Computational complexity

lecture 14

Today

- approximation
- complexity & cryptography
- zero-knowledge proofs
- quantum computing

Approximation

We have already seen:

19.12 – Approximation of VERTEX-COVER with factor 1/2:

In a loop – add both ends of some edge to the cover, and remove them from the graph, together with their neighbors

Every cover contains at least one of these ends, thus our cover is at most twice larger (error 1/2)

Approximation

We have already seen:

19.12 – Approximation of VERTEX-COVER with factor 1/2

16.01 – PCP theorem:

- MAX-CLIQUE – approximation almost impossible
- MAX-3CNFSAT – approximation possible with factor 1/8, better approximation impossible

Approximation

We have already seen:

19.12 – Approximation of VERTEX-COVER with factor 1/2

PCP → approximation factor cannot be better than 0.265

16.01 – PCP theorem:

- MAX-CLIQUE – approximation almost impossible
- MAX-3CNFSAT – approximation possible with factor 1/8, better approximation impossible

Approximation

We have already seen:

19.12 – Approximation of VERTEX-COVER with factor 1/2

PCP → approximation factor cannot be better than 0.265

16.01 – PCP theorem:

- MAX-CLIQUE – approximation almost impossible
- MAX-3CNFSAT – approximation possible with factor 1/8, better approximation impossible

Today:

- traveling salesman – approximation almost impossible (no PCP thm.)
- knapsack problem – strong PTAS (very good approximation)

Difficult approximation

Traveling salesmen problem: the approximation factor is 1 (there is no effective approximation at all), unless **P=NP**.

Proof:

Suppose that there is an algorithm with error $\varepsilon<1$. Out of it, we will create a precise algorithm finding Hamiltonian cycles. For an arbitrary graph we create an instance of the traveling salesman problem: as the distance between nodes u, v we take:

- 1 if there is an edge between u and v in the original graph
- $|V|/(1-\varepsilon)$ if there is no such edge

Difficult approximation

Traveling salesmen problem: the approximation factor is 1 (there is no effective approximation at all), unless **P=NP**.

Proof:

Suppose that there is an algorithm with error $\varepsilon<1$. Out of it, we will create a precise algorithm finding Hamiltonian cycles. For an arbitrary graph we create an instance of the traveling salesman problem: as the distance between nodes u, v we take:

- 1 if there is an edge between u and v in the original graph
- $|V|/(1-\varepsilon)$ if there is no such edge

We run the approximate algorithm of the traveling salesmen problem on this instance. There are two possibilities:

- the algorithm gives a route having cost $|V|$; then we have a Hamiltonian cycle
- the algorithm gives a route having cost $C>|V|/(1-\varepsilon)$ (when at least one edge of such a cost is used); the error is ε, so $C\leq O/(1-\varepsilon)$ (where O – optimal cost), so $O>|V|$ – there is no Hamiltonian cycle

Difficult approximation

Suppose that there is an algorithm with error $\varepsilon<1$. Out of it, we will create a precise algorithm finding Hamiltonian cycles. For an arbitrary graph we create an instance of the traveling salesman problem: as the distance between nodes u, v we take:

- 1 if there is an edge between u and v in the original graph
- $|V|/(1-\varepsilon)$ if there is no such edge

We run the approximate algorithm of the traveling salesmen problem on this instance. There are two possibilities:

- the algorithm gives a route having cost $|V|$; then we have a Hamiltonian cycle
- the algorithm gives a route having cost $C>|V|/(1-\varepsilon)$ (when at least one edge of such a cost is used); the error is ε, so $C\leq O/(1-\varepsilon)$ (where O – optimal cost), so $O>|V|$ – there is no Hamiltonian cycle

Remark: This shows not only that there is no approximation with a constant factor, but even such that $1-\varepsilon$ goes exponentially to 0 (i.e., where $C/O\leq$exponential_function), since then the instance of the traveling salesmen problem is still of polynomial size

Easy approximation (strong PTAS)

Knapsack problem:

Input: n pairs (v_i,w_i) – there are n objects, i-th of them has value v_i and weight w_i

threshold $W_{\max}$.

Output: a subset of these objects, such that the sum of v_i is as large as possible, but the sum of w_i does not exceed $W_{\max}$.

For this problem: approximation factor is 0, there is a strong PTAS – for every $\varepsilon>0$ there is an algorithm that approximates the solution with error ε, and works in time $O(n^3)$ [proof: next slides]

Easy approximation (strong PTAS)

Solution:

The problem can be solved by dynamic programming (which is polynomial time assuming that v_i are given in unary):

Let $V=max(v_i)$.

For every $i=0...n$ and $v=0...nV$ we compute $W(i,v)$ – the minimal weight, which can be obtained by choosing among the first i objects so that the value is precisely v.

We have a recursive formula:

$$W(i+1,v)=min(W(i,v), W(i,v-v_{i+1})+w_{i+1})$$

Running time: $O(n^2V)$ (formally: times logarithmic factors)

Since V is given in binary, this is not polynomial in the size of the input.

BTW. We see that the knapsack problem is FPT with parameter V

BTW2. Another algorithm (also dynamic programming) works in time $O(nW_{\max})$. So the problem is FPT also with parameter $W_{\max}$

Easy approximation (strong PTAS)

Thus, for $V=max(v_i)$ we have an exact algorithm in time $O(n^2V)$.

We use it as follows: for some b we take $v'_i=\lfloor v_i/2^b \rfloor$ (i.e., we cut off the last b bits of every value). Now the running time is $O(n^2V/2^b)$.

How precise is the solution?

Let S and S' be the optimal choices for v_i and v'_i. It holds that:

$$\sum_{i\in S} v_i \geq \sum_{i\in S'} v_i \geq \sum_{i\in S'} 2^b v'_i \geq \sum_{i\in S} 2^b v'_i \geq \sum_{i\in S} v_i - n2^b$$

Thus S' approximates the optimal solution of the original problem with error $\leq \varepsilon = n2^b/V$ (notice that V is a lower bound for the solution, assuming that no weight exceeds $W_{\max}$).

For a fixed ε we take $b=\lceil log(\varepsilon V/n) \rceil$. Then the algorithms gives error $\leq \varepsilon$, and the running time is $O(n^2V/2^b)=O(n^3/\varepsilon)$

Complexity and cryptography

- Basic goal of cryptography: encode a message so that the adversary cannot decode it
- We should think about the adversary as about a device with a limited computational power – for example RSA: the encoded message and the public key uniquely determine the original message, but we cannot compute it quickly
- Thus: the desired situation is that the problem of decoding messages reduces to (i.e., is not easier than) some difficult computational problem

Complexity and cryptography

- A security proof has to base at least on the **P≠NP** conjecture (or on some stronger conjecture)
  For example consider public-key cryptography (RSA): having access to nondeterminism, one can easily decode – it is enough to guess the original message, and check that after encoding it gives the encoded message (encoding is easy). Thus if **P=NP**, we can quickly decode deterministically.

Complexity and cryptography

- A security proof has to base at least on the **P≠NP** conjecture (or on some stronger conjecture)
  For example consider public-key cryptography (RSA): having access to nondeterminism, one can easily decode – it is enough to guess the original message, and check that after encoding it gives the encoded message (encoding is easy). Thus if **P=NP**, we can quickly decode deterministically.
- Initially, people tried to base security of cryptographic protocols on hardness of some **NP**-complete problem (i.e., on the **P≠NP** conjecture). Without any success, till now.
- The reason (generally): **NP**-completeness talks about hardness of problems for the worst input. But in cryptography we need problems that are hard for most inputs. When we are encoding something, it should be hard for the adversary to decode (almost) every uncoded message, not only some of them (the hardest)

Complexity and cryptography

Some hard problems, on which security of cryptographic protocols is based:

- Finding factors of composite numbers. The best known algorithm works in time $\sim 2^{O(n^{1/3})}$. It is difficult to decompose numbers of the for $p \cdot q$, where p and q are prime numbers of similar size.
- Discrete logarithm. Let g be a generator of the group $\mathbb{Z}_p^*$. Knowing $(g^x \bmod p)$ it is difficult to find x. It can be shown that if this problem is difficult in the worst case, then it is also difficult in the average case.
- And other (e.g., basing on elliptic curves)

Complexity and cryptography

Discrete logarithm. Let g be a generator of the group $\mathbb{Z}_p^*$.

Knowing $(g^x \bmod p)$ it is difficult to find x. It can be shown that if this problem is difficult in the worst case, then it is also difficult in the average case.

More precisely: If a polynomial time algorithm computes the discrete logarithm for a $1/\log(p)$ (i.e. $1/poly(size\_of\_input)$) fraction of all inputs, then there is a randomized polynomial time algorithm for discrete logarithm for all inputs.

Proof

Suppose that we want to compute the discrete logarithm of x, that is, compute k such that $x=g^k \pmod p$.

We pick a random number b, and we try to compute the discrete logarithm of xg^b – a number k' such that $xg^b=g^{k'} \pmod p$.

Then $k=k'-b$.

We succeed with probability $1/\log(p)$, i.e., $1/poly(size\_of\_input)$.

Notice that $(xg^b \bmod p)$ is distributed uniformly over $\{0,\ldots,p-1\}$.

Complexity and cryptography

Some hard problems, on which security of cryptographic protocols is based:

- Finding factors of composite numbers. The best known algorithm works in time $\sim 2^{O(n^{1/3})}$. It is difficult to decompose numbers of the for $p \cdot q$, where p and q are prime numbers of similar size.
- Discrete logarithm. Let g be a generator of the group $\mathbb{Z}_p^*$. Knowing $(g^x \bmod p)$ it is difficult to find x. It can be shown that if this problem is difficult in the worst case, then it is also difficult in the average case.
- And other (e.g., basing on elliptic curves)

We want to prove security of cryptographic protocols assuming that these problems are hard.

What does it mean that they are hard for a majority of instances? How should this be defined?

One-way functions

A family of functions $f_n:\{0,1\}^n \to \{0,1\}^{m(n)}$ is $(\varepsilon(n),s(n))$-one-way,

if it is computable in polynomial time, and if for every randomized algorithm A working in time $s(n)$ it holds that $Pr[A$ reverses $f_n(x)] \leq \varepsilon(n)$

- „reverses" means: knowing $f_n(x)$ it finds y such that $f_n(y)=f_n(x)$
- the probability is over input strings $x \in \{0,1\}^n$ and over random bits tossed by A
- it is a reasonable assumption that the adversary can use random bits
- we could even assume that A is non-uniform, depending on n (i.e., from **P/poly**)

One-way functions

A family of functions $f_n:\{0,1\}^n \rightarrow \{0,1\}^{m(n)}$ is $(\varepsilon(n),s(n))$-one-way, if it is computable in polynomial time, and if for every randomized algorithm A working in time $s(n)$ it holds that $Pr[A \text{ reverses } f_n(x)] \leq \varepsilon(n)$

- we usually consider functions $s(n)$ growing faster than any polynomial; we suppose that the problems mentioned on the previous slide cannot be solved in subexponential time, i.e., faster than 2^{n^c} for some (maybe small) constant $c>0$
- the probability $\varepsilon(n)$ should decrease while increassing n (e.g., polynomially)
- Theorem (Yao): if there exist one-way functions for $\varepsilon(n)=1-1/n^c$ for some c (so-called "weak one-way function", which are difficult to reverse only for $1/n^c$ of inputs), then there exist one-way functions for $\varepsilon(n)<1/n^k$, for every k (so-called "strong one-way functions")

One-way functions

How our problems can be written as one-way functions:

- Discrete logarithm. Let $p_1, p_2, \ldots$ be a sequence of prime numbers, where p_i has i bits. Let g_i be a generator of $\mathbb{Z}_{p_i}^*$. Then the i-th function is given by: $x \to (g_i^x \bmod p_i)$.
- Factorization. It is known that there is a polynomial (wrt. n) randomized algorithm generating prime numbers of length n. We can treat it as a deterministic algorithm converting a sequence of random bits r to a prime number $A(r)$ of length n. Then the considered function (supposingly one-way) is $(r_1, r_2) \to A(r_1) \cdot A(r_2)$
- The RSA encoding is also a (supposingly one-way) function, etc.

Pseudorandom generators

The best encoding (one-time pad):

- The sender and the receiver both know a common (one-time) key x of the same length as the message m
- The sender sends (m XOR x)
- Receiver computes ((m XOR x) XOR x)=m
- A full security: the adversary, knowing $(m\ XOR\ x)$, but not knowing x, does not know anything about m (independently from his computational power)

Pseudorandom generators

The best encoding (one-time pad):

- The sender and the receiver both know a common (one-time) key x of the same length as the message m
- The sender sends (m XOR x)
- Receiver computes ((m XOR x) XOR x)=m
- A full security: the adversary, knowing $(m\ XOR\ x)$, but not knowing x, does not know anything about m (independently from his computational power)
- A difficulty: the parties need to share a very long key
- A connected difficulty: how to create such a long random string?
- A solution: pseudorandom generator – basing on a short (known to both parties) key, generate a long string, which looks like a random string

Pseudorandom generators

Definition:

- We consider a family of functions $g_n:\{0,1\}^n \rightarrow \{0,1\}^{m(n)}$ computable in polynomial time
- It is a $(\varepsilon(n),s(n))$-pseudorandom generator, if for any randomized algorithm A working in time $s(n)$, for every sufficiently large n:

$$|Pr_{x\in\{0,1\}^n}[A(g_n(x))=1]-Pr_{y\in\{0,1\}^n}[A(y)=1]|\leq\varepsilon(n)$$

(i.e., the generated sequence cannot be distinguished from a random sequence)

Pseudorandom generators

Definition:

- We consider a family of functions $g_n:\{0,1\}^n \rightarrow \{0,1\}^{m(n)}$ computable in polynomial time
- It is a $(\varepsilon(n),s(n))$-pseudorandom generator, if for any randomized algorithm A working in time $s(n)$, for every sufficiently large n:

$$|Pr_{x\in\{0,1\}^n}[A(g_n(x))=1]-Pr_{y\in\{0,1\}^n}[A(y)=1]|\leq\varepsilon(n)$$

(i.e., the generated sequence cannot be distinguished from a random sequence)

- Equivalent definition:

$$Pr_{x\in\{0,1\}^n}[A(g_n(x)[1..i])=g_n(x)[i+1]]\leq\varepsilon(n)$$

(when we see i bits of a pseudorandom sequence, we cannot compute the next bit)

The equivalence is not obvious, but it is not difficult

Pseudorandom generators

Theorem: There exist one-way functions $\Leftrightarrow$ there exist pseudorandom generators

Implication $\Leftarrow$ almost obvious: a pseudorandom generator has to be a one-way function

Implication $\Rightarrow$ much harder (using a one-way function in an appropriate way, one can create a pseudorandom generator)

Zero-knowledge proofs

A zero-knowledge proof – a procedure in which one party can prove to the other party that it has access to some information, without revealing this information

Zero-knowledge proofs

A zero-knowledge proof – a procedure in which one party can prove to the other party that it has access to some information, without revealing this information:

- If the prover indeed knows the information, he can always convince the verifier about this
- If the prover does not have the information, he can convince the verifier only with a small probability
- Verifier (even while cheating, i.e., not following the protocol) cannot reveal any knowledge about the information (this can be formalized appropriately)
- Both parties have a limited computational power (these are standard algorithms, working in polynomial time, having access to random bits).

Zero-knowledge proofs

Example – graph isomorphism:

- P claims that he knows an isomorphism between G_1 and G_2
- V wants to confirm this
- P sends a graph H
- V sends a number $i \in \{1,2\}$
- P sends an isomorphism between H and G_i

Zero-knowledge proofs

Example – graph isomorphism:

- P claims that he knows an isomorphism between G_1 and G_2
- V wants to confirm this
- P sends a graph H
- V sends a number $i \in \{1,2\}$
- P sends an isomorphism between H and G_i
- If P knowns an isomorphism between G_1 and G_2, he generates H by randomly permuting node labels of G_1. Then he can easily find an isomorphism between H and any G_i.
- If P does not know an isomorphism between G_1 and G_2, then he cannot find a graph H, from which he knows an isomorphism to both G_1 and G_2 – he succeeds with probability 1/2
- Knowing an isomorphism between G_i and a random permutation of G_i does not help V in finding an isomorphism between G_1 and G_2 (because this is a knowledge which could be generated by V himself)

Quantum computing

- a new computational model
- one supposes, that it will be possibly to realize them physically, in the future (oppositely to e.g. nondeterministic or alternating computations) – in principle, they cannot be realized currently; but it seems that the physical lows allow their existence
- although quantum computers do not exist, the mathematical model of quantum computations is well defined

Quantum computing

- a new computational model
- one supposes, that it will be possibly to realize them physically, in the future (oppositely to e.g. nondeterministic or alternating computations) – in principle, they cannot be realized currently; but it seems that the physical lows allow their existence
- although quantum computers do not exist, the mathematical model of quantum computations is well defined
- it is possible that in some problems quantum computers are exponentially faster than classical computers (deterministic or randomized)
- thus, potentially, this is in contrary with the strong Church-Turing thesis (every physically realizable computational device can be simulated on a Turing machine, with a polynomial overhead)

Quantum computing

How do we compute?

- m qubits = a vector v (of norm 1) from $\mathbb{C}^{2^m}$
- an operation = multiplying the vector v by a fixed unitary matrix
  (unitary = not changing the norm of v)
- only operations “modifying ≤3 qubits” are allowed (*elementary operations*)
- a “read” operation: a value $i \in \{0,1\}^m$ is read with probability $|v_i|^2$ (we cannot directly read the whole vector v)

Quantum computing

How do we compute?

- m qubits = a vector v (of norm 1) from $\mathbb{C}^{2^m}$
- an operation = multiplying the vector v by a fixed unitary matrix
  (unitary = not changing the norm of v)
- only operations "modifying ≤3 qubits" are allowed (*elementary operations*)
  Can we apply any elementary operation? There are infinitely many of them.
  ⇒ Every elementary operation can be simulated (approximated) by a composition of two basic operations, called *Hadamard operation* and *Tofolli operation*.
  What about operations touching >3 qubits?
  Every operation on k qubits can be simulated by composing $2^{O(k)}$ elementary operations
- a "read" operation: a value $i \in \{0,1\}^m$ is read with probability $|v_i|^2$ (we cannot directly read the whole vector v)

Quantum computing

How do we compute?

- m qubits = a vector v (of norm 1) from $\mathbb{C}^{2^m}$
- an operation = multiplying the vector v by a fixed unitary matrix
  (unitary = not changing the norm of v)
- only operations "modifying ≤3 qubits" are allowed (*elementary operations*)
- a "read" operation: a value $i \in \{0,1\}^m$ is read with probability $|v_i|^2$ (we cannot directly read the whole vector v)

How is this related to a classical computation?

- a classical computation, in which the memory content is $i \in \{0,1\}^m$ corresponds to a situation when $v_i=1$ and $v_j=0$ $\forall j \neq i$
- thus (intuitively): in a quantum computation we perform the same computation in parallel for many possible memory contents, and at the end we pick randomly one of the obtained outputs

Quantum computing

What can be computed?

- Let **BQP** – languages recognizable on a quantum computer in polynomial time (defined similarly to **BPP** – the probability of getting a correct result $>3/4$)
- It is easy to prove that **BPP**$\subseteq$**BQP**
- On the other hand **BQP**$\subseteq$**PSPACE**

Quantum computing

What can be computed?

- Let **BQP** – languages recognizable on a quantum computer in polynomial time (defined similarly to **BPP** – the probability of getting a correct result $>3/4$)
- It is easy to prove that **BPP**$\subseteq$**BQP**
- On the other hand **BQP**$\subseteq$**PSPACE**

  Proof sketch:

  - A recursive procedure $Coeff(i,t)$, where $i \in \{0,1\}^m$ and t is the number of steps – compute the value of v_i after t steps.
  - This value depends on $Coeff(j,t-1)$ for 8 values of j – because only 3 qubits are modified.
  - Because t is polynomial, the memory used for computing $Coeff(i,t)$ (the depth of the stack) is polynomial

Quantum computing

What can be computed?

- Let **BQP** – languages recognizable on a quantum computer in polynomial time (defined similarly to **BPP** – the probability of getting a correct result $>3/4$)
- It is easy to prove that **BPP**$\subseteq$**BQP**
- On the other hand **BQP**$\subseteq$**PSPACE**
- In **BQP** we can decompose numbers into prime factors (Shor's algorithm, 1994) – and classically we cannot!
- A general formulation of search problems (**NP**): given a function $f:\{0,1\}^n \rightarrow \{0,1\}$ computable in polynomial time, does there exist some a such that $f(a)=1$? Classically one needs 2^n calls to f, on quantum computers $2^{n/2}$ is enough (quadratically less) – Grover's algorithm

Quantum computing

What can be computed?

- Let **BQP** – languages recognizable on a quantum computer in polynomial time (defined similarly to **BPP** – the probability of getting a correct result $>3/4$)
- It is easy to prove that **BPP**$\subseteq$**BQP**
- On the other hand **BQP**$\subseteq$**PSPACE**
- In **BQP** we can decompose numbers into prime factors (Shor's algorithm, 1994) – and classically we cannot!
- A general formulation of search problems (**NP**): given a function $f:\{0,1\}^n \rightarrow \{0,1\}$ computable in polynomial time, does there exist some a such that $f(a)=1$? Classically one needs 2^n calls to f, on quantum computers $2^{n/2}$ is enough (quadratically less) – Grover
- If f is given as an oracle, one cannot do better (i.e., $\mathbf{NP}^A \not\subseteq \mathbf{BQP}^A$ for some oracle A)
- Thus a quantum computer can quickly solve a search problem only by taking advantage of the structure of the problem (like for finding prime factors)