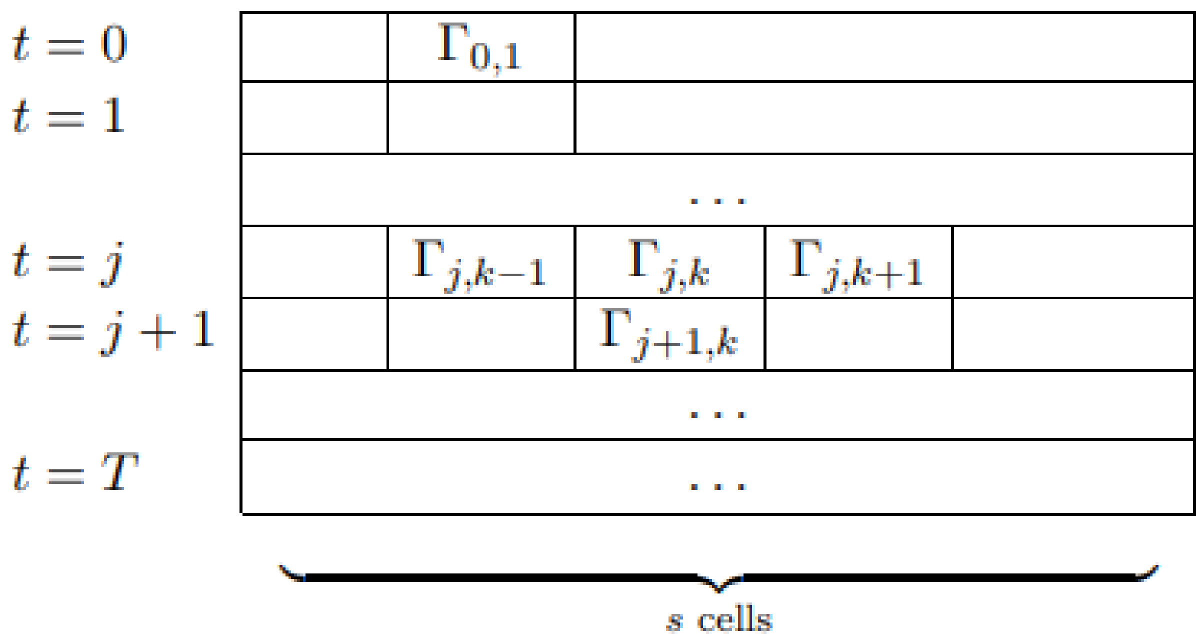
P ⊂ P/poly

There are local rules that determine the contents of a cell $\Gamma_{j+1,k}$ if we know the contents of three neighboring cells in row j, i.e., $\Gamma_{j,k-1}$, $\Gamma_{j,k}$ and $\Gamma_{j,k+1}$.



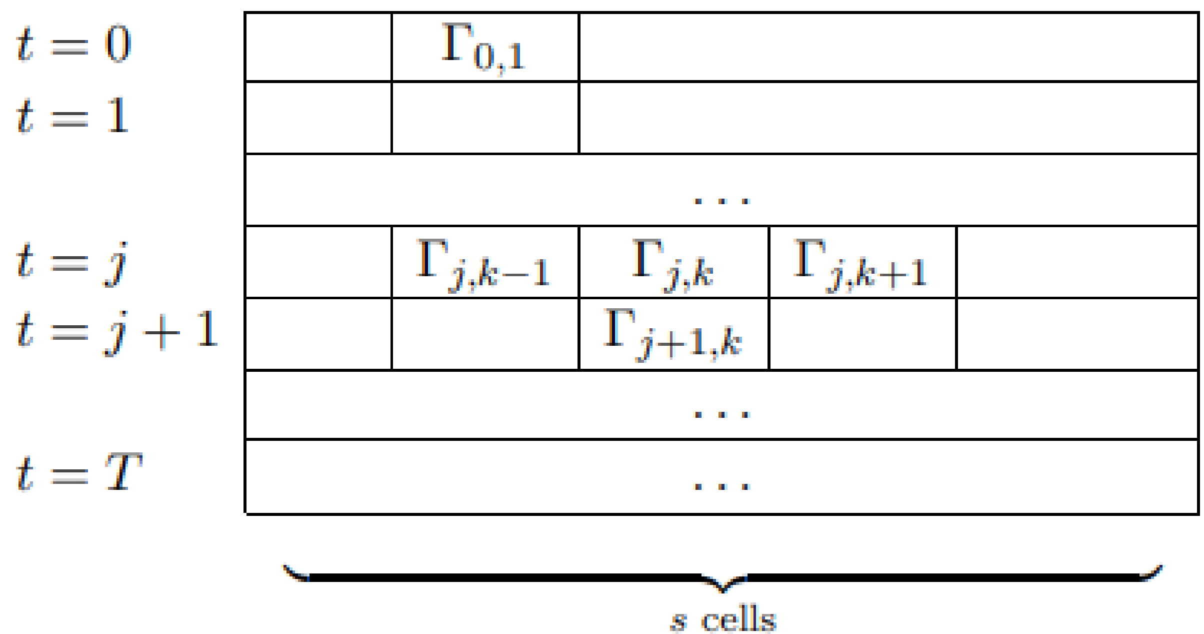
P ⊂ P/poly

We construct a circuit that computes $F(x)$ for inputs x of length n. The contents of each table cell can be encoded by a constant (i.e., independent of n) number of Boolean variables.



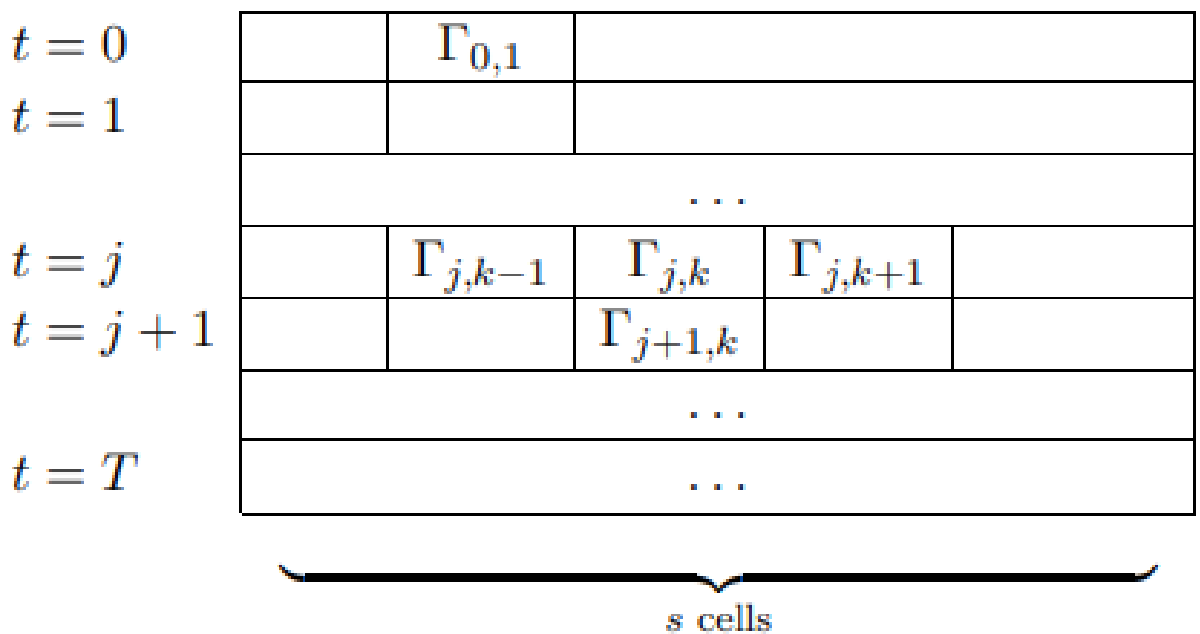
$P \subset P/poly$

Each variable encoding the cell $\Gamma_{j+1,k}$ depends only on the variables that encode $\Gamma_{j,k-1}$, $\Gamma_{j,k}$ and $\Gamma_{j,k+1}$. This dependence is a Boolean function and can be computed by circuits of size $O(1)$.



P ⊂ P/poly

Combining these circuits, we obtain a circuit that computes all of the variables which encode the state of every cell. The size of this circuit is $O(sT)O(1) = poly(n)$, where s – space of TM, T – time of TM.

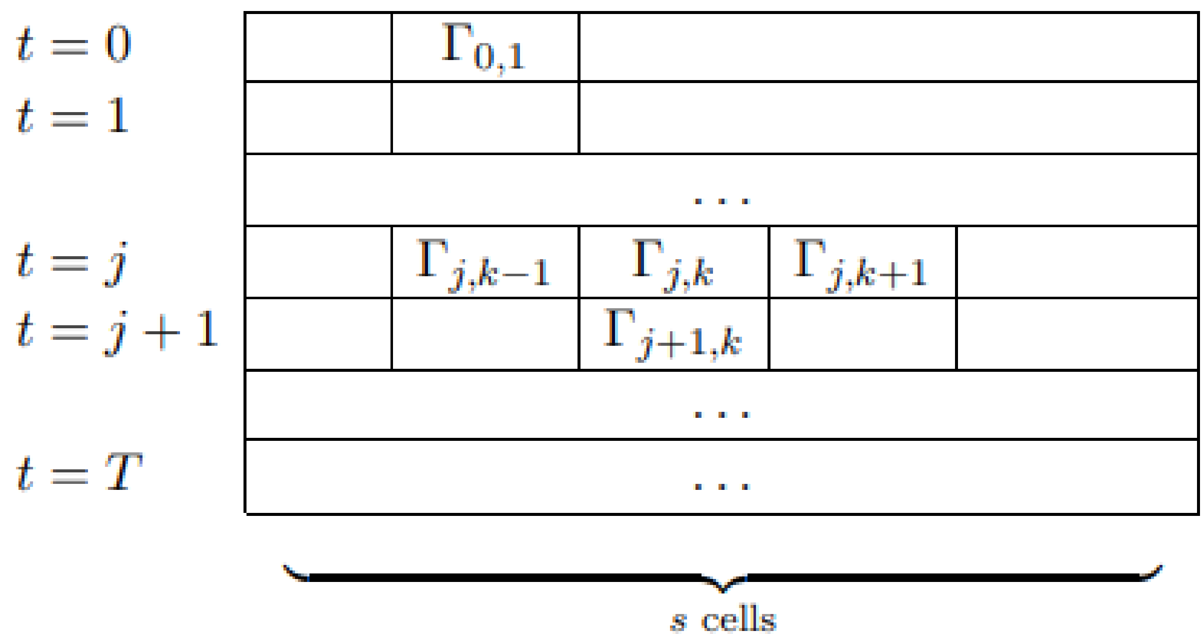


P ⊂ P/poly

Row 0 – $poly(n)$ assignments of n input bits.

Output can be computed in last row, size $O(1)$.

We get a $poly(n)$-size circuit that simulates the behavior of M for inputs of length n -> computes F_n.



P/poly vs P

The class P/poly is bigger than P.

Let ϕ: $N \to B$ be an arbitrary function.

Predicate F_ϕ such that $F_\phi(x) = \phi(|x|)$.

The restriction of F_ϕ to strings of length n is a constant function (0 or 1), so the circuit complexity of $(F_\phi)_n$ is $O(1)$.

F_ϕ for any ϕ belongs to P/poly, although for a noncomputable ϕ the predicate F_ϕ is not computable and thus does not belong to P.

$|x|$ - length of string x.

P/poly vs P

P/poly seems to be a good approximation of P for many purposes. Indeed, the class P/poly is relatively small: out of 2^{2^n} Boolean functions in n variables only $2^{poly(n)}$ functions have polynomial circuit complexity.

n variables -> 2^n rows in a truth table, each row can have 0 or 1 -> total 2^{2^n} possibilities.

x_1	x_2	F(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	1

More about 2^{2^n} possibilities

Suppose that our Boolean function has n variables.

Then, to each variable we can assign value 0 or 1, so 2 possibilities for each value of the variable. In total we get 2^n possible assignments. We can write it as a table with 2^n rows.

For each assignment (row) there are two possible outputs – 0 or 1. For 2^n rows it means that there are possible 2^{2^n} different combinations of outputs, each representing different function.

In this example we have $n = 2$, so we have 4 possible assignments to variables, which makes possibility of 16 different output combinations.

x_1	x_2	F(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	1

P/poly vs P

The difference between uniform and nonuniform computation is more important for bigger classes.

EXPTIME, the class of predicates decidable in time $2^{poly(n)}$, is a nontrivial computational class. However, the nonuniform analog of this class includes all predicates!

We can encode all possibilities in our large (exponential) circuits.

Alternative P theorem

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.

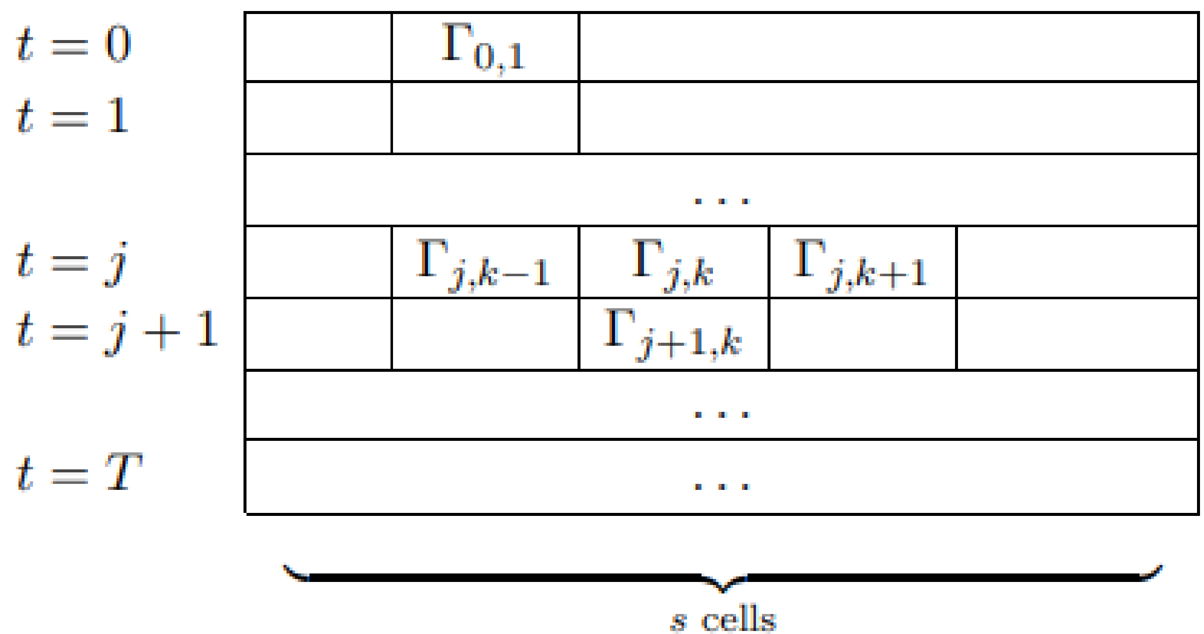
Alternative P theorem

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.

TM is not running in polynomial time since its running time is polynomial in n but not in $logn$ (the number of bits in the binary representation of n). Note also that we implicitly use some natural encoding for circuits when saying "TM constructs a circuit".

P theorem - proof

The circuit for computing F_n has regular structure, and it is clear that the corresponding sequence of assignments can be produced in polynomial time when n is known.



P theorem - proof

We compute the size of the input string x, then apply the TM to construct a circuit $C_{|x|}$ that computes $F_{|x|}$. Then we perform the assignments indicated in $C_{|x|}$, using x as the input, and get $F(x)$. All these computations can be performed in polynomial (in $|x|$) time.

P/poly vs P

For any function $\phi: N \to \{0,1\}$ the predicate $f_\phi(x) = \phi(|x|)$ belongs to P/poly. Now let ϕ be a computable function that is difficult to compute: no TM can produce output $\phi(n)$ in polynomial (in n) time. More precisely, we use a computable function ϕ such that for any TM M and any polynomial p with integer coefficients there exists n such that $M(1^n)$ does not produce $\phi(n)$ after $p(n)$ steps.

P/poly vs P

Diagonalization: we consider pairs (M, p) one by one; for each pair we select some n for which $\phi(n)$ is not defined yet and define $\phi(n)$ to be different from the result of $p(n)$ computation steps of M on input 1^n. (If computation does not halt after $p(n)$ steps, the value $\phi(n)$ can be arbitrary.)

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.

Thank you for your attention!