Computational complexity

lecture 4

Hierarchy theorems (previous lecture)

Space hierarchy theorem:

If:

- function $g(n)$ is space-constructible, and
- $f(n)=o(g(n))$

then DSPACE($f(n)$)≠DSPACE($g(n)$)

Time hierarchy theorem:

If:

- function $g(n)$ is time-constructible,
- $f(n)=o(g(n))$

then DTIME($f(n)$)≠DTIME($g(n)\log(g(n))$)

Gap theorems

- Functions being complexities of problems are distributed “quite densely”
- Simultaneously, we have the following gap theorems:

There is a computable function $f(n)\geq n$ such that DTIME($f(n)$)=DTIME($2^{f(n)}$).

There is a computable function $f(n)$ such that DSPACE($f(n)$)
=DSPACE($2^{f(n)}$).

A contradiction with hierarchy theorems?

No – the function f will not be constructible (it can be computed, but in a larger time / space)

At the same time: we see that in the hierarchy theorems the assumption about constructability is really needed

Gap theorems (✶)

Gap theorem – time

There is a computable function $f(n) \geq n$ such that DTIME($f(n)$)=DTIME($2^{f(n)}$).

Proof

Fix an input alphabet $\Sigma=\{0,1\}$ (another alphabet → time multiplied by a constant)

We construct a function $f(n)$ such that no machine stops between $f(n)$ and $2^{f(n)}$ steps:

- Assign numbers to Turing machines (in a computable way)

Gap theorems (✶)

Gap theorem – time

There is a computable function $f(n)\geq n$ such that DTIME($f(n)$)=DTIME($2^{f(n)}$).

Proof

Fix an input alphabet $\Sigma=\{0,1\}$ (another alphabet → time multiplied by a constant)

We construct a function $f(n)$ such that no machine stops between $f(n)$ and $2^{f(n)}$ steps:

- Assign numbers to Turing machines (in a computable way)
- We say that $P(n,k)$ is satisfied iff none among the first n machines on none among inputs of length n stops between k and $n\cdot 2^k$ steps (they stop earlier than k or later than $n\cdot 2^k$ or loop forever)

Gap theorems (✶)

Gap theorem – time

There is a computable function $f(n) \geq n$ such that DTIME($f(n)$)=DTIME($2^{f(n)}$).

Proof

Fix an input alphabet $\Sigma=\{0,1\}$ (another alphabet → time multiplied by a constant)

We construct a function $f(n)$ such that no machine stops between $f(n)$ and $2^{f(n)}$ steps:

- Assign numbers to Turing machines (in a computable way)
- We say that $P(n,k)$ is satisfied iff none among the first n machines on none among inputs of length n stops between k and $n \cdot 2^k$ steps (they stop earlier than k or later than $n \cdot 2^k$ or loop forever)
- Let $k_1(n)=n$ and $k_{m+1}(n)=n \cdot 2^{k_m(n)}$
- For a fixed n, every pair (input_of_length_n, machine_with_number_$\leq n$) can falsify $P(n,k_m(n))$ for at most one m,

  Thus there exists some $m \leq n \cdot 2^n$ such that $P(n,k_m(n))$ is true.

Gap theorems (✶)

Gap theorem – time

There is a computable function $f(n) \geq n$ such that DTIME($f(n)$)=DTIME($2^{f(n)}$).

Proof

Fix an input alphabet $\Sigma = \{0,1\}$ (another alphabet → time multiplied by a constant)

We construct a function $f(n)$ such that no machine stops between $f(n)$ and $2^{f(n)}$ steps:

- Assign numbers to Turing machines (in a computable way)
- We say that $P(n,k)$ is satisfied iff none among the first n machines on none among inputs of length n stops between k and $n \cdot 2^k$ steps (they stop earlier than k or later than $n \cdot 2^k$ or loop forever)
- Let $k_1(n)=n$ and $k_{m+1}(n)=n \cdot 2^{k_m(n)}$
- For a fixed n, every pair (input_of_length_n, machine_with_number_$\leq n$) can falsify $P(n,k_m(n))$ for at most one m,

  Thus there exists some $m \leq n \cdot 2^n$ such that $P(n,k_m(n))$ is true.
- We put $f(n)=k_m(n)$ for this value of m. This function is computable.

Gap theorems (✶)

Gap theorem – time

There is a computable function $f(n) \geq n$ such that DTIME($f(n)$)=DTIME($2^{f(n)}$).

Proof

- For every n, none among the first n machines on none among inputs of length n stops between $f(n)$ and $n \cdot 2^{f(n)}$ steps.
- Take any machine M with number m running in time $c \cdot 2^{f(n)}$
- For every input of length $n \geq max(m,c)$ the machine stops in $\leq c \cdot 2^{f(n)}$ steps, but not between $f(n)$ and $n \cdot 2^{f(n)}$ steps, hence in $\leq f(n)$ steps

Gap theorems (✶)

Gap theorem – time

There is a computable function $f(n) \geq n$ such that DTIME($f(n)$)=DTIME($2^{f(n)}$).

Proof

- For every n, none among the first n machines on none among inputs of length n stops between $f(n)$ and $n \cdot 2^{f(n)}$ steps.
- Take any machine M with number m running in time $c \cdot 2^{f(n)}$
- For every input of length $n \geq max(m,c)$ the machine stops in $\leq c \cdot 2^{f(n)}$ steps, but not between $f(n)$ and $n \cdot 2^{f(n)}$ steps, hence in $\leq f(n)$ steps
- There are only constantly many inputs of length $< max(m,c)$
- Thus the language can be recognized in time $O(f(n))$

Gap theorems

Remarks

- In the same way we can construct a function f such that DSPACE($f(n)$)=DSPACE($2^{f(n)}$) (Sipser's theorem needed here).
- Actually, for every computable function g such that $g(n)\geq n$ (instead of $g(n)=2^n$) we can find f a such that DTIME($f(n)$)=DTIME($g(f(n))$) or DSPACE($f(n)$)=DSPACE($g(f(n))$).
- The functions f grow very quickly.
- They are not time/space-constructible.
- But they are computable.

Just finished:

Deterministic Turing machines – basic facts

Next topic:

Boolean circuits

Later:

- Nondeterministic Turing machines, reductions
- Probabilistic computations
- Fixed parameter tractability (FPT)
- Interactive proofs
- Alternating Turing machines
- Probabilistically checkable proofs (PCP)
- ...

Nonuniform computation models

- Suppose that P≠NP. Then there is no algorithm which quickly solves all instances of the SAT problem.
- But maybe for every n there is a separate algorithm, which quickly solves all instances of size n?
- Even if these algorithms are difficult to find, this would mean that SAT can be solved in practice.

Nonuniform computation models

- Suppose that P≠NP. Then there is no algorithm which quickly solves all instances of the SAT problem.
- But maybe for every n there is a separate algorithm, which quickly solves all instances of size n?
- Even if these algorithms are difficult to find, this would mean that SAT can be solved in practice.
- A similar example: breaking the cryptographic algorithm RSA.
  If there is an algorithm, which quickly breaks the RSA encoding for a fixed (being currently used) key length, in practice we can treat the RSA code as insecure (even if the algorithm works only for one fixed n, not for all n).

Nonuniform computation models

- Suppose that P≠NP. Then there is no algorithm which quickly solves all instances of the SAT problem.
- But maybe for every n there is a separate algorithm, which quickly solves all instances of size n?
- Even if these algorithms are difficult to find, this would mean that SAT can be solved in practice.
- A similar example: breaking the cryptographic algorithm RSA. If there is an algorithm, which quickly breaks the RSA encoding for a fixed (being currently used) key length, in practice we can treat the RSA code as insecure (even if the algorithm works only for one fixed n, not for all n).

Hence, it makes sense to consider computation models in which for every n we apply a different algorithm.

One has to be careful, though: for every n, the language of instances of size n is regular.

Models of parallel computations

What if we have plenty of processors?

Example: matrix multiplication

- 1 processor: time $O(n^3)$ (the standard algorithm)
- n^2 processors: time $O(n)$
- n^3 processors: time $O(log(n))$ – an exponential speed up!

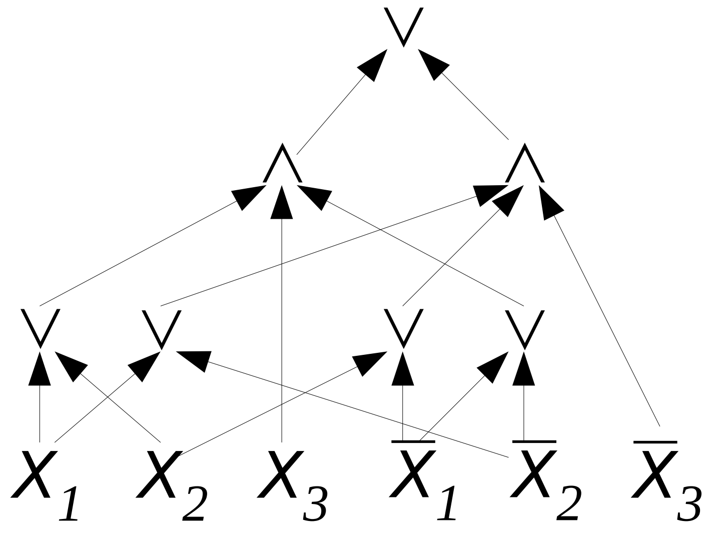
Question: Which algorithms do parallelize well, and which do not?

Boolean circuits

Another computational model: boolean circuits

idea: computing boolean functions using logical gates

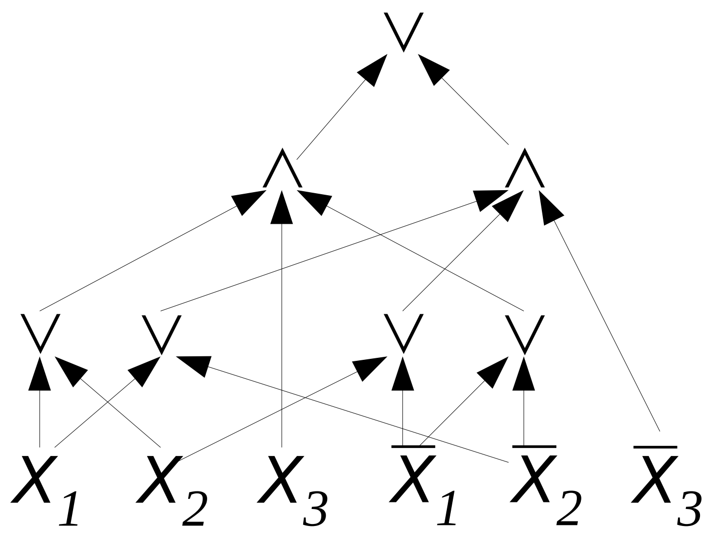
intuition: every gate represents a very simple processor


$\vee$
$\wedge$ $\wedge$
$\vee$ $\vee$ $\vee$ $\vee$
x_1 x_2 x_3 $\overline{x}_1$ $\overline{x}_2$ $\overline{x}_3$

Boolean circuits

Definition: a boolean circuit having input of size n is given by an acyclic directed graph, in which:

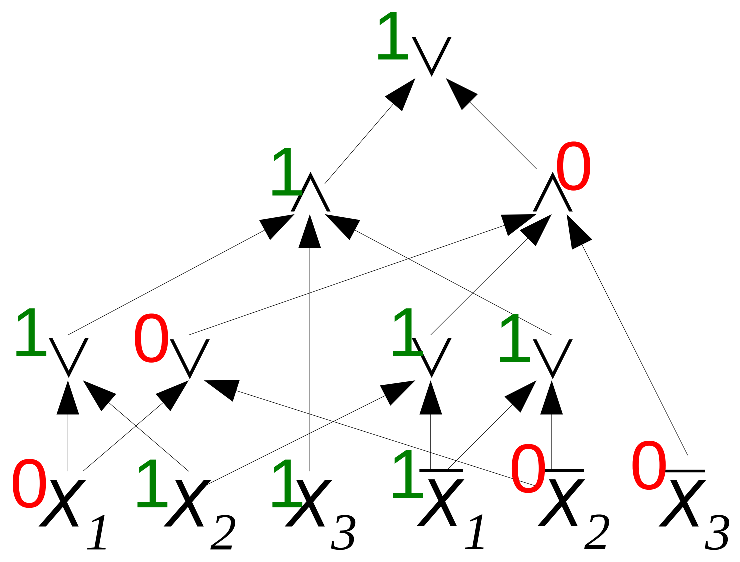
- there are $2n$ gates (nodes) of in-degree 0, denoted $X_1,\overline{X}_1,\ldots,X_n,\overline{X}_n$ (input gates)
- all other gates (having in-degree ≥ 0) are marked by one of the symbols $\wedge$ or $\vee$
- one of the gates (having out-degree 0) is marked as the output gate [another version: multiple outputs – when we compute a function]



Boolean circuits

For a fixed valuation $v:\{X_1,\ldots,X_n\}\to\{0,1\}$ we define:

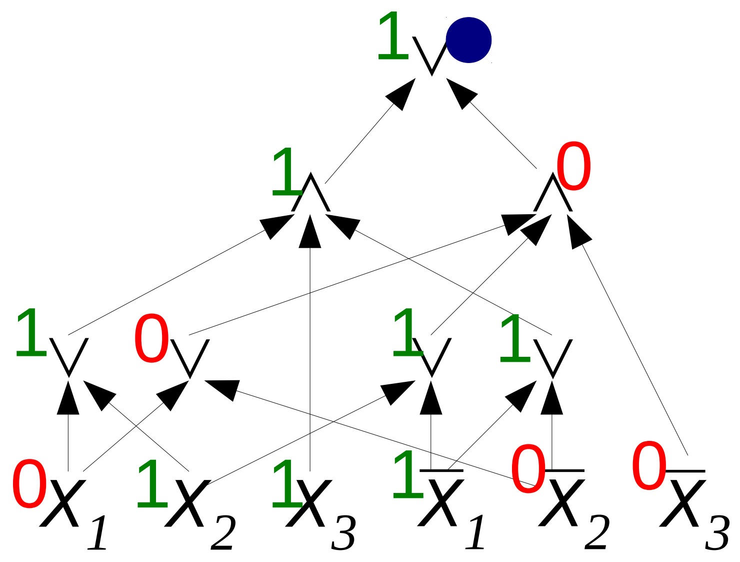
- the gate labeled by X_i gets value $v(X_i)$
- the gate labeled by $\overline{X}_i$ gets value $\neg v(X_i)$
- the value of an OR (AND) gate is computed as the disjunction (conjunction) of values of predecessors of the gate
- the value of the circuit = the value of the output gate
- the definition makes sense, because the graph is acyclic



Boolean circuits

An equivalent definition – a circuit as a game:

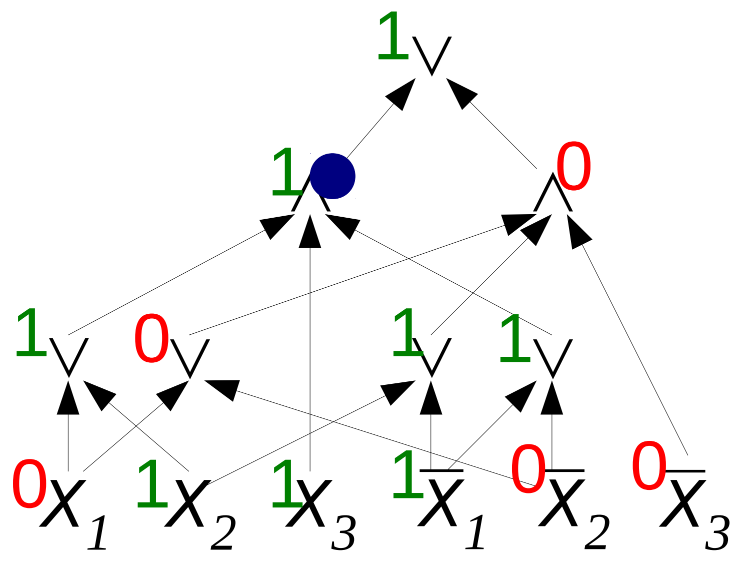
- two players (AND and OR) move a pawn over the graph, going back from the output gate
- AND (OR) decides in $\wedge$ nodes ($\vee$ nodes, respectively)
- OR wins, if the game finishes in X_i and $v(X_i)=1$, or in $\overline{X}_i$ and $v(X_i)=0$
- the value of the circuit is 1 if OR has a winning strategy



Boolean circuits

An equivalent definition – a circuit as a game:

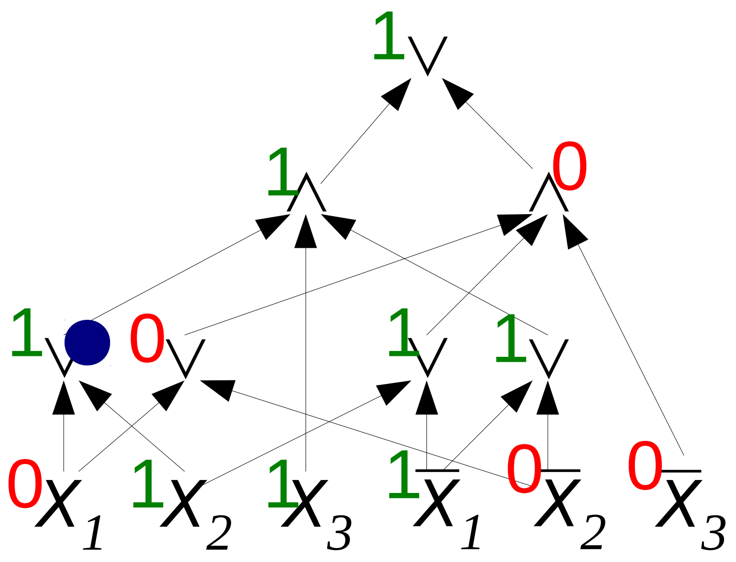
- two players (AND and OR) move a pawn over the graph, going back from the output gate
- AND (OR) decides in $\wedge$ nodes ($\vee$ nodes, respectively)
- OR wins, if the game finishes in X_i and $v(X_i)=1$, or in $\overline{X}_i$ and $v(X_i)=0$
- the value of the circuit is 1 if OR has a winning strategy



Boolean circuits

An equivalent definition – a circuit as a game:

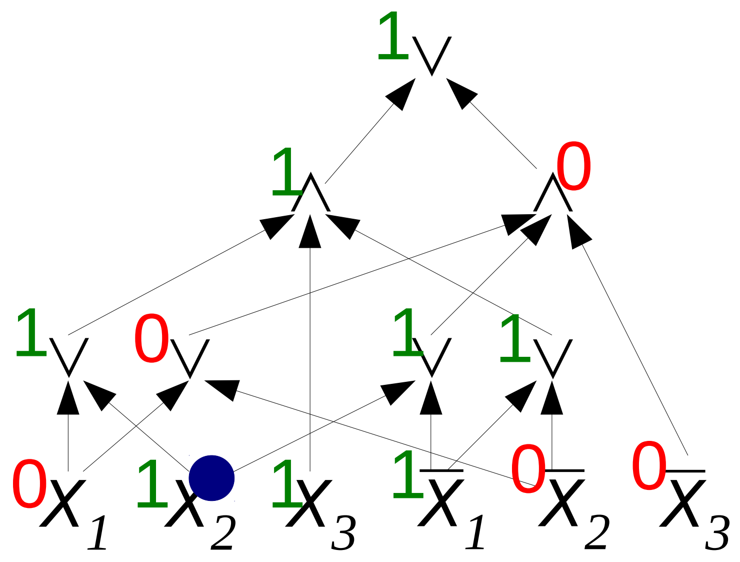
- two players (AND and OR) move a pawn over the graph, going back from the output gate
- AND (OR) decides in ∧ nodes (∨ nodes, respectively)
- OR wins, if the game finishes in X_i and $v(X_i)=1$, or in $\overline{X}_i$ and $v(X_i)=0$
- the value of the circuit is 1 if OR has a winning strategy



Boolean circuits

An equivalent definition – a circuit as a game:

- two players (AND and OR) move a pawn over the graph, going back from the output gate
- AND (OR) decides in $\wedge$ nodes ($\vee$ nodes, respectively)
- OR wins, if the game finishes in X_i and $v(X_i)=1$, or in $\overline{X}_i$ and $v(X_i)=0$
- the value of the circuit is 1 if OR has a winning strategy



Boolean circuits

Equivalence of the two definitions:

- if the output has value 1, we have a strategy for OR: descend always to a node labeled by 1
- if the output has value 0, we have a strategy for AND: descend always to a node labeled by 0

Boolean circuits

- For a fixed valuation $v:\{X_1,\ldots,X_n\}\rightarrow\{0,1\}$ we have defined the value of a circuit
- The input amounts to a word $v\in\{0,1\}^n$
- A circuit C computes a function $\{0,1\}^n\rightarrow\{0,1\}$, i.e., it recognizes a subset of $\{0,1\}^n$

Boolean circuits

Size?

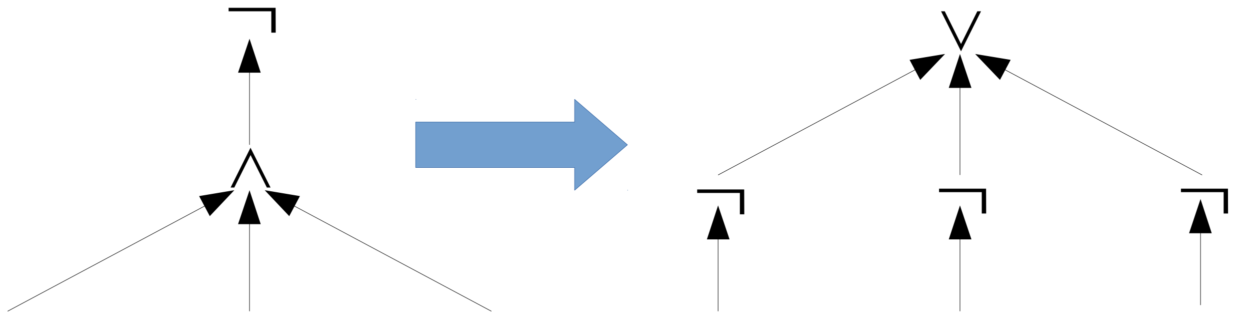
We have several parameters:

- the length of an input n
- the depth of a circuit (the length of the longest path)
- the number of gates B, the number of edges K
- the length of a representation of a circuit: $(B+K)\cdot log(B)$ (because numbers of gates have $log(B)$ bits)
- in-degree of gates (fan-in) – we consider circuits
  - ➔ with arbitrary fan-in
  - ➔ with fan-in ≤ 2

Boolean circuits

Negations?

- in our definition there are no NOT gates, but we have negated input gates
- this does not change anything: negations can be easily moved to leaves (De Morgan laws)



Boolean circuits

<u>Recognizing languages by sequences of circuits:</u>

- A circuit C_n having input of size n recognizes $L(C_n)$ – a subset of $\{0,1\}^n$ [in particular C_0 has no inputs, returns always 1 or always 0]
- Having a sequence of circuits $C_0, C_1, C_2, \ldots$ we can recognize a language containing words of any length:
  $$L((C_n)_{n\in\mathbb{N}}) = L(C_0) \cup L(C_1) \cup L(C_2) \cup \ldots$$
- What languages can be recognized using boolean circuits?

Boolean circuits

Recognizing languages by sequences of circuits:

- A circuit C_n having input of size n recognizes $L(C_n)$ – a subset of $\{0,1\}^n$ [in particular C_0 has no inputs, returns always 1 or always 0]
- Having a sequence of circuits $C_0,C_1,C_2,\ldots$ we can recognize a language containing words of any length:
  $$L((C_n)_{n\in\mathbb{N}})=L(C_0)\cup L(C_1)\cup L(C_2)\cup\ldots$$
- What languages can be recognized using boolean circuits?

Fact.
Every language can be recognized by some sequence of boolean circuits (having depth 2 and exponential size)

i.e., the size of C_n is exponential in n

Boolean circuits

<u>Recognizing languages by sequences of circuits:</u>

- A circuit C_n having input of size n recognizes $L(C_n)$ – a subset of $\{0,1\}^n$ [in particular C_0 has no inputs, returns always 1 or always 0]
- Having a sequence of circuits $C_0,C_1,C_2,\ldots$ we can recognize a language containing words of any length:

$$L((C_n)_{n\in\mathbb{N}})=L(C_0)\cup L(C_1)\cup L(C_2)\cup\ldots$$

- What languages can be recognized using boolean circuits?

<u>Fact</u>.
Every language can be recognized by some sequence of boolean circuits (having depth 2 and exponential size)

A more interesting question: Which languages can be recognized by a sequence of circuits of polynomial size?

Simulating machines by circuits

Theorem

Every language recognizable in time $T(n)$ on a single-tape machine can be recognized by a sequence of circuits $(C_n)_{n\in\mathbb{N}}$ of depth $O(T(n))$ and number of gates $O((T(n))^2)$.

(actually, a stronger variant can be proven: depth $O(T(n))$ and $O(T(n)\cdot log(T(n)))$ gates, even for a multi-tape machine)

Additionally, the circuit C_n can be generated in logarithmic space (thus: in polynomial time) in n. (i.e., there exists a TM working in logarithmic space, which on input 1^n outputs a representation of the circuit C_n)

Simulating machines by circuits

<u>Theorem</u>

Every language recognizable in time $T(n)$ on a single-tape machine can be recognized by a sequence of circuits $(C_n)_{n\in\mathbb{N}}$ of depth $O(T(n))$ and number of gates $O((T(n))^2)$.

<u>Proof</u>

- Fix some M recognizing our language in time $T(n)$; fix also some n.
- We can assume that runs of M on words of length n have length precisely $T(n)$ (if M stops earlier, we repeat the last configuration).
- M uses at most $T(n)$ tape cells.
- A computation of M can be written in a square $T(n)\times T(n)$

Simulating machines by circuits

A computation of M can be written in a square $T(n)\times T(n)$:

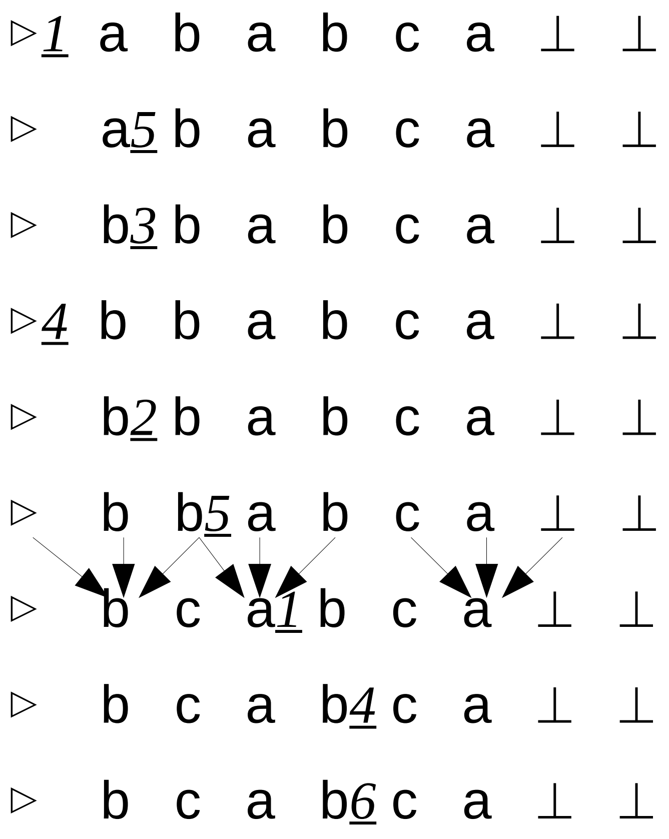
- Every row consists of a tape contents in some step
- In the cell over which the head is located, we additionally write the state.

▷$\underline{1}$	a	b	a	b	c	a	⊥	⊥
▷	a$\underline{5}$	b	a	b	c	a	⊥	⊥
▷	b$\underline{3}$	b	a	b	c	a	⊥	⊥
▷$\underline{4}$	b	b	a	b	c	a	⊥	⊥
▷	b$\underline{2}$	b	a	b	c	a	⊥	⊥
▷	b	b$\underline{5}$	a	b	c	a	⊥	⊥
▷	b	c	a$\underline{1}$	b	c	a	⊥	⊥
▷	b	c	a	b$\underline{4}$	c	a	⊥	⊥
▷	b	c	a	b$\underline{6}$	c	a	⊥	⊥

Simulating machines by circuits

A computation of M can be written in a square $T(n) \times T(n)$:

- Every row consists of a tape contents in some step
- In the cell over which the head is located, we additionally write the state.
- The content of a cell depends only on the three cells located directly over it.



▷<u>*1*</u>	a	b	a	b	c	a	⊥	⊥
▷	a<u>*5*</u>	b	a	b	c	a	⊥	⊥
▷	b<u>*3*</u>	b	a	b	c	a	⊥	⊥
▷<u>*4*</u>	b	b	a	b	c	a	⊥	⊥
▷	b<u>*2*</u>	b	a	b	c	a	⊥	⊥
▷	b	b<u>*5*</u>	a	b	c	a	⊥	⊥
▷	b	c	a<u>*1*</u>	b	c	a	⊥	⊥
▷	b	c	a	b<u>*4*</u>	c	a	⊥	⊥
▷	b	c	a	b<u>*6*</u>	c	a	⊥	⊥

Simulating machines by circuits

A computation of M can be written in a square $T(n)\times T(n)$:

- Every row consists of a tape contents in some step
- In the cell over which the head is located, we additionally write the state.
- The content of a cell depends only on the three cells located directly over it.
- Gate (i,j,z) – in the cell having coordinates i,j there is z
- The value of a gate (i,j,z) is a function of gates $(i\text{-}1,j\text{-}1,z')$, $(i\text{-}1,j,z')$, $(i\text{-}1,j+1,z')$ for all z' – it can be realized by a circuit of a constant size (the number of possible z,z' is fixed – independent on n)
- Output gate: in the last row there is an accepting state
- Details in notes of D.Niwiński

Simulating machines by circuits

Is it the case that every language recognizable by a sequence of circuits can be recognized by a Turing machine?

Simulating machines by circuits

Is it the case that every language recognizable by a sequence of circuits can be recognized by a Turing machine?

NO! – circuits need not to be uniform

(a sequence of circuits can recognize an arbitrary language,
a Turing machine cannot)

Simulating machines by circuits

Is it the case that every language recognizable by a sequence of circuits can be recognized by a Turing machine?

NO! – circuits need not to be uniform

(a sequence of circuits can recognize an arbitrary language,
a Turing machine cannot)

A theorem which is true:

There is a Turing machine (working in quadratic time), which inputs a representation of a circuit C_n and a word of w of length n, and computes the value of C_n on word w.

Turing machines with advice

A <u>Turing machine with advice</u> – a model that is non-uniform, but sequential.

Definition: A machine M together with a sequence of words $k_0, k_1, k_2, \ldots$ recognizes a language L iff

$$w \in L \Leftrightarrow k_{|w|}\$w \in L(M)$$

Turing machines with advice

A <u>Turing machine with advice</u> – a model that is non-uniform, but sequential.

Definition: A machine M together with a sequence of words $k_0, k_1, k_2, \ldots$ recognizes a language L iff

$$w \in L \Leftrightarrow k_{|w|}\$w \in L(M)$$

We consider the running time with respect to $|w|$, not with respect to the whole word.

E.g. an exponential advice enforces exponential running time (it is necessary to read it).

Turing machines with advice

A Turing machine with advice – a model that is non-uniform, but sequential.

Definition: A machine M together with a sequence of words $k_0, k_1, k_2, \ldots$ recognizes a language L iff

$$w \in L \Leftrightarrow k_{|w|}\$w \in L(M)$$

We consider the running time with respect to $|w|$, not with respect to the whole word.

E.g. an exponential advice enforces exponential running time (it is necessary to read it).

class **P/poly** – languages recognizable in polynomial time by a machine with advice (of polynomial size)

Turing machines with advice

class **P/poly** – languages recognizable in polynomial time by a machine with advice (of polynomial size)

Theorem

A language belongs to **P/poly** iff it is recognizable by a sequence of circuits of polynomial size.

Proof

Turing machines with advice

class **P/poly** – languages recognizable in polynomial time by a machine with advice (of polynomial size)

Theorem

A language belongs to **P/poly** iff it is recognizable by a sequence of circuits of polynomial size.

Proof

$\Rightarrow$ We convert the machine to a circuit.
The advice can be hard-coded in the circuit.

Turing machines with advice

class **P/poly** – languages recognizable in polynomial time by a machine with advice (of polynomial size)

Theorem

A language belongs to **P/poly** iff it is recognizable by a sequence of circuits of polynomial size.

Proof

$\Rightarrow$ We convert the machine to a circuit.
The advice can be hard-coded in the circuit.

$\Leftarrow$ k_n consists of a representation of C_n;
we evaluate C_n using a Turing machine

Turing machines with advice

The **P/poly** class is non-uniform – it contains undecidable languages.

For example:

$L=\{1^n :$ the n-th Turing machine halts on every input$\}$

Turing machines with advice

The **P/poly** class is non-uniform – it contains undecidable languages.

For example:

$L=\{1^n$: the n-th Turing machine halts on every input$\}$

The **P/poly** class is useful for modeling languages (problems), which can be solved quickly after a (probably very costly) preprocessing.
E.g., in cryptography one sometimes assumes that an intruder has computing power in **P/poly**.

Turing machines with advice

The **P/poly** class is non-uniform – it contains undecidable languages.

For example:

$L=\{1^n$: the n-th Turing machine halts on every input$\}$

The **P/poly** class is useful for modeling languages (problems), which can be solved quickly after a (probably very costly) preprocessing.
E.g., in cryptography one sometimes assumes that an intruder has computing power in **P/poly**.

Open problem: does **NP**$\not\subseteq$**P/poly**?

(this is a stronger statement than P$\neq$NP, because obviously P$\subseteq$P/poly)

Uniform sequences of circuits

A sequence of circuits $C_0,C_1,C_2,\ldots$ is <u>uniform</u> if it is computable in logarithmic space, i.e., there exists a TM working in logarithmic space, which on input 1^n outputs the representation of circuit C_n

Uniform sequences of circuits

A sequence of circuits $C_0, C_1, C_2, \ldots$ is uniform if it is computable in logarithmic space, i.e., there exists a TM working in logarithmic space, which on input 1^n outputs the representation of circuit C_n

Let us recall the definition – functions computable in logarithmic space:

- a read-only input tape
- working tapes of logarithmic length
- an output tape, over which the head may only move right

Notice that in logarithmic space one can compute an output which is much longer than logarithmic (but necessarily is polynomial)

Corollary: such a procedure can only generate circuits C_n that are of size polynomial in n.

Uniform sequences of circuits

A sequence of circuits $C_0,C_1,C_2,\ldots$ is <u>uniform</u> if it is computable in logarithmic space, i.e., there exists a TM working in logarithmic space, which on input 1^n outputs the representation of circuit C_n

Let us recall the definition – functions computable in logarithmic space:

- a read-only input tape
- working tapes of logarithmic length
- an output tape, over which the head may only move right

Notice that in logarithmic space one can compute an output which is much longer than logarithmic (but necessarily is polynomial)

Theorem
Functions computable in logarithmic space are closed under composition.

<u>Proof</u>
When the second TM wants to read the k-th bit of the output of the first machine, then we run the first TM, and we only check the value of the k-th bit of its output, ignoring the rest of the output.

Uniform sequences of circuits

Theorem

A language is recognizable by a uniform sequence of circuits iff it is in **P**.

Proof

⇒ obvious: having an input word of length n generate the n-th circuit, and compute its value

⇐ the algorithm given previously, which constructs a circuit basing on a Turing machine and on the input length n, works in logarithmic space (it only has to remember for which cell of the square it currently outputs gates; this fits in a logarithmic space)