CM 32025 Notes

Prof. N.N. Vorobjob for CM30073
Edited by JHD/AKU

October 20, 2025

Contents

Chapter 1

Lecture Notes 1

1. About this unit

The unit will cover several closely related topics.

The central one is NP-completeness. That is the study of a certain, very wide class of computational problems (called NP-complete) to solve which no fast algorithms are known at present. The reason why we are interested in these hard-to-solve problems is mainly their practical importance. Many fundamental and natural problems from areas as diverse as logic, automata and language theory, networks, algebra and number theory, mathematical programming and graph theory, are NP-complete.

Example 1. The classical *Travelling salesman problem* is formulated as follows.

Input : A finite set of "cities" $\{1, \ldots, n\}$ and a set of positive integer "distances" $d(i, j)$ between i and j for each pair $i < j$.

Output : A permutation $i_1, \ldots, i_n$ of cities $1, \ldots, n$ such that the sum

$$\sum_{1 \leq j \leq n-1} d(i_j, i_{j+1}) + d(i_1, i_n)$$

is as small as possible. This sum is the length of a tour starting in the city i_1, visiting *all* the cities in a certain order and returning back to i_1.

An obvious algorithm for this problem simply computes the length for each possible tour and then finds the one with the smallest length. It's easy to see that the number of all tours is exactly $n!$ (the number of all permutations of n elements) which is an extremely large value for reasonable practical values of n. We would like to have an algorithm whose number of steps does not grow as fast as that with the increase of input data.

Example 2. This example is from elementary logic. Consider a Boolean formula, that is an expression obtained from *variables* (letters in a finite alphabet)

by means of basic logical operations $\wedge$, $\vee$, $\rightarrow$, $\neg$. For instance, $\neg x_1 \wedge (x_2 \rightarrow x_1)$ is a Boolean formula. Replacing each variable in a formula by one of two values **true** or **false** we can evaluate the whole formula using the well-known rules. A formula is called *satisfiable* if it's not identically false, i.e., if there exists an assignment of values **true** or **false** to variables such that the resulting value of the formula is **true**.

Input : A Boolean formula F with variables $x_1, \ldots, x_n$.

Output : **Yes** if F is satisfiable, else **No**.

As in the Example 1, there is an obvious "brute force" algorithm for this problem, which examines in an arbitrary order *all possible* **true/false** *evaluations* of variables and for each of them computes the corresponding truth value of F. If for at least one evaluation the value of F is **true**, then F is satisfiable and the output is **Yes**, else the output in **No**. Since there exists exactly 2^n different assignments of truth values to variables, the algorithm makes at least 2^n elementary steps.

In both examples "brute force" algorithms use a practically unacceptable number of elementary steps (*complexity,* see the reminder below). An *acceptable* number of steps is the one that grows *polynomially* with the growth of the size of the input. An important observation is, that though Travelling Salesman (TS) and Satisfiability (SAT) appear to be the problems from two quite different areas, concerning seemingly unrelated objects, computationally they are equivalent. More precisely, if there is an effective (= polynomial time complexity) algorithm for TS, then there exists an effective algorithm for SAT, and vice versa.

We are going to define a very large class, called NP, of problems which includes Examples 1 and 2. The defining characteristic of class NP will be the possibility to solve each of it's problems by direct "brute force" search through exponential number of all possible "cases". Then we shall define a subclass of problems in NP, called NP-complete which are "hardest" in NP. The latter means that if there exists an effective algorithm for solving an NP-complete problem, then there exists an effective algorithm for solving *any* problem from NP. We'll prove that both TS and SAT are NP-complete. Presently (Feb. 2020) no algorithms of polynomial time complexity for NP-complete problems are known. In fact, finding such an algorithm or proving its impossibility is a *major open challenge* in theoretical computer science.

Sometimes it's makes sense to relax the definition of NP-completeness and consider problems which are *at least as hard* as any problem from the class NP, but themselves not necessary belonging to NP. Such problems are called NP-hard.

Of course, we will give in due time full mathematical definitions, based on *Turing machines*, for all mentioned NP-related concepts.

Proving that a computational problem Π is NP-complete or NP-hard, means that has Π a certain complexity status. Namely, Π is as hard to solve as Traveling Salesman and other famous problems. (See the illustration.)

In our study of NP-completeness we will consider many examples of the problems from *graph theory*. Fortunately, not all important fundamental computational problems for graphs are NP-hard. Many accept effective (polynomial-time) algorithms, which are, however rather nontrivial. Some of them are discussed a further part of this unit. We'll also consider the question: what to do if a graph problem (or any other) turns out to be NP-hard. That problem could happen to be too important practically to be simply abandoned. One of the options in this situations is to seek an "approximate" solution which sometimes is possible to find effectively. We will construct a polynomial-time approximation algorithms for Traveling Salesman and for some other problems for graphs.

Another major topic will be *quantum computing*. It turns out that a (hypothetical) quantum computer is able to solve very efficiently some computational problems that are out of the reach of conventional bit-based computers. We will introduce a theoretical model of a quantum computer, and will describe an efficient algorithms for some specific computational problems.

Finally, we will study some topological methods of proving complexity lower bounds for problems related to sorting.

2. O-notation

Definition. $O\big(g(n)\big) = \{f(n) :$ there exist constants $c > 0$ and $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.

Convention. If $f(n) \in O\big(g(n)\big)$, then we write: $f(n) = O\big(g(n)\big)$.

Examples.

(1) $a_0 n^d + a_1 n^{d-1} + \cdots + a_{n-1} n + a_n = \Theta(n^d)$.

(2) (Geometric progression) $1 + a + a^2 + \cdots + a^n = (a^{n+1} - 1)/(a - 1) = \Theta(a^n)$.

(3) $n^d = O(n^t)$ for $t \ge d$, $c = 1$, $n_0 = 1$. If $t > d$ then n^t is *not* $O(n^d)$.

(4) $n^d = O(2^n)$ (but 2^n is *not* $O(n^d)$). This follows from Taylor's expansion for 2^n:

$$2^n = 1 + n/p + n^2/(2!p^2) + n^3/(3!p^3) + \cdots,$$

where $p = \log e$, $e = 2,71828\ldots$.

Actually, $n^d = O(a^n)$ for any $a > 1$.

(5) $f(n) = O(1)$ means that $f(n)$ is less than a constant for all sufficiently large n.

(6) $\log n = O(n^\varepsilon)$ for any real number $\varepsilon > 0$.

(7) $n \log n = O(n^2)$.

(8) $2^n = O(n!)$.

3. Ω and Θ-notations

Definition. $\Omega\big(g(n)\big) = \{f(n) :$ there exist constants $c > 0$, $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.

Examples — reverse examples for O-notation.

Definition. *If for any two functions f and g holds $f(n) = O\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$ we write $f(n) = \Theta\big(g(n)\big)$. In other words, recalling that $O\big(g(n)\big)$ and $\Omega\big(g(n)\big)$ are sets of functions, we define $\Theta\big(g(n)\big) := O\big(g(n)\big) \cap \Omega\big(g(n)\big)$.*

4. Necessity of a rigorous definition of an algorithm

When designing new algorithms and estimating their complexities, we usually don't mention a precise mathematical model of the computation but rather describe the procedure in general mathematical terms. A programming language code is a good approximation to a mathematical description, but is not rigorous enough since most practical languages have ambiguities, contradictions, etc. We need a proper mathematical definition to be able to talk not only about existence but also about nonexistence of an algorithm in a given class. To make sure that an algorithm exists it is sufficient to present one (e.g., in a form of programming code), while to prove that no algorithm exists we have to make clear what exactly does not take place.

5. Input encoding

A convenient model of computation for our purpose is a *Turing machine*. A Turing machine accepts as inputs *words* or *strings* in a finite alphabet. Therefore if the possible inputs are not the words already, then they should be encoded as words. Usually such an encoding is a trivial matter. We consider as an example encodings for graphs.

Definition. *Graph $G = (V, E)$ consists of a finite set $V = \{v_1, \dots, v_n\}$ of vertices and a finite set E of edges, where E is a relation on V. In other words, E is a subset of the set of all pairs (v_i, v_j) with $v_i, v_j \in V$. If every pair is ordered (i.e., pairs (v_i, v_j) and (v_j, v_i) are considered as different), then G is called directed graph. If every pair is not ordered (i.e., (v_i, v_j) and (v_j, v_i) are identified), then G is undirected.*

Example. We can represent any undirected graph by a list of edges, for example

$$(v_5, v_7), (v_1, v_3,), (v_1, v_7). \tag{1}$$

Now we encode it as a word in the alphabet $\{v,\ 0,\ 1,\ 2,\ \dots,\ 9\}$:

$$v5v7v1v3v1v7$$

Another way for graph representation is in a form of *adjacency matrix*, that is the $(|V| \times |V|)$-matrix ($|V|$ denotes the number of vertices) whose (i, j)-element is equal to 1 if $(v_i, v_j) \in E$, else equal to 0. Writing the lines of adjacency

matrix in single line end-to-end we get an encoding of a graph as a word in the alphabet {0, 1}. For instance graph (1) will be encoded by

$$0010001000000010000000000000000000010000000010001 00$$

We see that the lengths of the input differ for distinct encoding schemes. Nevertheless, any two "reasonable" schemes differ "at most polynomially". We formulate the following informal thesis.

For any computational problem Π*, for any two reasonable encoding schemes* S_1, S_2 *there exists a polynomial* $p(n)$ *such that for any input* I *the following inequalities holds:*

$$|S_1(I)| \leq p(|S_2(I)|),$$

where $|S_i(I)|$ *denotes the length of the code* $S_i(I)$*.*

This is not, of course, a formal statement since it refers to a "reasonable" scheme. A scheme is "reasonable" if it is "compact" enough, that is, if it does not lead to artificial "blowing up" of the input. Blowing up can lead to an increase of the input so that an exponential time algorithm will turn into a polynomial time.

6. Computational problems and languages

In our discussion of NP-completeness we shall usually restrict ourselves to the computational problems with output **Yes** or **No** for any input. We will call such problems – **Yes - No** *problems*. In many cases other problems can be reduced to **Yes - No** problems with polynomial complexity (formal definition of reduction with polynomial complexity will follow). For example, Traveling Salesman can be reduced with polynomial complexity to the following problem.

Input : A finite set of "cities" $\{1, \ldots, n\}$, a set of integer "distances" $d(i, j) > 0$ between i and j for each pair $i < j$, and a natural number $B > 0$.

Output : **Yes**, if there exists a permutation $i_1, \ldots, i_n$ of cities $1, \ldots, n$ such that

$$\sum_{1 \leq j \leq n-1} d(i_j, i_{j+1}) + d(i_1, i_n) \leq B,$$

else **No**.

Definition. *Let* A *be a finite alphabet. An arbitrary (finite or infinite) set of words (strings of letters) in* A *is called a language (over* A*).*

Remark. The set of words in the definition of the language is supposed to be *semi-decidable (recursively enumerable).*

Every **Yes - No** computational problem corresponds to a language over a certain alphabet, namely to the set of all input codes having output **Yes**. Conversely, for a given (by a formal grammar) language $\mathcal{L}$ over an alphabet A one considers a **Yes - No** *membership* problem:

Input : A word w in A.

Output : **Yes** if $w \in \mathcal{L}$, else **No**.

7. Turing Machine

Turing Machine consists of the following ingredients:

(1) Tape alphabet $\Gamma = \{a_1, \ldots, a_l, \mathrm{b}\}$ (where b denotes blank);

(2) Input alphabet $\Sigma \subset \Gamma \setminus \{\mathrm{b}\}$;

(3) Set of states $Q = \{q_0, \ldots, q_m, q_Y, q_N\}$, where
q_0 is the *initial* state,
q_Y is the *final* **Yes**-state,
q_N is the *final* **No**-state;

(4) Transition function

$$\delta \, : \, (Q \setminus \{q_Y, q_N\}) \times \Gamma \, \longrightarrow Q \times \Gamma \times \{R, S, L\}.$$

TM is supposed to work in the following way. An input word is written on a tape that is infinite in both directions and is divided into cells. Each cell of the tape contains one letter from Γ. Before the computation starts an input word is written on the tape occupying subsequent cells from left to right, TM is in the initial state q_0, and its "working head" observes the leftmost letter of the input word (or, in other words, the cell in which that letter is written).

The initial configuration for a *generic step* is similar: a word over Γ is written on the tape, TM is in a state q_i, its working head observes a certain cell containing a letter a_j. At a generic step TM applies the transition function δ to the pair (q_i, a_j) to obtain a triple (q_l, a_r, T) where $T \in \{R, S, L\}$. TM adopts the state q_l, replaces a_j by a_r in the cell observed, and shifts the working head one cell to the right (if $T = R$) or to the left (if $T = L$) or does not shift the head at all (if $T = S$). The computation process terminates if and only if TM adopts either the state q_Y or the state q_N.

We will say that a TM *accepts* an input w of a **Yes - No** problem if the computation terminates on w with the state q_Y. Otherwise TM terminates on w with the state q_N, and it does not accept w. In the first case we will also say that the output on w is **Yes**, in the second case we will say that the output on w is **No**.

8. Complexity for Turing Machines

Complexity or *running time* or *working time* of a TM which terminates on all inputs is a function $T : \mathbf{N} \longrightarrow \mathbf{N}$ from natural numbers to natural numbers defined by equality:
$T(n) = \max\{m :$ there exists an input w consisting of n letters such that TM uses m steps.$\}$

9. Examples of Turing Machine and their complexities

Exercise. Construct a TM for recognizing all even non-negative integer numbers in *unary* representation. Make sure that the complexity $T(n)$ of that TM is $n + 1$.

Definition. *A palindrome in alphabet A is a word $a_1 a_2 \cdots a_n$ such that $a_i \in A$ and $a_i = a_{n-i+1}$ for any i, $1 \leq i \leq n$, e.g.* abba *is a palindrome in alphabet* $\{\mathrm{a}, \mathrm{b}\}$.

Example. TM recognizing palindromes in alphabet A can work as follows. For each letter a of the input alphabet Σ TM has two states q_a, q'_a, herein for two distinct letters $a, b \in \Sigma$ the states q_a, q'_a, q_b, q'_b are pair-wise distinct. Let the input code be $a_1 a_2 \cdots a_n$. TM "remembers" a_1 in the input code by adopting the corresponding state q_{a_1}, erases a_1 and, remaining in the state q_{a_1}, moves to the end of the input code. The last letter a_n in the input is the one appearing immediately before the first occurrence of the blank b. If a_n does not coincide with a_1 (does not match the state q_{a_1}), then TM terminates its work by adopting the state q_N, else TM erases the last letter $a_n = a_1$, "remembers" a_{n-1} by adopting the state $q'_{a_{n-1}}$, erases a_{n-1}, and moves to the beginning of the word on tape which now starts with a_2. If a_2 does not coincide with a_{n-1}, then TM adopts the state q_N, else it erases a_2, "remembers" a_3, and the cycle continues in a similar way. It is easy to understand how TM should behave on the last steps in case the input is indeed a palindrome.

The complexity of the described TM is $O(n^2)$. A difficult theorem states that for *any* TM solving the palindrome problem the complexity is $\Omega(n^2)$. That theorem remains true for all "reasonable" modifications of the concept of TM with one tape and one working head. If we, however, allow *two* tapes (with one working head per tape) it's trivial to construct a machine for recognizing palindromes in linear time $O(n)$.

A formal definition of 2-tape Turing Machine (2TM) straightforwardly follows the pattern of the definition of ordinary TM. Not going into the details, let us mention that the input code appears on the tape 1, on a given step the machine performs writing–erasing and shifting independently and simultaneously on each tape. Thus, the transition function is defined on triples (state, symbol on tape 1, symbol on tape 2), and its value is a 5-tuple (new state, new symbol on tape 1, new symbol on tape 2, direction of shift on tape 1, direction of shift on tape 2).

A 2TM for palindrome recognition first copies the input code on tape 2 ($O(n)$ steps), then returns the working head on tape 1 to the beginning of the input ($O(n)$ steps), and then compares the words on both tapes reading the one on

tape 1 from left to right, and the one on tape 2 from right to left ($O(n)$ steps). The total amount of steps is $O(n)$.

10. Class P

Definition. *P is the class of all* **Yes - No** *computational problems (languages) $\mathcal{L}$ such that there exists TM and a polynomial $p(n)$ such that TM solves $\mathcal{L}$ with complexity $T(n) \leq p(n)$ for all $n \geq 1$.*

Examples.

(1) "Even unary non-negative numbers" is in P, take $n+1$ as polynomial $p(n)$ from the definition;

(2) "Palindromes" is in P, $p(n) = cn^2$ for a large enough positive constant c;

(4) "Primality" is in P. This is a problem of deciding for a given integer number $N > 0$, represented in decimal (or binary) form, whether N is prime.

(4) "Traveling Salesman" and "Satisfiability" are *probably* not in P, but no proofs for any of them are known.

Chapter 2

Lecture Notes 2

1. Nondeterministic Turing Machine

Nondeterministic Turing Machine (NTM) consists of the same ingredients as "ordinary" Turing Machine, namely, of tape alphabet Γ; input alphabet $\Sigma \subset \Gamma \setminus \{\mathrm{b}\}$; set of states Q which includes the initial state q_0 and final states q_Y, q_N; transition function δ. The difference between TM and NTM is in the way they work.

NTM has a "guessing module" with write-only head attached to it. Before NTM starts working, an input word is written on the tape (the tape is infinite in both directions and divided into cells) from left to right starting from the cell 1; the working head observes cell 1; guessing head observes cell -1.

NTM works in two stages:

(1, guessing stage) Guessing head moves along the tape from right to left shifting to the next cell on each step. On each step guessing head writes a letter from Γ on the tape. The process may or may not terminate. During this stage the control module of NTM and its working head remain passive; NTM is not in any "state". When (if ever) the guessing state terminates, NTM moves to the second stage.

(2, verifying stage) NTM adopts the initial state q_0 and workes as ordinary TM with the combination of the "guessed word" and the original input word, as its input. During this second stage the guessing module and its guessing head are passive.

Definition. *For a fixed input word x, the sequence of steps from stages (1) and (2) is called* computation *for x*

2. Accepted inputs and complexity

Definition. *NTM accepts an input x if there exists a computation (for x) ending with the state q_Y.*

Observe that if NTM accepts x there might also be computations (for x) ending with q_N. There might be many computations (for x) leading to q_Y.

Definition. *Let NTM accept x. Complexity of accepting x is the number of steps in the* shortest *accepting computation for x.*

Definition. Complexity *of NTM is the function*

$$T(n) = \max\{m : \mathrm{there exists} x, |x| = n \mathrm{such that the complexity of accepting} x \mathrm{is} m\}.$$

If such x does not exist let $T(n) = 1$.

3. Class NP

Definition. *We say that NTM solves the* **Yes - No** *problem $\mathcal{L}$ if it accepts exactly all* **Yes** *inputs in $\mathcal{L}$.*

Definition. *Class NP consists of* **Yes - No** *problems (languages) $\mathcal{L}$ such that there exist polynomial $p(n)$ and NTM for solving $\mathcal{L}$ with complexity $T(n) \leq p(n)$ for any $n \geq 1$.*

Examples.

(1) Travelling Salesman $\in NP$. To show this, we need to construct a NTM for solving this problem with polynomial complexity. On the guessing stage the NTM produces an arbitrary permutation of cities, and on the verifying stage, working like an "ordinary" TM, checks whether the length of the guessed tour does not exceed the boundary B.

An accepting computation for an input x exists if and only if there is a short tour. This computation consists in writing out a guess and finding the corresponding sum of all distances. Clearly the complexity of this computation is polynomial. According to the definition, a non-accepting computation does not contibute to the complexity.

(2) Boolean satisfiability $\in NP$. NTM first guesses a satisfying assignment of truth values to variables, and then verifies the guess in polynomial time.

4. P $\subset$ NP

Theorem. $P \subset NP$.

Proof. Let $\mathcal{L}$ be a **Yes - No** problem from P, and M be a (deterministic) TM for solving $\mathcal{L}$ in polynomial time. We get a NMT for solving $\mathcal{L}$ in polynomial time by immitating M on the verification stage and ignoring the guessing stage. Theorem is proved.

The question whether $NP \subset P$ and, therefore, $P = NP$, is a famous open problem.

5. Complexity relations between P and NP

Theorem. *Let $\mathcal{L} \in NP$. Then there exist a polynomial $p(n)$ and TM for solving $\mathcal{L}$ with complexity $O(2^{p(n)})$.*

Proof. Let N be an NTM for solving $\mathcal{L}$ with complexity $T_N(n) \leq r(n)$ where $r(n)$ is a polynomial. By the definition of NTM, for every accepted input x with $|x| = n$ there is a guess word in the tape alphabet Γ of the length not greater than $r(n)$ such that N adopts the state q_Y in no more than $r(n)$ steps.

The total number of possible guesses is less than $|\Gamma|^{r(n)+1}$.

Now we construct a (deterministic) TM M as follows. M examines every possible guess in turn, and for each of them runs the verification stage of N up to $r(n)$ steps. That will take

$$O(r(n)|\Gamma|^{r(n)})$$

steps. By the definition of O-symbol, that's the same as $O(2^{p(n)})$ for a certain polynomial $p(n)$.

The theorem is proved.

6. Polynomial transformation: definition

Let $\mathcal{L}_1$ be a language over an alphabet A_1, and $\mathcal{L}_2$ be a language over an alphabet A_2. A_i^* will denote the set of *all* words over A_i for $i = 1, 2$.

Definition. *We say that $\mathcal{L}_1$ is* polynomially transformable *to $\mathcal{L}_2$ if there exists a function*

$$f \ : \quad A_1^* \longrightarrow A_2^*$$

such that

(1) f is computable in polynomial time (i.e. there is a polynomial P_f such that $f(w)$ can be computed in at most $P_f(|w|)$ steps);

(2) for any $x \in A_1^*$

$$x \in \mathcal{L}_1 \text{ if and only if } f(x) \in \mathcal{L}_2.$$

Notation: $\mathcal{L}_1 \propto \mathcal{L}_2$.

7. Polynomial transformation: problems solvable in polynomial time

Theorem. *Let $\mathcal{L}_1 \propto \mathcal{L}_2$ and $\mathcal{L}_2 \in P$. Then $\mathcal{L}_1 \in P$. (Equivalently: if $\mathcal{L}_1 \notin P$, then $\mathcal{L}_2 \notin P$).*

Proof. Since $\mathcal{L}_2 \in P$, there exists TM M_2 deciding for $y \in A_2^*$ whether or not $y \in \mathcal{L}_2$.

Now we construct a new TM, M_1, deciding membership to $\mathcal{L}_1$. Let $f : A_1^* \longrightarrow A_2^*$ be a function realizing the polynomial transformation of $\mathcal{L}_1$ to $\mathcal{L}_2$, and M_f be a TM for computing f in polynomial time. For a given input $x \in A_1^*$ the machine M_1 works in two stages: on the first one, M_f computes $f(x)$; on the second, M_1 decides whether $f(x) \in \mathcal{L}_2$ using the machine M_2 as a subroutine.

To complete the proof it remains to estimate complexity of M_1. Let $T_f(n)$ and $T_2(n)$ be the complexities of M_f and M_2 respectively. Then the complexity of M_1 is

$$O\big(T_f(|x|) + T_2(|f(x)|)\big).$$

But $|f(x)| \le T_f(|x|)$, therefore the complexity of M_1 is

$$O\big(T_f(|x|) + T_2(T_f(|x|)),$$

i.e., is polynomial in $|x|$.

Theorem is proved.

8. Transitivity of $\propto$

Theorem. *For three languages $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, if*

$$\mathcal{L}_1 \propto \mathcal{L}_2 \quad \text{and} \quad \mathcal{L}_2 \propto \mathcal{L}_3, \tag{$*$}$$

then $\mathcal{L}_1 \propto \mathcal{L}_3$.

Proof. Let $f_{1,2}$ and $f_{2,3}$ be functions realizing polynomial transformations $(*)$. Define the function $f_{1,3}$ as a composition of $f_{1,2}$ and $f_{2,3}$:

$$f_{1,3}(x) = f_{2,3}\big(f_{1,2}(x)\big).$$

It is easy to check that $f_{1,3}$ realizes a polynomial transformation $\mathcal{L}_1 \propto \mathcal{L}_3$.

Theorem is proved.

9. Polynomial equivalence

Definition. *A language $\mathcal{L}_1$ is* polynomially equivalent *to a language $\mathcal{L}_2$ if $\mathcal{L}_1 \propto \mathcal{L}_2$ and $\mathcal{L}_2 \propto \mathcal{L}_1$.*

We will denote this relation by $\mathcal{L}_1$ p.e. $\mathcal{L}_2$.

Theorem. *Polynomial equivalence is an* equivalence relation *on the set of all* **Yes - No** *problems (languages), i.e., for any two problems $\mathcal{L}_1$, $\mathcal{L}_2$ the following holds:*

(1) $\mathcal{L}_1$ p.e. $\mathcal{L}_1$*;*

(2) if $\mathcal{L}_1$ p.e. $\mathcal{L}_2$ *then* $\mathcal{L}_2$ p.e. $\mathcal{L}_1$*;*

(3) relation p.e. is transitive.

Proof. (1), (2) are trivial; (3) is the content of Theorem, Section 8.

Example. Any two languages from class P are polynomially equivalent.

10. NP-completeness

Definition. **Yes - No** *problem (language) $\mathcal{L}$ is called* NP-complete *if*

(1) $\mathcal{L} \in NP$;

(2) *for any problem* $\mathcal{L}'$, *if* $\mathcal{L}' \in NP$, *then* $\mathcal{L}' \propto \mathcal{L}$.

Remark. Any two NP-complete languages are polynomially equivalent.

Theorem. *If* $\mathcal{L}_1, \mathcal{L}_2 \in NP$, $\mathcal{L}_1 \propto \mathcal{L}_2$, *and* $\mathcal{L}_1$ *is* NP*-complete, then* $\mathcal{L}_2$ *is also* NP*-complete.*

Proof. Let $\mathcal{L}'$ be an arbitrary language from NP. Because $\mathcal{L}_1$ is NP-complete, $\mathcal{L}' \propto \mathcal{L}_1$. From transitivity of $\propto$ then follows that $\mathcal{L}' \propto \mathcal{L}_2$.
Theorem is proved.

This Theorem implies that to prove NP-completeness of a **Yes - No** problem $\mathcal{L}$, it is sufficient to show that:

(1) $\mathcal{L} \in NP$;

(2) for *at least one* already known NP-complete problem $\mathcal{L}'$, it holds that $\mathcal{L}' \propto \mathcal{L}$.

Note the difference between this strategy of proving NP-completeness, and the method that follows from the definition of NP-completeness: in the latter case we need to consider somehow transformations of *all* problems from NP, while in the former case — a transformation of only one particular problem, which is apparently much easier.

2.11 Polynomial Transformations and SAT

Notation 1 (Literal) *We use* $\vee$ *for logical (inclusive) or, and* $\wedge$ *for logical and. We can use either prefix* $\neg$ *or overlining for negation. Because of* $\overline{a \vee b} \Rightarrow \overline{a} \wedge \overline{b}$ *and its converse* $\overline{a \wedge b} \Rightarrow \overline{a} \vee \overline{b}$ *we can end up with negation only applied to variables: a variable or negated variable is called a* literal.

Definition 1 (CNF) *A Boolean expression is said to be in* Conjunctive Normal Form *(CNF) if it is* $\bigwedge_i C_i$ *where each clause* C_i *is* $\bigvee_j l_{i,j}$ *and the* $l_{i,j}$ *are literals.*

Definition 2 (DNF) *A Boolean expression is said to be in* Disjunctive Normal Form *(DNF) if it is* $\bigvee_i C_i$ *where each clause* C_i *is* $\bigwedge_j l_{i,j}$ *and the* $l_{i,j}$ *are literals.*

Example 1 (The obvious transformation from CNF to DNF can be exponential) *Consider the CNF*

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_n \vee b_n). \tag{2.1}$$

This has length $\mathcal{O}(n)$, *but the corresponding DNF is*

$$(a_1 \wedge a_2 \wedge \cdots \wedge a_n) \vee (b_1 \wedge a_2 \wedge \cdots \wedge a_n) \vee \cdots \tag{2.2}$$

with 2^n *terms, one for each* n*-string of* a, b.

Example 2 (The obvious transformation from DNF to CNF can be exponential) *Consider the DNF*

$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \cdots \vee (a_n \wedge b_n). \tag{2.3}$$

This has length $\mathcal{O}(n)$, but the corresponding DNF is

$$(a_1 \vee a_2 \vee \cdots \vee a_n) \wedge (b_1 \vee a_2 \vee \cdots \vee a_n) \wedge \cdots \tag{2.4}$$

with 2^n terms, one for each n-string of a, b.

Transformation 1 *Tseitin [Tse68] We can replace $p \vee q$ by $(z \vee p) \wedge (\overline{z} \vee q)$, where z is a new variable. In our case p is $a_1 \wedge b_1$, so $z \vee (a_1 \wedge b_1)$ becomes $(z \vee a_1) \wedge (z \vee b_1)$. Hence $(a_1 \wedge b_1) \vee (a_2 \wedge b_2)$ becomes a $\wedge$ of four $\vee$: call it $P_{1,2} = \bigwedge c_i$. Note that each a_i, b_i occurs only once in $P_{1,2}$.*

We then have $P_{1,2} \vee (a_3 \wedge b_3)$, and apply the same trick, but again with a new *variable, say y, giving*

$$(y \vee \bigwedge c_i) \wedge (\overline{y} \vee (a_3 \wedge b_3), \tag{2.5}$$

which becomes

$$\left(\bigwedge (y \vee c_i)\right) \wedge (\overline{y} \vee a_3) \wedge (\overline{y} \vee b_3). \tag{2.6}$$

The process continues, and the blowup is almost linear (almost because we are introducing $\mathcal{O}(n)$ new symbols, which will require $\mathcal{O}(\log_2 n)$ bits each to represent them).

2.12 Probabilistic Algorithms

Note that $\exists$ variants and Wikipedia is uncharacteristically dubious dubious

2.12.1 Monte Carlo

A polynomial time algorithm that returns the right result with high probability (but not necessarily 1).

Example 3 (Rabin's primality test [Rab80]) *. If N is prime returns "prime", if not, returns "prime" with probability $\leq 1/4$ (so if run 10 times on N, $\leq 1/4^{10} < 10^{-6}$). [Dav92, We need truly random].*

This has one-sided error, in that "not prime" is always correct. There are also algorithms with two-sided errors.

Definition 3 (RP) *An algorithm A for recognising L is in the class RP (*randomised polynomial time*) if there is a constant $c < 1/2$ such that A always recognises $\notin L$, but given a string $w \in L$ may claim that it is not in L with probability $\leq c$ on each run of A on w, i.e. A is not consistently wrong.*

Similarly there is a class *co-RP* that may fail to recognise $\notin L$ with probability $\leq c$, but never falsely asserts that a word is not in L. So Rabin is RP for compositeness, and therefore co-RP for primality.

Definition 4 *An algorithm A for recognising L is in the class BPP (*bounded-error probabilistic polynomial*) if there is a constant $c < 1/2$ such that A recognises L with probability $\geq 1 - c$, but given a string $w \notin L$ may claim that it is in L with probability $\leq c$ on each run of A on w, i.e. A is not consistently wrong.*

2.12.2 Las Vegas

Always the right result and polynomial time with high probability (expected polynomial time).

Definition 5 (ZPP (1)) *An algorithm A for recognising L is in the class ZPP (*zero-error probabilistic polynomial time*) if it always returns the correct answer, and, for every input, the* expected *running time is polynomial.*

Theorem 1 *ZPP=RP∩co-RP.*

A consequence of this is that ZPP=co-ZPP. The proof is in three parts.

ZPP⊃ RP∩co-RP Run both alternately until they agree! More precisely, run the RP algorithm, and if it says "no" return "no" (because this is correct). If the RP algorithm says "yes", then run the co-RP algorithm, and, if this says "yes", return "yes" (because this is correct). Otherwise one algorithm has given the wrong result (an event of probability $\leq c$), and we try again. Our expected running time is $1/(1-c)$ times the greater running time of the RP and co-RP algorithms.

ZPP⊂ RP Let T be the expected running time of the ZPP algorithm on the input. Run it for time $2T$, and if it produces an answer, return it. If there is no answer, say NO. The probability that it will stop before $2T$ is at least $1/2$, so we have an RP algorithm.

ZPP⊂ co-RP The same, except we return YES on a time-out.

2.12.3 ZPP'

These items inspire an alternative definition of ZPP.

Definition 6 (ZPP (2)) *An algorithm A for recognising L is in the class ZPP (*zero-error probabilistic polynomial time*) if it returns the correct answers, "yes" or "no", or possibly "don't know", and there is a constant $c < 1/2$ such that the probability of returning "don't know" is at most c. It must also run in* guaranteed *polynomial time.*

Theorem 2 *The two definitions are equivalent.*

(1) ⇒ (2) Let T be the expected running time of the ZPP (1) algorithm on the input. Run it for time $2T$, and if it produces an answer, return it. If there is no answer, say "don't know". The probability that it will stop before $2T$ is at least $1/2$, so we have a ZPP (2) algorithm.

(2) ⇒ (1) Run the (2) algorithm, and return its answer, unless it says "don't know", when we run again, until we get a definite answer.

2.12.4 Atlantic City

Polynomial time with high probability and right result with high probability. Though many programs fall into this class, JHD doesn't know any underpinning theory.

2.13 SAT Solving

2.13.1 Cook's Theorem

This states that SAT is NP-complete [Coo66].

So *any* NP problem is polynomially transformable to SAT. Nevertheless *in practice* many SAT problems are easily soluble.

For example, since 2000, every car produced by Mercedes-Benz (and all the other German manufacturers) has been the output of a SAT solver [KS00]. The SAT solver verifies all sorts of things, that used to be bespoke logic.

1. The choice of options is consistent.
2. `UK` → `left-hand drive` etc.
3. `manual` → `gear lever` etc.
4. $X \wedge Y \wedge Z \rightarrow \neg($`small alternator`$)$ etc.
5. The SAT solver has a variable for each possible part so generates the complete list of parts etc.

2.13.2 Notation

variables x_i (but we may write x, y, z) $1 \leq i \leq n$.

Operators $\wedge$ for "and" and $\vee$ for (inclusive) "or".

Negation $\overline{x_i}$ rather than $\neg x_i$. Because $\overline{x \vee y}$ is equivalent to $\overline{x} \wedge \overline{y}$ etc., ew can insist that negation is only applied to variables, hence the next concept.

Literal Either an x_i or an $\overline{x_i}$.

Clause $C_i = l_{i,1} \vee l_{i,2} \vee \cdots l_{i,n_i}$ where the $l_{i,j}$ are literals.

Dr Uncu Can have $n_i = 3$ (3-SAT).

CNF Conjunctive Normal Form

$F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$; or $S := \{C_i\}$ and $F = \bigwedge S$

2.13.3 How to solve SAT

We assume we have an expression in CNF. There have been many developments, of which the following list is a small subset. There are annual contests (see e.g. [BFH+21]) and steady improvements.

1. Brute force enumeration of all ^{n} choices for x_i
2. Slightly lazy enumeration in the hope that choices for $x_, \ldots, x_k$ are sufficient for some $k < n$ (2^{n-k} choices for the price of one!)
3. Resolution [DP60, §4]
4. DPLL [DLL62]
5. CDCL [MSS96]
6. Restarts [LSZ93]
7. "Two watched literals" [MMZ+01]

Let us look at some of these in more detail.

2.13.4 Resolution [DP60, §4]

while $\{x_i\} \neq \emptyset$

1. **If** Some clause is just x_i, set x_i to true, remove all clauses containing x_i from S (as they are satisfied), and delete all $\overline{x_i}$ from clauses in S.
2. **If** Some clause is just $\overline{x_i}$, set x_i to false, remove all clauses containing $\overline{x_i}$ from S, and delete all x_i from clauses in S.
3. **If** There are no $\overline{x_i}$ in clauses in S, set x_i to true, remove all clauses containing x_i from S
4. **If** There are no x_i in clauses in S, set $\overline{x_i}$ to true, remove all clauses containing $\overline{x_i}$ from S
5. **Let** $S = S_{x_i} \cup S_{\overline{x_i}} \cup S_{\text{free}}$ (with x_i, $\overline{x_i}$ or neither, for some choice[1] of x_i).

   Write $\bigwedge S_{x_i}$ as $x_i \vee T$, $\bigwedge S_{\overline{x_i}}$ as $\overline{x_i} \vee U$, where T and U are conjunctions.

   $S = S_{\text{free}} \cup \{T \vee U\}$ (needs rewiting in CNF), and we have one fewer variable. This works because if T is true, we can afford to set x_i to false, then $\overline{x_i}$ is true and U is unnecessary, and *vice versa*.

[1] The algorithm terminates for any choice, but it may make a big difference in running time in practice.

The bad news is that if $|S_{x_i}| = \alpha$, $|S_{\overline{x_i}}| = \beta$ then $T \vee U$ writes as $\alpha\beta$ clauses, typically each twice as long. A folk theorem says that you can get by with $\alpha + \beta$ clauses, but still twice as long

2.13.5 DPLL [DLL62, §4]

while $S \neq \emptyset$

1. **If** Some clause is just x_i, set x_i to true, remove all clauses containing x_i from S, and delete all $\overline{x_i}$ from clauses in S.
2. **If** Some clause is just $\overline{x_i}$, set x_i to false, remove all clauses containing $\overline{x_i}$ from S, and delete all x_i from clauses in S.
3. **If** There are no $\overline{x_i}$ in clauses in S, set x_i to true, remove all clauses containing x_i from S
4. **If** There are no x_i in clauses in S, set $\overline{x_i}$ to true, remove all clauses containing $\overline{x_i}$ from S
5. Pick some x_i compute DPLL($S \cup \{x_i\}$). If this works, return DPLL($S \cup \{x_i\}$) with $x_i = T$
6. Otherwise return DPLL($S \cup \{\overline{x_i}\}$) with $x_i = F$

NB Can do $\overline{x_i}$ and x_i in either order

Essentially Resolution meets lazy brute force.

2.13.6 Conflict Directed Clause Learning [MSS96]

Consider DPLL, which is essentially intelligent guessing.

Assume we were assigning values in the order $x_1, x_2, \ldots$, and we first get UNSAT after assigning to x_k (say at clause C) and $\overline{x_k}$ (say at clause C')

So the values we are assigned to $x_1, x_2, \ldots, x_{k-1}$ were (collectively) not all right: there is at least one error.

But not all of these $x_,x_2, \ldots, x_{k-1}$ need occur in C or C'. Suppose x_m is the last one to really feature in C or C'.

Then we can "backjump" to level x_m and change our decision here

In this context we have a contradiction $l_{i_1} \wedge l_{i_2} \wedge \cdots l_m$, so we have learned $\overline{l}_{i_1} \vee \overline{l}_{i_2} \vee \cdots \overline{l}_m$. We *may* add this to S (different SAT solvers have different heuristics here).

2.13.7 Restarts [LSZ93]

One problem is that we may be working on some subset of S but elsewhere in S (where we aren't currently looking) we have $\{x \vee y, x \vee \overline{y}, \overline{x} \vee y, \overline{x} \vee \overline{y}\}$. *If* we were to look here, we would very rapidly get UNSAT.

Hence *from time to time* we stop and start again in a different place.

Theorem 3 ([LSZ93]) *Let A be a Las Vegas algorithm which we run under a strategy S which says "run for t_1, then restart for t_2, ... Let the expected running time this way be $T(A,S)$ and $\ell_A = \inf_S T(A,S)$ the best time. Then the strategy S^{univ} has $T(A,S^{univ}) = \mathcal{O}(\ell_A \log \ell_A)$. $S^{univ} = [1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,\ldots]$*

This is normally applied to CDCL *without* throwing away the learned clauses (or throwing away some but keeping others) so in theory the theorem doesn't apply, because we have changed the problem, albeit to an equivalent one.

2.13.8 "Two watched literals" [MMZ+01]

SAT solvers represent a clause as a vector of literals (x_i represented as ia, $\overline{x}_i$ as $-i$). These are normally not changed, and a separate data structure represents the current state of the x_i (positive/negative/unassigned). S can be many megabytes, much more than cache, so we end up running at main memory speed.

Instead represent each clause C as a small (fixed-size, maybe 8 or 16 bytes) header, including "two watched literals", i.e. two literals that occur in C, and currently don't have values (`true`/`false`) assigned. As long as this remains the case, C doesn't help us. If one gets assigned, then find another unassigned literal in Ci, which is the only time we look at the whole of C.

If we can't then there's only one unassigned literal, so we know it must be true, because the clause has to be true (and isn't already).

The net result is that we spend most of the time with the headers, a much smaller data structure. JHD has observed performance drops as this ceases to fit entirely in cache. Often believed to be worth $\times 10$ in performance.

2.13.9 Schur Number Five [Heu18]

The Schur number k, denoted by $S(k)$, is defined as the largest number n for which there exists a k-coloring of the positive numbers up to n with no monochromatic solution of $a + b = c$.

$S(2) = 4$: Say 1=red. 1+1=2 so 2 is blue. so 4 is red. 1+3=4 so 3 is blue. 1+4=5, so 5 must be blue, but then 2+3=5 is monochromatic.

Also S(3)=13, S(4)=44 (computer search 1965).

Theorem 4 ([Heu18, 14 CPU years]) $S(5) = 160$.

Various claims from [Heu18].

- We constructed a propositional formula that is satisfiable if and only if $S(5) \geq 161$. Our proof of unsatisfiability for this formula is over two petabytes in size
- We certified the proof using a program formally verified by ACL [KSM11], thereby providing high confidence in the correctness of our result
- We enumerated all 447 113 088 five-colorings of the numbers to 160 without a monochromatic $a + b = c$.

- We designed a decision heuristic that allows solving Schur number problems efficiently and enables linear-time speedups even when using thousands of CPUs
- We developed an efficient hardness predictor for partitioning a hard problem into millions of easy subproblems

2.13.10 Satisfiability Threshold Conjecture

The *satisfiability threshold conjecture* asserts if Φ is a formula drawn uniformly at random from the set of all k-CNF formulas with n variables and m clauses, there exists a real number r_k such that $\lim_{n\to\infty} Pr(\Phi \text{ is satisfiable})$ vanishes for $m/n > r_k$, and is equal to one for $m/n < r_k$.

Proved [CO14] with $r_k = 2^k \log 2 - \frac{1}{2}(1 + \log 2) + o(1)$.

Common belief that SAT is hard when m/n is close to r_k.

That seems to be true for random problems: "industrial" ones behave differently [FKRS17].

Car manufacturing [KS00] has a lot of $P \to Q$ $(\overline{P} \vee Q)$: wiper $\to$ wiper motor etc. Figure 2.1 shows the phase transition, and time takes to solve, for $k = 3, 4$.

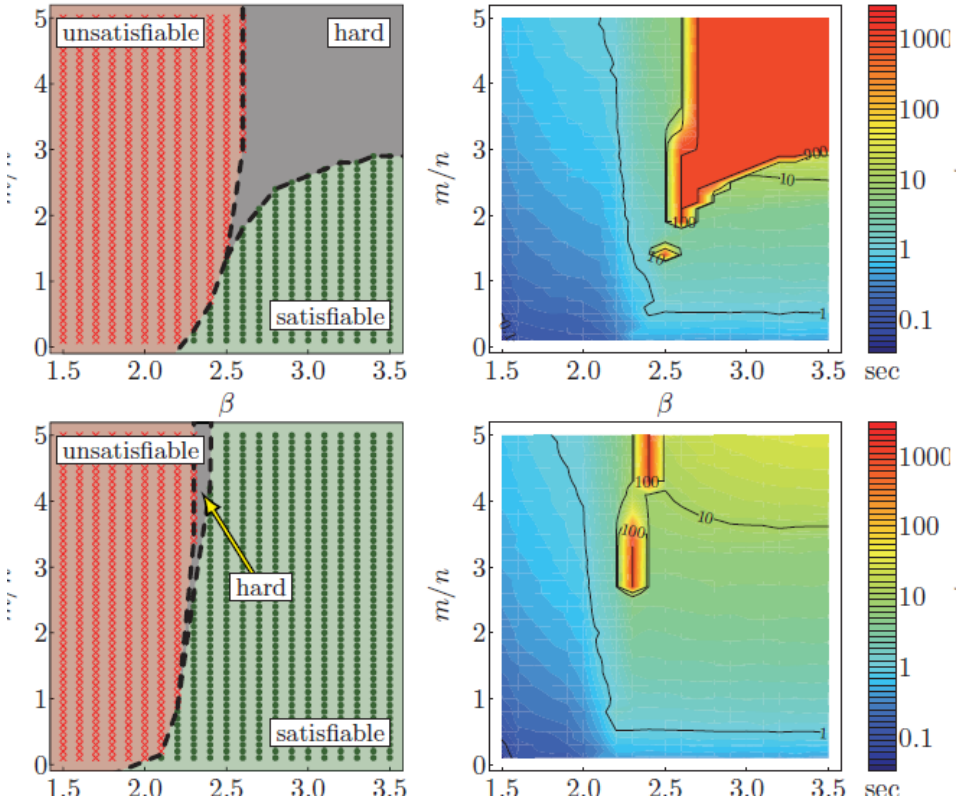
The variable x_i occurs in a clause with probability

$$\frac{1}{n}\frac{\beta - 2}{\beta - 1}\left(\frac{n}{i}\right)^{1/(\beta-1)}$$

[except that I think β in the graph is offset by 1].

Note that 4-SAT looks very different from 3-SAT: a 4-SAT clause can be translated into a 3-SAT clause, but a *random* 4-SAT doesn't become a *random* 3-SAT.

Figure 2.1: $k3$ (top) and $k = 4$, $n = 10^6$: [FKRS17, Figure 3]



Chapter 3

Lecture Notes 3

1. Cook's Theorem: formulation

Theorem. *Boolean Satisfiability problem is NP-complete.*

Actually we will prove that CNF-satisfiability problem is NP-complete.

2. Cook's Theorem: idea of the proof

We will use the fact that a Boolean formula has two different faces. Firstly, a formula is a formal object, combination of symbols, constructed from the letters of a certain alphabet using certain rules. Secondly, a formula is a sentence, i.e., a meaningful statement, if the variables are sentences.

We are going to represent an arbitrary problem from the class NP by the Nondeterministic Turing Machine M which solves this problem with polynomial complexity. To prove the theorem we need to produce a function f from the definition of the polynomial transformation. An argument x for this function will be an input of M, while the value $f(x)$ will be a Boolean formula F expressing the statement that M accepts x. If M does accept x, then $F = f(x)$ is satisfiable, else F is not satisfiable.

3. Proof: representation of the language and description of variables

Fix a **Yes** − **No** problem (language) $\mathcal{L} \in NP$. By the definition of the class NP, there is a Nondeterministic Turing Machine M for solving $\mathcal{L}$ with polynomial complexity. The machine M *completely determines* $\mathcal{L}$, and, in this sense, is identical to $\mathcal{L}$.

As any NTM, the machine M has

$\Gamma = \{s_0 = \mathrm{b}, s_1, s_2, \ldots, s_v\}$, *the tape alphabet*;

$Q = \{q_0, q_1 = q_Y, q_2 = q_N, q_3, \ldots, q_r\}$, the set of *states*;

δ, the *transition function* such that

$$\delta(q_k, s_l) = (q_{k'}, s_{l'}, \Delta),$$

where $\Delta \in \{-1, 0, 1\}$ is playing the role of $T \in \{L, S, R\}$ in our previous description of NTM.

Let $p(n)$ be a polynomial such that the complexity $T_M(n)$ of M satisfies the inequality

$$T_M(n) < p(n)$$

for all $n \geq 1$.

The formula $F = f(x)$ that we are building contains the following variables. In what follows, a comment that follows the introduction of a group of variables explains their interpretation in F; the words "At the moment $i \ldots$" mean "After execution of the step number i **of the verification stage** ...

$Q[i,k]$, $\quad 0 \leq i \leq p(n)$, $0 \leq k \leq r$. At the moment i the machine M is in the state q_k.

$H[i,j]$, $\quad 0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$. At the moment i the working head observes the cell j.

$S[i,j,k]$, $\quad 0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq v$. At the moment i the cell j contains the letter s_k.

Observe that any computation with input x induces on the set of variables a certain truth assignment assuming that at the moment 0 on the tape of M the input word x is written in the the cells from 1 to n, while the guess-word w is written (from right to left) in cells from -1 to $-|w|$.

Note that an arbitrary assignment of truth values to variables not necessarily corresponds to a computation. For instance, assignment

$$Q[i,1] - \text{true}; \quad Q[i,2] - \text{true}$$

means that at the moment i the machine M is simultaneously in the state q_1 and in the state q_2, which can't be in any computation.

We are going to construct a formula F in a conjunctive normal form such that the truth assignment to variables makes F true if and only if this assignment is induced by a certain accepting computation which uses not more than $p(n)$ steps for verification.

4. Proof: description of the formula

Formula F, being in CNF, is a conjunction of disjunctions of literals. We distinguish six groups $G_1, \ldots, G_6$ of disjunctions, here are their interpretations:

G_1: At any moment i the machine M is in the exactly one state.

G_2: At any moment i the working head observes the exactly one cell.

G_3: At any moment i every cell contains the exactly one letter from Γ.

G_4: At the moment 0 the computation is in the initial configuration of the verification stage with input x.

G_5: Not later than after $p(n)$ steps the machine M adopts the state q_Y.

G_6: For any moment i, the configuration of M at the moment $i+1$ is obtained from the configuration at the moment i by a single application of the transition function δ.

It is clear that the disjunctions in groups $G_1, \ldots, G_6$ are simultaneously true if and only if M accepts the input x. We now formally describe each group. To understand why the disjunction groups have the required interpretation it may be helpful to recall two well-known logical equivalences:

$$\neg(A \wedge B) \quad \text{equivalent to} \quad \neg A \vee \neg B \tag{1}$$

$$A \to B \text{ equivalent } \neg A \vee B. \tag{2}$$

$G_1: \quad Q[i,0] \vee Q[i,1] \vee \cdots \vee Q[i,r], \quad 0 \le i \le p(n),$
$\neg Q[i,j] \vee \neg Q[i,j'], \quad 0 \le i \le p(n),\ 0 \le j < j' \le r.$

$G_2: \quad H[i,-p(n)] \vee H[i,-p(n)+1] \vee \cdots \vee H[i,p(n)+1], \quad 0 \le i \le p(n),$
$\neg H[i,j] \vee \neg H[i,j'], \quad 0 \le i \le p(n),\ -p(n) \le j < j' \le p(n)+1.$

$G_3: \quad S[i,j,0] \vee S[i,j,1] \vee \cdots \vee S[i,j,v], \quad 0 \le i \le p(n),\ -p(n) \le j \le p(n)+1,$
$\neg S[i,j,k] \vee \neg S[i,j,k'],\ 0 \le i \le p(n),\ -p(n) \le j \le p(n)+1,\ 0 \le k < k' \le v.$

$G_4: \quad Q[0,0], H[0,1], S[0,0,0],$
$S[0,1,k_1], S[0,2,k_2], \ldots, S[0,n,k_n],$
$S[0,n+1,0], S[0,n+2,0], \ldots, S[0,p(n)+1,0]$, where $x = s_{k_1} s_{k_2} \cdots s_{k_n}$.

$G_5: \quad Q[p(n),1].$

Let us analyse, for example, group G_1. The first $p(n)+1$ of these disjunctions are simultaneously true if and only if *at any moment* i the machine M is in *at least* one state. The rest disjunctions of G_1 are simultaneously true if and only if *at no moment* i M is in *more than one* state (here (1) might be helpful). Thus, the group G_1 indeed does what was claimed.

The analysis for groups $G_2 - G_5$ is very similar and is left as an exercise.

Now we describe the group G_6. We subdivide G_6 into two subgroups: $G_{6.1}$ and $G_{6.2}$.

$G_{6.1}: \quad \neg S[i,j,l] \vee H[i,j] \vee S[i+1,j,l], \quad 0 \le i \le p(n), -p(n) \le j \le p(n)+1, 0 \le l \le v.$

Interpretation: if the working head at the moment i *does not* observe cell j, then the letter in the cell j will not change at the moment $i+1$ (by (1) and (2) the disjunction is equivalent to

$$(S[i,j,l] \wedge \neg H[i,j]) \to S[i+1,j,l]).$$

$G_{6.2}$: For every 4-tuple i, j, k, l such that

$$0 \leq i < p(n),\ -p(n) \leq j \leq p(n)+1,\ 0 \leq k \leq r,\ 0 \leq l \leq v$$

the following three disjunctions:

$$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee H[i+1, j+\Delta],$$

$$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee Q[i+1,k'],$$

$$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee S[i+1,j,l'],$$

where Δ, k', l' are defined as follows:

if $q_k \in Q \setminus \{q_Y, q_N\}$, then $\delta(q_k, s_l) = (q_{k'}, s_{l'}, \Delta)$,

if $q_k \in \{q_Y, q_N\}$, then $\Delta = 0, k' = k, l' = l$.

Interpretation: Passing from one configuration of M to the next is done according to transition function δ.

The resulting formula $F = f(x)$ is now

$$F = G_1 \wedge G_2 \wedge \cdots \wedge G_6.$$

If $x \in \mathcal{L}$, then there is an accepting computation for x on M performing at most $p(n)$ steps, and this computation induces the truth—false value assignment for variables such that F becomes true. Conversely, formula F is designed in such a way that any truth—false assignment for variables (leading to the value **true** for F) corresponds to a certain accepting computation for an input x on M. Thus, $F = f(x)$ is satisfiable if and only if $x \in \mathcal{L}$.

The proof that F can be constructed from x (and M) by a (deterministic) TM in time polynomial in $|x|$ is straightforward.

Cook's theorem is proved.

Chapter 4

Lecture Notes 4

noindent **1. Structure of the class** NP

Recall that it is not known whether or not $P = NP$, the widely accepted hypothesis being that this equality is wrong. Let $NPC \subset NP$ denote the class of all NP-complete problems, and denote $NPI := NP \setminus (NPC \cup P)$.

Theorem. *If $P \neq NP$, then:*

(1) $NPI \neq \emptyset$*;*

(2) there exist the problems $A, B \in NPI$ such that neither $A \propto B$, nor $B \propto A$.

Proof is nontrivial, we don't consider it in this unit (see the book by M. Garey and D. Johnson).

The item (2) of the Theorem states that under the hypothesis $P \neq NP$ the set NPI is divided into more that one equivalence classes with respect to polynomial transformation.

2. Complements to languages

Definition. *Let $\mathcal{L} \subset \Sigma^*$ be a language over an alphabet Σ. The set of words $\overline{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}$ is called the* complement *to $\mathcal{L}$.*

Observe that $\overline{\mathcal{L}}$ is not necessarily a *language*, i.e. a set of words generated by a grammar (or accepted by a TM). For all problems from NP the complement is indeed a language, however it is not at all obvious that if $\mathcal{L} \in NP$, then $\overline{\mathcal{L}} \in NP$. That is because the definition of NTM (unlike the definition of **Yes - No** TM) is non-symmetric with respect to **Yes - No** output.

Example. For the problem *CNF SAT*, the complement is the following **Yes - No** problem.

Input: Boolean formula F in conjunctive normal form.

Output: **Yes** if F is *not* satisfiable (i.e., identically false), else **No**.

There is no apparent way of taking advantage of the non-determinism (i.e., guessing module) for solving this problem. It seems to be nothing essentially better than just examining one by one every **true/false** assignment to all variables and checking whether it gives the value **false** to F.

3. Class co-NP

Definition.

(1) $\text{co}-NP = \{\overline{\mathcal{L}} : \quad \mathcal{L} \in NP\}$;

(2) $\text{co}-P = \{\overline{\mathcal{L}} : \quad \mathcal{L} \in P\}$;

Hypothesis. $NP \neq \text{co}-NP$.

This hypothesis is "stronger" than $P \neq NP$ since obviously $P = \text{co}-P$, thus if $P = NP$, then $NP = \text{co}-NP$.

Theorem. *Let $\mathcal{L}$ be an NP-complete problem such that $\overline{\mathcal{L}} \in NP$. Then $NP = \text{co}-NP$.*

Proof is relatively straightforward. We don't consider it in this course.

4. Some famous candidates for problems in NPI

(1) **Graph Isomorphism**

Input: Two graphs $G = (V, E)$ and $G' = (V', E')$.

Output: **Yes** if there is a bijective (one-to-one) map (function), called isomorphism, $f : \quad V \longrightarrow V'$ such that $(v, w) \in E$ is equivalent to $(f(v), f(w)) \in E'$; else **No**.

The status of Graph Isomorphism is unknown. Despite many efforts no polynomial time algorithm was found. On the other hand, the problem seems to be too "rigid" to be NP-complete. Compare it with the following NP-complete problem *Isomorphism to Subgraph*.

Input: Two graphs $G = (V, E)$ and $G' = (V', E')$.

Output: **Yes** if there is a *subgraph* in G which is isomorphic to G'; else **No**.

Isomorphism to Subgraph is NP-complete problem which is proved by polynomially transforming to it the q-clique problem (take as G' the *complete* graph with q vertices).

(2) **Linear Programming**

Input : System of linear *inequalities* in several variables with integer coefficients.

Output : **Yes** if the system is consistent (i.e., has a solution) in real numbers, else **No**.

The status of Linear Programming was open for a long time until in late seventies a polynomial-time algorithm was discovered. Even before that it seemed quite unlikely for the problem to be NP-complete since the complement to Linear Programming is in NP (follows from so-called Duality Theorem in linear programming). If Linear Programming was NP-complete, then, by Theorem from previous section, $NP = \text{co}-NP$, which is probably wrong.

5. Search problems

The computational problems we considered so far were **Yes - No** problems, which were associated with languages of inputs with the output **Yes**. It appears important, however to consider also the problems with more complicated output. In fact, natural modifications of our basic NP-complete problems require a search of an object whose existence is checked. For instance, a modification of the Travelling Salesman problem asks to produce one of the shortest tours, a modification of SAT requires construction of a satisfying assignment. Observe that these modifications (we shall call them *search problems*) are "harder" than the corresponding **Yes - No** problems in the sense that any algorithm for a search problem automatically solves the **Yes - No** version.

The theory we had developed so far is not very well adjusted for search problems. Firstly, the concept of nondeterministic Turing Machine is formulated for **Yes - No** problems. Secondly, and most importantly, the definition of $\propto$ relation is very specific, formulated essentially in terms of languages. We will start with generalization of the concept of polynomial transformation.

6. Turing transformation: informal definition

Let us first recall that for two languages $\mathcal{L}_1, \mathcal{L}_2$ the relation $\mathcal{L}_1 \propto \mathcal{L}_2$ informally means that there is an algorithm such that:

(1) for given input word x_1 of the first problem the algorithm computes an input word x_2 of the second problem, herewith $x_1 \in \mathcal{L}_1$ if and only if $x_2 \in \mathcal{L}_2$;

(2) the algorithm uses a "subroutine" to solve the problem 2 on the input x_2 and finds output y_2;

(3) y_2 is also the answer for problem 1.

The algorithm uses polynomial time for all its work, except the "subroutine". Now we generalize this informal definition for search problems $\mathcal{L}_1$ and $\mathcal{L}_2$:

We say that $\mathcal{L}_1$ is Turing transformable *to $\mathcal{L}_2$ if there exists an algorithm which for a given input x_1 of $\mathcal{L}_1$ computes the required output using (zero, or one, or more times) an algorithm for the problem $\mathcal{L}_2$ as a subroutine. The running time of the algorithm is polynomial if each call of the subroutine is counted as one elementary step.*

7. Oracle Turing Machine

Oracle Turing Machine (OTM) differs from ordinary (deterministic) TM with arbitrary output in the following way. Apart from the usual working tape, OTM has an additional *oracle tape* equipped with the *oracle read-write head* which is attached to the control module. OTM works like ordinary TM, but can at an arbitrary moment adopt (via "asking" state) a special *asking mode* in which:

(1) the oracle head writes a word (which was generated during the previous normal work of the machine) on the oracle tape (oracle input);

(2) after (1) is done, the oracle head writes a word on the oracle tape (oracle output).

Passage from (1) to (2) is determined by a function

$$g: \quad \Gamma^* \longrightarrow \Gamma^*, \tag{$*$}$$

where Γ is the tape alphabet. Each application of asking mode counts as one step for complexity. Since OTM depends on a concrete build-in function g, the notation OTM_g might be more appropriate. The *formal* definition of OTM and its complexity is straightforward.

8. Turing transformation: definition

Definition. *Let $\mathcal{L}_1, \mathcal{L}_2$ be search problems, herewith $\mathcal{L}_2$ is a problem of computing a function g of the form $(*)$. We say that $\mathcal{L}_1$ is* Turing transformable *to $\mathcal{L}_2$ (notation: $\mathcal{L}_1 \propto_T \mathcal{L}_2$) if there exists OTM_g for solving $\mathcal{L}_1$ using as oracle the function g and having polynomial complexity.*

Theorem 1. *For any two* **Yes - No** *problems (languages $\mathcal{L}_1, \mathcal{L}_2$), if $\mathcal{L}_1 \propto \mathcal{L}_2$, then $\mathcal{L}_1 \propto_T \mathcal{L}_2$.*

Proof. Trivial, since as we had seen before, $\propto$ is a particular case of $\propto_T$.

Remark. It is not known whether the statement inverse to Theorem 1 is true or not *for languages.*

Theorem 2. *The relation $\propto_T$ is transitive.*

Proof is straightforward.

9. NP-hard problems

Definition. *A search problem $\mathcal{L}$ is NP-hard if for any* language $\mathcal{L}' \in NP$ *the following holds:*

$$\mathcal{L}' \propto_T \mathcal{L}. \tag{$**$}$$

Theorem. *A search problem $\mathcal{L}$ is NP-hard if and only if for at least one (thus, for any) NP-complete language $\mathcal{L}'$ relation $(**)$ is true.*

Proof. If $\mathcal{L}$ is NP-hard, then for any language from NP, in particular, for any NP-complete language $\mathcal{L}'$ the relation $(**)$ is true.

Conversely, if there exists an NP-complete language $\mathcal{L}'$ such that $(**)$ is true, then for any language $\mathcal{L}'' \in NP$ we have

$$\mathcal{L}'' \propto \mathcal{L}' \propto_T \mathcal{L},$$

which, by Theorems 1 and 2 of Section 8, implies that $\mathcal{L}'' \propto_T \mathcal{L}$, thus $\mathcal{L}$ is NP-hard by the definition.

Example. Consider a modification of Travelling Salesman problem as a search problem in which the output is the tour of the smallest length. Call this modification *Optimal Travelling Salesman* (OptTS), as opposed to **Yes - No** modification (TS) which involves a threshold.

Theorem. *$TS \propto_T OptTS$, therefore $OptTS$ is NP-hard.*

Proof. The OTM for TS with threshold B first uses the oracle to solve the OptTS with the same set of cities and distances as the input of TS (one step of computation), then in polynomial time finds the length l of the produced optimal tour and checks whether $B \geq l$. If the latter inequality is true, then the answer is **Yes**, else – **No**.

Observe that this example does not use the relation $\propto_T$ in full force, addressing the oracle only once.

Chapter 5

Lecture Notes 5: Approximation Algorithms

Many NP-complete or NP-hard problems are so important practically, that they simply can not be abandoned because of practical insolvability. The first natural thing to do when a problem had turned out to be NP-hard is to revise the initial setting because it is possible that actually a special case of the problem is needed. Another possibility is that we may need to solve the problem for the input data of a relatively small size, so that even an exponential-time algorithm is acceptable. Finally, one can try to solve the problem *approximately*. The latter approach is suitable for *optimization search problems*.

Examples of optimization search problems.

(1) *Travelling Salesman* in optimization setting asks to find a *shortest* tour.

(2) *Clique* requires to find a *largest* (containing the largest number of vertices) clique in a graph.

(3) *Vertex Covering* consists in finding the *smallest* vertex covering in a graph.

2. Ratio bound and relative error

In an optimization problem we always have an *objective function* describing the "quality" of a candidate for a solution. In the above examples these are respectively: length of a tour, size of a clique (number of vertices), size of a covering. In what follows we will assume (as in the examples) that the values of an objective function are always positive.

An approximation algorithm finds a solution which may be not *optimal*, that is, the one at which the objective function attains its maximal or minimal value, but is in some sense "close" to optimal. We formalize the measure of "closeness" as follows. Let C^* denote the value of the objective function at an optimal solution.

Definition 1. *Approximation algorithm for an optimization problem has* ratio bound $\rho(n)$ *if for any input of the size* n *the value* C *of the objective function at the approximate solution, produced by this algorithm, satisfies the inequality*

$$\max\{C/C^*,\, C^*/C\} \leq \rho(n).$$

Observe that for maximization problem $0 < C \leq C^*$, thus

$$\max\{C/C^*,\, C^*/C\} = C^*/C,$$

while for minimization problem $0 < C^* \leq C$, thus

$$\max\{C/C^*,\, C^*/C\} = C/C^*.$$

Obviously, $\rho(n) \geq 1$. Equality $\rho(n) = 1$ means that approximate solution is optimal. Of course, a large ratio bound might mean that approximate solution is much worse than an optimal one.

Definition 2. *For* C, C^* *and* n *defined as above, the* relative error *is the fraction*

$$|C^* - C|/C^*.$$

Any function $\varepsilon(n)$ *with*

$$|C^* - C|/C^* \leq \varepsilon(n)$$

is called relative error bound.

Easy computation shows that $\rho(n) - 1 \geq |C^* - C|/C^*$, i.e., $\rho(n) - 1$ is a relative error bound.

For many problems approximation algorithms with polynomial complexity were developed, in which $\rho(n) = \text{const}$, i.e., the ratio bound does not depend on the size of the input. The remaining part of this LN's describes two of such problems and algorithms, with $\rho(n) = 2$.

3. Vertex Covering problem

First we recall the *Vertex Covering* in optimization version.

Input : Graph $G = (V, E)$.

Output : A minimal $S \subset V$ such that for every $(v, w) \in E$ either $v \in S$ or $w \in S$ (here "minimal" means having the smallest possible number of elements).

Approximation algorithm for Vertex Covering

(1) $S \leftarrow \emptyset$;

(2) $E' \leftarrow E$;

(3) while $E' \neq \emptyset$ do
let (v, w) be arbitrary edge of E'
$S \leftarrow S \cup \{v, w\}$
remove from E' every edge having either v or w as a vertex;

(4) return S.

An example of how this algorithm works is presented on a separate sheet.

Theorem.

(1) The approximation algorithm for Vertex Covering is correct (i.e., indeed produces a covering).

(2) The algorithm has polynomial complexity.

(3) The algorithm has a ratio bound $\rho(n) = 2$.

Proof.

(1) Correctness of the algorithm is obvious since it loops until every edge is covered by some vertex.

(2) Straightforward observation.

(3) Let A denote the set of all edges that are chosen during the execution of the item (3) of the algorithm. No two edges of A have a common vertex, since once the edge (v, w) is chosen, all edges having either v or w as vertices are deleted. Thus, each execution of (3) adds exactly two new vertices to S and $|S| = 2|A|$. Any covering, in particular optimal S^*, must cover every edge in A, thus any covering has at least $|A|$ vertices. So,

$$|S^*| \geq |A| = |S|/2.$$

It follows that

$$|S|/|S^*| \leq 2.$$

4. Travelling Salesman with triangle inequality

The optimization version of Travelling Salesman problem if formulated as follows.

Input : Integral matrix of distances

$$\|d_{ij}\|,\ 1 \leq i, j \leq m,\ d_{ij} > 0.$$

Output : Permutation $i_1, i_2, \ldots, i_m$ (called "tour") of numbers $1, 2, \ldots, m$ such that the value

$$\sum_{1 \le j < m-1} d_{i_j i_{j+1}} + d_{i_m i_1}$$

is minimal possible.

Travelling Salesman with triangle inequality has an additional property of the input: it is required that for any three indices i, j, k the following inequality holds:

$$d_{ij} \le d_{ik} + d_{kj}.$$

It can be proved (not a part of this course) that *Travelling Salesman with triangle inequality* is NP-hard.

To describe an approximation algorithm for this problem we need a subroutine for a problem called *Minimum Spanning Tree*, so we first give a definition of a tree and then formulate the problem.

Definition. *A* tree *is any connected graph without cycles. Sometimes a vertex is designated in the tree, called the* root.

Input : Graph $G = (V, E)$, weight function $w(u, v) \ge 0$ for every edge $(u, v) \in E$.

Output : Subgraph T of G with the set of vertices V, herein T is a tree and the value

$$\sum_{(u,v) \in T} w(u, v)$$

is minimal possible.

Theorem. *There is an algorithm, having polynomial complexity, for computing a minimum spanning tree for a graph.*

Proof can be conducted by presenting a concrete algorithm. There are two famous ones: of Kruskal and of Prim both of which are quite elementary and simple.

We need two different ways of listing all the vertices of a tree, called "Full Walk" and "Preorder Walk". An example of a Full Walk is shown on a separate sheet. This example describes sufficiently enough the concept of a Full Walk. Note that Full Walk visits every edge of the tree exactly twice. Preorder Walk is obtained from Full Walk by deleting from the Full Walk every non-first occurrence of a vertex.

4. Approximation algorithm for Travelling Salesman with triangle inequality

Let $G = (V, E)$ be a *complete* graph with the set of vertices $V = \{1, 2, \ldots, n\}$ and weights $w(i, j) = d_{ij}$ where d_{ij} are the distances from the input of *Traveling Salesman*.

(1) Select a vertex $r \in V$ to serve as a root for the future tree;

(2) Build a minimum spanning tree T for G with the root r using a polynomial-time minimum spanning tree algorithm as a subroutine;

(3) Construct the list L of vertices being a Preorder Walk of T;

(4) Return L as an approximate tour in G.

Theorem.

(1) The approximation algorithm for Travelling Salesman with triangle inequality is correct (i.e., produces a tour).

(2) The algorithm has polynomial complexity.

(3) The algorithm has a ratio bound $\rho(n) = 2$.

Proof.

(1) By its definition, the Preorder Walk is the Full Walk from which duplicates are deleted. The Full Walk includes *all* vertices of G, thus L consists of all vertices of G.

(2) Straightforward analysis.

(3) Let L^* denote an optimal tour, $d(L)$ be the length of L, $d(L^*)$ be the length of L^*.

Observe that a spanning tree for G can be obtained by deleting any edge from any tour. Thus, for a minimal spanning tree T, with sum of weights on all edges $d(T)$, the following inequality holds

$$d(T) \leq d(L^*).$$

Let W be a Full Walk of T, and $d(W)$ be its length. Since W visits every edge of T exactly twice, $d(W) = 2d(T)$. It follows that

$$d(W) \leq 2d(L^*).$$

By the triangle inequality, Preorder Walk L is not longer than W: $d(L) \leq d(W)$. As a result, we have:

$$d(L) \leq 2d(L^*),$$

or, equivalently,

$$d(L)/d(L^*) \leq 2.$$

5. No ratio bound for general Travelling Salesman

Theorem. *If $P \neq NP$ then there is no approximation algorithm for general Travelling Salesman with polynomial complexity and constant ratio bound.*

Proof. We show how to solve Hamiltonian Cycle (HC) problem in polynomial time using a hypothetical approximation algorithm, with polynomial complexity and constant ratio bound ρ, for general Travelling Salesman (TS), as a subroutine.

Let $G = (V, E)$ be an input of HC. Construct an input of TS with the set of cities V and the distance function:

$d(u, v) = 1$ if $(u, v) \in E$

$d(u, v) = \rho|V| + 1$ if $(u, v) \notin E$.

Clearly, this input of TS can be constructed from G in polynomial time.

If G contains a hamiltonian cycle, then its length will be $|V|$. If G *does not* contain a hamiltonian cycle, then there is a pair (u, v) of subsequent cities in any tour such that $(u, v) \notin E$, thus the size of any tour will be at least

$$(\rho|V| + 1) + (|V| - 1) = (\rho + 1)|V| > \rho|V|.$$

Thus, we can decide whether G contains a hamiltonian cycle by approximately solving the instance of TS. If the size of the solution tour is $\leq \rho|V|$, then G contains a hamiltonian cycle, else G has no hamiltonian cycles.

Chapter 6

Lecture Notes 6

1. Complexity lower bounds

A function $L(n)$ is called a *(complexity) lower bound* for a computational problem $\mathcal{L}$ in a given class $\mathcal{M}$ of models of computation if for any algorithm from $\mathcal{M}$ which solves $\mathcal{L}$ with complexity $T(n)$ the following relation holds:

$$T(n) = \Omega(L(n)).$$

Finding non-trivial lower bounds is an important part of a comprehensive analysis of an algorithm, which might save time spent on trying to construct a better algorithm.

Very few natural problems have proven lower bounds in the class of all Turing machines because these models are powerful. An example of the quadratic lower bound for the problem of recognizing *palindromes* in the class of one-tape, one-head Turing machines was mentioned earlier in this unit. If we assume $P \neq NP$ hypothesis, then any polynomial serves as a lower bound for any NP-complete problem in the class of all Turing machines. Clearly it should be easier to prove lower bounds for narrower classes $\mathcal{M}$ of algorithms. On the other hand such a class should be "natural enough" for the lower bound to be of some practical use.

This part of Lecture Notes describes a general method for proving complexity lower bounds for *algebraic computation trees*.

2. Algebraic computation trees: examples

An example of an algebraic computation tree appears in a proof of the $n \log n$ lower bound for the problem of *sorting by comparisons*.

Input : A sequence of real numbers $x_1, \ldots, x_n$.

Output : The permutation $i_1, \ldots, i_n$ of $1, 2, \ldots, n$ such that $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_n}$.

The only operation that is allowed, is the *comparison* of any two numbers in the input sequence.

An obvious straightforward algorithm has complexity (number of comparisons in worst case) $O(n^2)$. *Mergesort* algorithm, based on *divide-and-conquer* approach, has complexity $O(n \log n)$. We now prove that the lower complexity bound for sorting is $\Omega(n \log n)$.

Fix n. Any sorting by comparison algorithm with inputs of size n can be represented as a binary tree. At each vertex, except leaves, some two numbers from the input sequence are compared. Leaves correspond to all possible outputs, i.e., all permutations $i_1, \ldots, i_n$ of $1, 2, \ldots, n$. Complexity (number of comparisons needed for sorting of n numbers) is the *height* of the tree (number of edges in the longest branch from the root to a leaf).

The following proposition is easily proved by induction.

Proposition. If the number of leaves in a binary tree is ℓ, then the height is $\geq \log \ell$.

In the tree for a sorting by comparison algorithm the number of leaves is the same as the number of all permutations of $1, 2, \ldots, n$ which is $n! = 1 \cdot 2 \cdots n$. Then the Proposition implies that the complexity of the algorithm is at least $\log(n!)$. It follows from the *Stirling formula* that $\log(n!) = \Omega(n \log n)$ (it is also easy to prove it directly, using the obvious inequality $n! \geq (n/2)^{n/2}$). We conclude that the complexity of the algorithm is $\Omega(n \log n)$.

Another example of a tree model for an algorithm is the problem of *membership to a circle*:

Input : A point $(x, y) \in \mathbf{R}^2$.

Output : **Yes** if and only if (x, y) satisfies the inequality $X^2 + Y^2 \leq 1$.

On a separate sheet an algebraic computation tree is shown for solving this problem. The complexity of this algorithm (the height of the tree) is 5, i.e., a constant. This is quite natural since we assume that the size of the input is a constant ($= 2$).

Further, we wish to consider only the trees in which all vertices except leaves have exactly three children. To satisfy this requirement we can redraw the tree for membership to a circle. In the new tree at each vertex (except leaves) an arithmetic operation is conducted and the branching corresponds to the sign of the expression obtained. Observe that the complexity (height) of the new tree is still 5.

Exercise. Prove that recognizing membership to an n-dimensional ball defined by the inequality $X_1^2 + \cdots + X_n^2 \leq 1$ can be done on algebraic computation tree model with complexity $2n + 1$.

3. Algebraic computation trees: definition

Definition. *A subset $S \subset \mathbf{R}^n$ is called (basic) semialgebraic if it is of the form*

$$S = \{\mathbf{x} \in \mathbf{R}^n |\, f_1(\mathbf{x}) = \cdots = f_k(\mathbf{x}) = 0, f_{k+1}(\mathbf{x}) > 0, \ldots, f_{k+r}(\mathbf{x}) > 0\},$$

where f_i, $1 \leq i \leq k+r$ are polynomials in n variables, i.e., S is a set of all solutions of a system of equations and strict inequalities.

Algebraic computation tree $\mathcal{T}$ in variables $X_1, \ldots, X_n$ is a tree with the root v_0 such that to every vertex v (except leaves) an arithmetic operation (addition, subtraction or multiplication) and a polynomial f_v are attached. At the root v_0 the corresponding arithmetic operation, say $+$, is performed on a pair of variables, say X_i, X_j, and $f_{v_0} = X_i + X_j$ is the result of this operation. Let $v_0, v_1, \ldots, v_l$ be the sequence of vertices along the (unique) branch leading from the root v_0 to v_l. An arithmetic operation at v_l is performed on a pair from

$$X_1, \ldots, X_n, f_{v_0}, \ldots, f_{v_{l-1}}, a \in \mathbf{R}$$

and f_{v_l} is the result of this operation.

Every v has three children in $\mathcal{T}$ corresponding to the sign of f_v ($> 0, < 0, = 0$). Let $*_i \in \{>, <, =\}$ for $0 \leq i < l$ be the sign of f_{v_i}. Then to v_l one can assign a *semialgebraic* set

$$U_{v_l} = \{f_{v_0} *_0 0, f_{v_1} *_1 0, \ldots, f_{v_{l-1}} *_{l-1} 0\}.$$

Finally, to each leaf w of $\mathcal{T}$ an output **Yes** or **No** is assigned. We call U_w an *accepting* set if to w the output **Yes** is assigned. We say that $\mathcal{T}$ *tests the membership to the union of all accepting sets.*

The computation process works as follows. A specific point $x \in \mathbf{R}^n$ is taken as an input. Then the value $f_{v_0}(x)$ is computed and the sign of this value is determined. According to the sign, the algorithm goes to the corresponding son v_1 of v_0. Then the arithmetic operation attached to v_1 (i.e., the value f_{v_1}) is computed and so on. If the process eventually arrives to a **Yes**-vertex, then x belongs to an accepting set, and, therefore, to the union of all accepting sets.

Example. List all the accepting sets for the tree for the membership to a circle problem. Their union should coincide with $\{X^2 + Y^2 - 1 \leq 0\}$.

The complexity for algebraic computation tree model is the height of $\mathcal{T}$. A separate sheet shows a part of general algebraic computation tree.

Remarks.

1. We defined the tree model for **Yes-No**-problems, since in what follows we will consider only them. However, it's easy to extend the definition to the case of search problems.

2. If in each vertex the computation is of the kind $X_i - X_j$ then $\mathcal{T}$ is called *Decision computation tree.* Decision tree is, therefore, based entirely on comparisons between the components of the input vector.

4. *Distinctness* problem: upper bound

The following is the *Distinctness* problem.

Input : $x = (x_1, \ldots, x_n) \in \mathbf{R}^n$.

Output : **Yes** if all components of x are pairwise distinct, i.e., $x_i \neq x_j$ for all pairs i, j such that $i \neq j$, else **No**.

An algorithm for this problem having complexity $O(n \log n)$ first sorts the numbers $x_1, \ldots, x_n$, then checks all pairs of *neighbours* in the sorted sequence. It's clear that this algorithm can be represented in a form of algebraic computation tree of the height $O(n \log n)$ (compare with sorting).

In the remaining part of these notes we are going to develop a general method for proving lower complexity bounds for algebraic computation trees, and will apply this method to prove the $\Omega(n \log n)$ *lower bound* for *Distinctness*. Unlike a very elementary proof of $(n \log n)$-lower bound for sorting, the proof for *Distinctness* employs some modern mathematics and apparently no elementary proof is known.

5. Connected components of semialgebraic sets

Informally, a finite union W of semialgebraic sets is called *connected* if for every $x, y \in W$ there is a "continuous" curve in W containing both x and y. A formal definition can be found in textbooks on topology.

Definition. *Any maximal (with respect to the set-theoretical inclusion) connected subset of W is called* connected component *of W.*

Theorem. *Every finite union W of semialgebraic sets can be uniquely represented as a union of a finite number of its connected components:*

$$W = \bigcup_{1 \leq i \leq k} W_i,$$

which are finite unions of semialgebraic sets.

We are not proving the theorem in this course.

Examples.

1. The union of open intervals $W = (0, 1) \cup (2, 3) \subset \mathbf{R}$ is *not* connected. Intervals $(0, 1)$ and $(2, 3)$ are connected components of W.

2. The union $W = (0, 1) \cup (1, 2)$ is *not* connected with $(0, 1)$, $(1, 2)$ being W's connected components.

3. The union $W = (0, 1) \cup 1 \cup (1, 2) = (0, 2)$ is connected and is its own unique connected component.

4. The semialgebraic set $W = \{X \neq Y\} \subset \mathbf{R}^2$ (which also can be written in the form $W = \{X^2 - 2XY + Y^2 > 0\}$) is *not* connected and has two connected components: $\{X - Y > 0\}$ and $\{Y - X > 0\}$.

Lemma. *Projection of a connected set $W \subset \mathbf{R}^{n+m}$ on a coordinate subspace $\mathbf{R}^n$ is also connected.*

Proof is suggested as an exercise.

6. Lower bound for membership to a semialgebraic set: decision computation trees

Theorem. *Let the union Σ of all accepting sets of a* **decision** *computation tree $\mathcal{T}$ have $\nu(\Sigma)$ connected components. Then the height k of $\mathcal{T}$ (the complexity of testing membership to Σ) is $\Omega(\log(\nu(\Sigma)))$.*

Proof. Observe that any non-empty accepting set W of $\mathcal{T}$ is a set of all points in $\mathbf{R}^n$ satisfying a system of linear equations and strict linear inequalities, and is, therefore, a convex polyhedron in $\mathbf{R}^n$. It follows that $\nu(W) = 1$.

Since there is at most 3^k leaves in $\mathcal{T}$, we get

$$\nu(\Sigma) \leq 3^k. \tag{2}$$

Taking the binary logarithm in both parts of (2), we obtain the inequality

$$k \geq c(\log(\nu(\Sigma))),$$

for a constant $c > 0$, which proves the theorem.

7. Thom-Milnor's bound

Theorem. (R. Thom, J. Milnor) *Let a semialgebraic set W be defined by a system of equations and strict inequalities with polynomials of degree at most d in n variables, having m inequalities. Then the number of all connected components of W does not exceed $((m+1)d)^{cn}$ for a constant $c > 0$.*

The known proofs of Thom-Milnor's bound use complex arguments from differential and algebraic topology, we are not considering them in the present course.

Corollary. *Let W be a semialgebraic set satisfying the conditions of the theorem. Then the number of the connected components of a projection of W on any coordinate subspace does not exceed $((m+1)d)^{cn}$ for a constant $c > 0$.*

8. Lower bound for membership to a semialgebraic set: algebraic computation trees

Theorem. (M. Ben-Or) *Let the union Σ of all accepting sets of an* **algebraic** *computation tree $\mathcal{T}$ have $\nu(\Sigma)$ connected components. Then the height k of $\mathcal{T}$ (the complexity of testing membership to Σ) is $\Omega(\log(\nu(\Sigma)) - n)$.*

Proof. (OPTIONAL) Consider one of the accepting sets of $\mathcal{T}$

$$W = \{f_{v_0} = \cdots = f_{v_{k_1}} = 0,\ f_{v_{k_1+1}} > 0, \ldots, f_{v_k} > 0\}$$

(here the permutation of signs = and > could be arbitrary).

Observe that the degree $\deg(f_{v_q}) \leq 2^k$ for any $0 \leq q \leq k$, since *at most one* multiplication occurs at each step of a computation. Therefore, according to Thom-Milnor's bound, we have

$$\nu(W) \leq ((k - k_1 + 1)2^k)^{cn}. \tag{1}$$

Our aim is to solve (1) relative to k. We have two obstacles to do this, firstly k occurs in two different places in (1), secondly k is multiplied by n in the exponent. We shall overcome these difficulties by introducing new variables.

For each strict inequality $f_{v_q} > 0$ introduce a new variable Y_q, and write $f_{v_q} - Y_q^2 = 0$, which is equivalent to $f_{v_q} \geq 0$. Adding a new variable Z and an equation $Y_1 \cdots Y_{k-k_1} Z = 1$ reduces all strict inequalities in the definition of W into equations.

Now rewrite each of the equations as a system of equations of degrees at most 2, by adding at most $\log(\text{degree}) \leq \log(2^k) = k$ new variables. In all we had added at most $2k$ new variables. The new system of quadratic equations defines a set $W' \subset \mathbf{R}^{n+2k}$ such that W is the projection of W' on the subspace of the coordinates $X_1, \ldots, X_n$.

We have:

$$\nu(W) = \nu(\text{projection}(W')) \leq \nu(W') \leq 2^{c(n+k)}.$$

(Here, to pass from projection(W') to W' we have used the *corollary, Section 6.)

Since there is at most 3^k leaves in $\mathcal{T}$, we get

$$\nu(\Sigma) \leq 3^k 2^{c(n+k)}. \tag{3}$$

Taking logarithm in both parts of (3), we obtain the inequality

$$k \geq c'(\log(\nu(\Sigma)) - n),$$

for a constant $c' > 0$, which proves the theorem.

9. *Distinctness* problem: lower bound

Theorem. Distinctness *problem has a complexity lower bound* $\Omega(n \log n)$.

Proof. The union Σ of all accepting sets in this problem is the complement to the union of hyperplanes defined by all possible linear equations of the kind $X_i = X_j$ (where $i \neq j$), in other words, at a point $x = (x_1, \ldots, x_n) \in \Sigma$ all coordinates are pairwise distinct (compare with Example 4, Section 5). Herewith, the coordinates of x are somehow ordered:

$$x_{i_1} < \cdots < x_{i_n}.$$

It is easy to see that, at each point of a connected component Σ_i of Σ the order of the coordinates is the same. Indeed, suppose this is wrong, and for

two points $y = (y_1, \ldots, y_n)$, $z = (z_1, \ldots, z_n)$ we have $y_l < y_m$, but $z_l > z_m$. By the definition of connectedness, there is a continuous curve in Σ_i connecting y and z. Since the values of coordinates change continuously along the curve, there should be a point $w = (w_1, \ldots, w_n)$ on the curve such that $w_l = w_m$. Thus, $w \in \{X_l = X_m\}$, that is, w belongs to the complement of Σ, which is a contradiction.

We have proved that the number of connected components of Σ is not less than the number $n!$ of all permutations of n coordinates (in fact these numbers coincide). According to the theorem from Section 7, we get a lower bound $\Omega(\log(n!))$ or $\Omega(n \log n)$.

Chapter 7

Lecture Notes 7

1. Complex numbers

Complex numbers play a very special role in quantum computing. A complex number may have three principal representations.

1. As a pair of real numbers (a, b), traditionally written as $a+ib$, where $a, b \in \mathbf{R}$ and i is *a posteriori* interpreted as $\sqrt{-1}$.

Arithmetic operations are defined as follows.

$$(a+ib)+(c+id) = (a+c)+i(b+d), \quad (a+ib)(c+id) = (ac-bd)+i(ad+bc).$$

It is easy to check that $i^2 = (0+i)(0+i) = -1$ which justifies the interpretation of i as $\sqrt{-1}$.

The set of all complex numbers equipped with these operations is denoted by $\mathbf{C}$. One can easily verify that $\mathbf{C}$ is a *field*, i.e., the operations satisfy usual properties (like commutativity, associativity, distributivity, existence of 0 and 1, etc.)

If $z = a + ib \in \mathbf{C}$, then a is called the *real part* of z (notation: $a = \mathrm{Re}\ z$), while b is called *imaginary part* of z (notation: $b = \mathrm{Im}\ z$).

2. *Polar representation.* A complex number $z = a + ib$ has a naturel representation as a vector (a, b) in the real plane $\mathbf{R}^2$. The *norm* of this vector, $|z| = \sqrt{a^2 + b^2}$, is called the *modulus* of z, while the angle $\varphi = \arctan(b/a)$ between vector (a, b) and the positive direction of the horizontal axis is called the *argument* of z. Denote modulus $|z|$ by m. The pair (m, φ) completely determines z and is called *polar representation* of z. Note that in physics literature modulus is usually called *magnitude*, while argument is called *phase*. We pass from polar representation (m, φ) to the original representation $a + ib$ via formulas: $a = m\ \cos\varphi$, $b = m\ \sin\varphi$. A formula for multiplication of two complex numbers is particulary easy in the polar representation: $(m_1,\ \varphi_1)(m_2,\ \varphi_2) = (m_1 m_2,\ \varphi_1 + \varphi_2)$.

3. *Exponential representation.* From $a = m\ \cos\varphi$, $b = m\ \sin\varphi$ we deduce that $a+ib = m(\cos\varphi + i\sin\varphi)$. According to the Euler formula, $\cos\varphi + i\sin\varphi = e^{i\varphi}$,

where $e = 2.71828\ldots$ is a famous constant. This leads us to the *exponential representation*: $a + ib = me^{i\varphi}$. Multiplication formula in this case is $m_1 e^{i\varphi_1}\ m_2 e^{i\varphi_2} = m_1 m_2 e^{i(\varphi_1+\varphi_2)}$.

For a complex number $z = a + ib$ its *complex conjugate* is the number $\bar{z} = a - ib$. It is clear that if $z \in \mathbf{R}$, then $\bar{z} = z$.

2. Vectors and matrices

We consider an n-dimensional vector space $\mathbf{C}^n$ over complex numbers. Its elements are column vectors $\mathbf{z} = (z_1, \ldots, z_n)^T$, where every $z_i \in \mathbf{C}$ (here T is the symbol of the *transposition* operation). In physics (and quantum computing) vector $\mathbf{z}$ is also denoted by $|\mathbf{z}\rangle$. This is called *ket*-notation. There is also a sister *bra*-notation, $\langle \mathbf{z}|$, standing for the row vector $(\bar{z}_1, \ldots, \bar{z}_n)$. The bra-ket (bracket) notation was introduced by P. Dirac. The notation $\langle \mathbf{x}|\mathbf{y}\rangle$, for two vectors $\mathbf{x} = (x_1, \ldots x_n)$ and $\mathbf{y} = (y_1, \ldots y_n)$, means the *dot (inner) product* of these vectors, namely the number $\bar{x}_1 y_1 + \cdots + \bar{x}_n y_n$. The notation $|\mathbf{y}\rangle\langle \mathbf{x}|$ also has some meaning, discussed later.

As usual, linear maps between vector spaces are described by matrices, in our case having complex numbers as elements. We will need the following **special types of matrices**.
(a) Given a matrix A, its *adjoint matrix*, $A^\dagger$, is defined as $(\overline{A})^T = \overline{A^T}$, where the bar denotes element-wise complex conjugation.
(b) A matrix A is *Hermitian* if $A^\dagger = A$. Observe that A is a square matrix, and all its diagonal elements are real numbers.
(c) A square matrix U is *unitary* if $UU^\dagger = U^\dagger U = I$, where I is the *identity* matrix of the appropriate size. In other words, a matrix is unitary if its adjoint is its inverse.

Examples. The following matrix with real elements is unitary:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Another important example of a unitary matrix is the *Hadamard* matrix:

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Note that the *Hadamard* matrix is Hermitian.

Now we consider two types of matrix multiplications. The first type is the "usual" matrix multiplication of two matrices $A = \|a_{ij}\|$ and $B = \|b_{ij}\|$ with complex elements, applicable in case when the sizes of matrices "match". The latter means that A is $(m \times n)$-matrix while B is $(n \times k)$-matrix for some positive integers m, n, k. The product AB is an $(m \times k)$-matrix whose (i, j)-element is $\sum_{1 \le \nu \le n} a_{i\nu} b_{\nu j}$.

In quantum computing we need also another type of matrix multiplication, called *tensor (or Kronecker) product.*

Definition. *Let $A = \|a_{ij}\|$ and $B = \|b_{ij}\|$ be two matrices of sizes $m \times n$ and $p \times q$ respectively, with complex elements. Their tensor product, $A \otimes B$, is the $(mp \times nq)$-matrix:*

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \cdots & \cdots & \cdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}.$$

(As usual, $a_{ij}B$ means that each element of B is multiplied by a_{ij}.) Of course operation $\otimes$ is applicable to a particular case, that of vectors, i.e., if $\mathbf{x} = (x_1, \ldots, x_m)^T$ and $\mathbf{y} = (y_1, \ldots, y_n)^T$, then

$$\mathbf{x} \otimes \mathbf{y} = (x_1\mathbf{y}, \ldots, x_m\mathbf{y})^T.$$

Let us list the main properties of tensor product (proofs of these properties are straightforward). For matrices A, B, C, D, vectors $\mathbf{u}, \mathbf{x}, \mathbf{y}$ of appropriate dimensions, and numbers a, b the following hold:

$$(A \otimes B)(C \otimes D) = AC \otimes BD$$

$$(A \otimes B)(\mathbf{x} \otimes \mathbf{y}) = A\mathbf{x} \otimes B\mathbf{y}$$

$$(\mathbf{x} + \mathbf{y}) \otimes \mathbf{u} = \mathbf{x} \otimes \mathbf{u} + \mathbf{y} \otimes \mathbf{u}$$

$$\mathbf{u} \otimes (\mathbf{x} + \mathbf{y}) = \mathbf{u} \otimes \mathbf{x} + \mathbf{u} \otimes \mathbf{y}$$

$$a\mathbf{x} \otimes b\mathbf{y} = ab(\mathbf{x} \otimes \mathbf{y})$$

$$(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger$$

It is important to notice that, in general, the operation $\otimes$ is not *commutative*:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \neq \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

3. Qubits

A classical *bit* is one of the numbers 0 or 1.

Definition. *Quantum bit (qubit) is a unit vector in $\mathbf{C}^2$ for which a particular orthonormal basis is fixed. Basis is denoted by $|\mathbf{0}\rangle$, $|\mathbf{1}\rangle$, it might be, e.g., $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. A basis is fixed throughout the theory. Basis vectors are identified with classical bits, 0 and 1, respectively.*

Qubit, therefore, is $a|\mathbf{0}\rangle+b|\mathbf{1}\rangle$*, it might be a* $\begin{pmatrix}1\\0\end{pmatrix}+b\begin{pmatrix}0\\1\end{pmatrix}$*, for some* $a,b\in\mathbf{C}$ *such that* $|a|^2+|b|^2=1$*. When a qubit* $a|\mathbf{0}\rangle+b|\mathbf{1}\rangle$ *is* **measured (=observed)** *with respect to basis* $|\mathbf{0}\rangle,\ |\mathbf{1}\rangle$*, the qubit collapses either to* $|\mathbf{0}\rangle$ *(with probability* $|a|^2$ *or to* $|\mathbf{1}\rangle$ *(with probability* $|b|^2$*).*

We could have defined a qubit without the restriction $|a|^2+|b|^2=1$, but then the probabilities should be measured as $\frac{|a|^2}{|a|^2+|b|^2}$ and $\frac{|b|^2}{|a|^2+|b|^2}$, respectively. Qubits that differ by a constant multiplicative factor are considered as equivalent.

4. Bloch sphere

Consider a qubit $a|\mathbf{0}\rangle+b|\mathbf{1}\rangle$, where $a,b\in\mathbf{C}$ such that $|a|^2+|b|^2=1$. Each of complex numbers a and b is defined by two real numbers: $a=a_1+ia_2,\ b=b_1+ib_2$, where $a_1,a_2,b_1,b_2\in\mathbf{R}$. We now show that two reals are sufficient to specify a qubit.

First, as discussed above, represent a and b in polar form:

$$a=re^{i\varphi},\quad b=se^{i\psi}.$$

We get $re^{i\varphi}|\mathbf{0}\rangle+se^{i\psi}|\mathbf{1}\rangle$. Since we get an equivalent qubit by multiplying given qubit by a constant, we get

$$r|\mathbf{0}\rangle+se^{i(\psi-\varphi)}|\mathbf{1}\rangle.$$

The equation $|a|^2+|b|^2=1$ we can re-write as $r^2|e^{i\varphi}|^2+s^2|e^{i\psi}|^2=1$. Since $|e^{i\varphi}|^2=|e^{i\psi}|^2=1$, we get $r^2+s^2=1$. There exist $\alpha,\ 0\le\alpha\le\pi$ such that $r=\cos(\alpha/2),\ s=\sin(\alpha/2)$. Let $\beta=\psi-\varphi$. It follows that the qubit can be written as

$$\cos(\alpha/2)|\mathbf{0}\rangle+\sin(\alpha/2)e^{i\beta}|\mathbf{1}\rangle,$$

where $0\le\alpha\le\pi,\ 0\le\beta<2\pi$. This is called **canonical representation** of qubit. It follows that just two real numbers, α and β, completely represent the qubit. Canonical representation is unique except the case where the qubit is one of $|\mathbf{0}\rangle,\ |\mathbf{1}\rangle$.

Therefore, we can re-interpret $\alpha,\ \beta$ as polar coordinates on a sphere, and hence qubit can be interpreted as a point on a sphere: α is co-latitude with respect to z-axis, while β is longitude with respect to x-axis. This interpretation is known as the *Bloch sphere*.

Each point on Bloch sphere (i.e., each qubit) is expressed via $\alpha,\ \beta$ as follows:

$$x=\sin\alpha\cos\beta,\ y=\sin\alpha\sin\beta,\ z=\cos\alpha.$$

The following calculation shows that two antipodal points on Bloch sphere correspond to orthogonal qubits.

Consider a qubit in canonical representation:

$$|\mathbf{x}\rangle=\cos\left(\frac{\alpha}{2}\right)|\mathbf{0}\rangle+\sin\left(\frac{\alpha}{2}\right)e^{i\beta}|\mathbf{1}\rangle.$$

Its antipodal qubit on the Bloch sphere is

$$|\mathbf{y}\rangle = \cos\left(\frac{\pi-\alpha}{2}\right)|\mathbf{0}\rangle + \sin\left(\frac{\pi-\alpha}{2}\right)e^{i(\beta+\pi)}|\mathbf{1}\rangle =$$

$$= \cos\left(\frac{\pi-\alpha}{2}\right)|\mathbf{0}\rangle - \sin\left(\frac{\pi-\alpha}{2}\right)e^{i\beta}|\mathbf{1}\rangle.$$

So

$$\langle\mathbf{y}|\mathbf{x}\rangle = \cos\left(\frac{\alpha}{2}\right)\cos\left(\frac{\pi-\alpha}{2}\right) - \sin\left(\frac{\alpha}{2}\right)\sin\left(\frac{\pi-\alpha}{2}\right)$$

(here we use that, by definition, $\langle\mathbf{y}|\mathbf{x}\rangle$ means $\overline{\mathbf{y}}^T\mathbf{x}$, hence $e^{-i\beta}$ will be multiplied by $e^{i\beta}$ which gives 1). But $\cos(a+b) = \cos a\cos b - \sin a\sin b$, so $\langle\mathbf{y}|\mathbf{x}\rangle = \cos(\pi/2) = 0$. Hence antipodal points correspond to orthogonal qubits.

Chapter 8

Lecture Notes 8

1. Multiple qubits

Imagine a *macroscopic* physical object breaking apart and multiple pieces flying off in different directions. The state of this system can be described completely by describing the state of each of its component pieces separately. A surprising and nonintuitive aspect of the state space of an n-particle *quantum* system is that the state of the system cannot always be described in terms of the state of its component pieces.

A qubit can be represented by a vector in $\mathbf{C}^2$ spanned by a basis $\{|\mathbf{0}\rangle,\ |\mathbf{1}\rangle\}$. From the experience in classical physics, one would expect that a system of two qubits will be represented by a pair of vectors in $\mathbf{C}^2$, i.e., by a vector in $\mathbf{C}^4$ spanned by some basis $\{|\mathbf{0}\rangle,\ |\mathbf{1}\rangle,\ |\mathbf{0}'\rangle,\ |\mathbf{1}'\rangle\}$. However, the experiment shows that a system of two qubits should be represented by a vector in $\mathbf{C}^4$ spanned by the basis $\{|\mathbf{0}\rangle\otimes|\mathbf{0}\rangle, |\mathbf{0}\rangle\otimes|\mathbf{1}\rangle, |\mathbf{1}\rangle\otimes|\mathbf{0}\rangle, |\mathbf{1}\rangle\otimes|\mathbf{1}\rangle\}$. This is usually abbreviated to $\{|\mathbf{00}\rangle, |\mathbf{01}\rangle, |\mathbf{10}\rangle, |\mathbf{11}\rangle\}$. Hence, a representation of a qubit pair looks as follows: $a|\mathbf{00}\rangle + b|\mathbf{01}\rangle + c|\mathbf{10}\rangle + d|\mathbf{11}\rangle$, where $a, b, c, d \in \mathbf{C}$ and $|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$. Note that if we choose as $\{|\mathbf{0}\rangle,\ |\mathbf{1}\rangle\}$ the standard basis in $\mathbf{C}^2$, i.e., $\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}$, then performing tensor products shows that the basis $\{|\mathbf{00}\rangle, |\mathbf{01}\rangle, |\mathbf{10}\rangle, |\mathbf{11}\rangle\}$ in $\mathbf{C}^4$ can be written in a more familiar form:

$$\left\{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}\right\}.$$

Exercise. Write out the basis in $\mathbf{C}^4$ for a two-qubit system if bases in $\mathbf{C}^2$ for first and second qubits are chosen as follows:

(a) $\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}$ for the first qubit, $\left\{\begin{pmatrix}1/\sqrt{2}\\1/\sqrt{2}\end{pmatrix}, \begin{pmatrix}1/\sqrt{2}\\-1/\sqrt{2}\end{pmatrix}\right\}$ for the second qubit;

(b) $\left\{ \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}, \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} \right\}$ for the first as well as for the second qubit.

Check that in both cases, (a) and (b), bases in $\mathbf{C}^4$ are orthonormal, i.e., both orthogonal and normalised.

The difference between classical case and quantum case becomes even more dramatic when we consider systems of more than two qubits. For example, in the case of three qubits, the representing vector will belong not to $\mathbf{C}^6$, as one would have in classical case, but to the vector space $\mathbf{C}^8$ with the basis $\{|\mathbf{000}\rangle, |\mathbf{001}\rangle, |\mathbf{010}\rangle, |\mathbf{011}\rangle, |\mathbf{100}\rangle, |\mathbf{101}\rangle, |\mathbf{110}\rangle, |\mathbf{111}\rangle\}$. Here $|\mathbf{ijk}\rangle$ stands for $|\mathbf{i}\rangle \otimes |\mathbf{j}\rangle \otimes |\mathbf{k}\rangle$, where $\mathbf{i},\ \mathbf{j},\ \mathbf{k} \in \{\mathbf{0},\ \mathbf{1}\}$. The dimension of vector space grows exponentially with the number qubits.

We get a qubit pair if we take the tensor product of two qubits:

$$(a|\mathbf{0}\rangle + b|\mathbf{1}\rangle) \otimes (c|\mathbf{0}\rangle + d|\mathbf{1}\rangle) = ac|\mathbf{00}\rangle + ad|\mathbf{01}\rangle + bc|\mathbf{10}\rangle + bd|\mathbf{11}\rangle.$$

(Observe that conditions $|a|^2 + |b|^2 = 1$ and $|c|^2 + |d|^2 = 1$ imply that $|ac|^2 + |ad|^2 + |bc|^2 + |bd|^2 = 1$.) On the other hand, not every qubit pair can be represented as a tensor product, for example $\frac{1}{\sqrt{2}}|\mathbf{00}\rangle + 0|\mathbf{01}\rangle + 0|\mathbf{10}\rangle + \frac{1}{\sqrt{2}}|\mathbf{11}\rangle = \frac{1}{\sqrt{2}}|\mathbf{00}\rangle + \frac{1}{\sqrt{2}}|\mathbf{11}\rangle$. If it was possible to find $a, b, c, d \in \mathbf{C}$ such that

$$\frac{1}{\sqrt{2}}|\mathbf{00}\rangle + \frac{1}{\sqrt{2}}|\mathbf{11}\rangle = (a|\mathbf{0}\rangle + b|\mathbf{1}\rangle) \otimes (c|\mathbf{0}\rangle + d|\mathbf{1}\rangle) = ac|\mathbf{00}\rangle + ad|\mathbf{01}\rangle + bc|\mathbf{10}\rangle + bd|\mathbf{11}\rangle,$$

then we would have $ac = bd = \frac{1}{\sqrt{2}}$. Also $ad = 0$, hence either $a = 0$ or $d = 0$. But $a = 0$ implies that $ac = 0$, while $d = 0$ implies that $bd = 0$. Both alternatives lead to contradiction. We see that multiple qubits can't always be described in terms of it's components – individual qubits. This phenomenon is known as *entanglement*.

2. Measurement of multiple qubits

The process of measurement (observation) of a system of two qubits

$$a_{00}|\mathbf{00}\rangle + a_{01}|\mathbf{01}\rangle + a_{10}|\mathbf{10}\rangle + a_{11}|\mathbf{11}\rangle,$$

where $a_{ij} \in \mathbf{C}$ and $|a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$, results in collapsing of the system to a basis state $|\mathbf{ij}\rangle$ with probability $|a_{ij}|^2$. In other words, $|a_{ij}|^2$ is the probability that the first qubit in the system collapses to $|\mathbf{i}\rangle$ and the second qubit collapses to $|\mathbf{j}\rangle$.

On the other hand, one can measure *separately* the first and the second qubits of an (even entangled) two-qubit system.

Measuring the **first qubit**:
probability of collapsing to $|\mathbf{0}\rangle$ is $|a_{00}|^2 + |a_{01}|^2$;
probability of collapsing to $|\mathbf{1}\rangle$ is $|a_{10}|^2 + |a_{11}|^2$.

Measuring the **second qubit**:
probability of collapsing to $|\mathbf{0}\rangle$ is $|a_{00}|^2 + |a_{10}|^2$;

probability of collapsing to $|\mathbf{1}\rangle$ is $|a_{01}|^2 + |a_{11}|^2$.

Example. Using the standard basis in $\mathbf{C}^2$:

$$|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |\mathbf{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

consider a two-qubit system

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a \\ b \\ 0 \\ 0 \end{pmatrix},$$

where $|a|^2+|b|^2 = 1$. Since this system can be represented as a tensor product of single qubits (recall that it's may not possible in general), we can measure each qubit separately. It is clear that the first qubit $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ collapses to $|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ with probability 1, because it's the basis qubit in first place. Now let us measure this first qubit from the representation

$$\begin{pmatrix} a \\ b \\ 0 \\ 0 \end{pmatrix},$$

using the general rule.

probability of collapsing to $|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is $|a|^2 + |b|^2 = 1$;

probability of collapsing to $|\mathbf{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ is $0 + 0 = 0$,

which is consistent with our first method of measurement.

3. Gates

So far we have looked at static quantum systems which change only when measured (observed). But quantum systems may have dynamics, changing when not being measured. Passing from one state of the system to the next is the result of a linear transformation of the complex vector space $\mathbf{C}^n$. Linear transformations are realised by multiplying vectors in $\mathbf{C}^n$ by suitable matrices. In quantum physics and computation these are unitary (in particular – square) matrices. Any unitary transformation is a legitimate quantum transformation. Recall that every unitary transformation is reversible.

From the viewpoint of quantum computing, unitary transformations represent "gates".

For example, applying the unitary transformation

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

to the basis vector $|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ we get the basis vector $|\mathbf{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, and visa versa. This allows us to interpret X as the negation gate as far as classical bits, $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$, are concerned. As for general qubits, an easy calculation shows that applying X to the Bloch sphere amounts to rotation of the sphere about x-axis by π (180°).

Another example of a unitary transformation is given by the Hadamard matrix,

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Application H to each of the standard basis vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ gives another orthonormal basis for $\mathbf{C}^2$: $\left\{ \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}, \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} \right\}$.

Unitary transformations can be applied to multiple qubit states. Let us consider a gate which realises an instance of **if–then** statement. It should work as follows. If qubit $|\mathbf{x}\rangle$ is **true**, i.e., equal to $|\mathbf{1}\rangle$, then some unitary transformation $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is applied to qubit $|\mathbf{y}\rangle = \alpha|\mathbf{0}\rangle + \beta|\mathbf{1}\rangle$. If $|\mathbf{x}\rangle$ is **false**, i.e., equal to $|\mathbf{0}\rangle$, then $|\mathbf{y}\rangle$ remains unchanged. The input of this gate will be a two-qubit state $|\mathbf{xy}\rangle = |\mathbf{x}\rangle \otimes |\mathbf{y}\rangle$. Using the representation of the input in the standard basis in $\mathbf{C}^4$, write this two-qubit in the form:

$$|\mathbf{1y}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \alpha \\ \beta \end{pmatrix}$$

for $|\mathbf{x}\rangle = |\mathbf{1}\rangle$, and

$$|\mathbf{0y}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ 0 \\ 0 \end{pmatrix}$$

for $|\mathbf{x}\rangle = |\mathbf{0}\rangle$. As unitary transformation matrix, choose:

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{pmatrix}.$$

(Check that U is really unitary.)

Applying U to our two inputs we get:

$$U|\mathbf{1y}\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ a\alpha + b\beta \\ c\alpha + d\beta \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} a\alpha + b\beta \\ c\alpha + d\beta \end{pmatrix}$$

and

$$U|\mathbf{0y}\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

as required. The fact that in this instance we managed to decompose gate outputs into tensor factors is just a good luck. In general, even if an input of a gate is represented as a tensor product, the output may well turn out to be an entangled multi-qubit. However, in any case, measuring appropriate qubits in the entangled multi-qubit will provide the necessary elements of the output.

An important particular case of the gate with the matrix U, described above, is the **controlled-not** gate, where $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is the negation gate matrix X.

Sometimes the measurement operation is considered as a quantum "gate" though it's not unitary or, in general, reversible. This operation is usually performed at the end of a computation when we want to measure qubits and find bits.

We may hope to find unitary transformations corresponding to other logical connectives, or, more generally, other operations on qubit systems, so that, being assembled together in a net, these gates would represent an algorithm.

Chapter 9

Lecture Notes 9

1. Functions from bits to bits

There are exactly four distinct functions $f : \{0,1\} \to \{0,1\}$. Representing classical bits $0, 1$ in quantum fashion, as $|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $|\mathbf{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, we can define all functions by following linear transformations:

(1) $f(0) = 0,\ f(1) = 0$ is defined by $\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$;

(2) $f(0) = 0,\ f(1) = 1$ is defined by $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$;

(3) $f(0) = 1,\ f(1) = 0$ is defined by $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$;

(4) $f(0) = 1,\ f(1) = 1$ is defined by $\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$.

Note that the first and the last matrices are not unitary (not even invertible), so they can't represent quantum gates. However we can turn all four gates into unitary using the following "control" trick.

For every type of function f we can construct a unitary (4×4)-matrix U_f having the following property. Instead of a single bit, $|\mathbf{0}\rangle$ or $|\mathbf{1}\rangle$, we will take, as an input of U_f, a two-qubit $|\mathbf{x}\rangle \otimes |\mathbf{y}\rangle = |\mathbf{x}\ \mathbf{y}\rangle$, where $\mathbf{x},\ \mathbf{y} \in \{\mathbf{0},\ \mathbf{1}\}$. Here the first element of the input, $|\mathbf{x}\rangle$, is the bit one wants to evaluate by f, while the second, $|\mathbf{y}\rangle$, "controls" the output. More precisely, the output, $U_f\ |\mathbf{x}\ \mathbf{y}\rangle$, is of the form $|\mathbf{x}\rangle \otimes |\mathbf{y} \oplus f(\mathbf{x})\rangle = |\mathbf{x}\ (\mathbf{y} \oplus f(\mathbf{x}))\rangle$, where $\oplus$ is XOR, the exclusive-or Boolean operation on bits. Observe that in case $|\mathbf{y}\rangle = |\mathbf{0}\rangle$, the bit $|\mathbf{y} \oplus f(\mathbf{x})\rangle$ turns into $|f(\mathbf{x})\rangle$. It follows that $U_f\ |\mathbf{x}\ \mathbf{0}\rangle = |\mathbf{x}\ f(\mathbf{x})\rangle$. The value of $f(\mathbf{x})$ can now be recovered by measuring the second qubit of $|\mathbf{x}\ f(\mathbf{x})\rangle$.

We can write out explicitly the matrix U_f for each f:

$$(1)\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, (2)\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, (3)\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, (4)\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

It's easy to check that each of these matrices is unitary, hence can serve as a quantum gate. (In fact one can show that any matrix satisfying the same properties as U_f is its own inverse.)

What we have achieved in this section. *We showed that each function from $\{0,1\}$ to $\{0,1\}$ can be computed usung an appropriate quantum gate. In order to achieve reversibility of the gate (unitary property of the gate matrix) we added a control element to the input. The output now becomes a two-qubit, but the single-bit value of f can be read from output by measuring it.*

2. Deutsch's algorithm

We will discuss a simple quantum algorithm, called Deutsch's algorithm, that solves a slightly contrived problem.

Suppose that one of the functions (1)–(4) from the previous section, call it f, is presented to us as a *black box*, i.e., we can evaluate an input but we are not told which function it's actually is. Our aim is to decide whether or not the function is a bijection. (Note that only functions (2) and (3) are bijections.)

A classical approach would be to evaluate f separately at each bit. The function f is bijective if and only if these evaluations will produce different bits. Thus, the classical algorithm requires *two* evaluations of black box function. We will now describe a quantum Deutsch's algorithm that uses a *single* evaluation, i.e., a single application of the transformation with matrix U_f.

First let us visualise Deutsch's algorithm by means of a flow chart (see the separate page at the end). Here each box labelled by H denotes the gate realising the transformation with the Hadamard matrix $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$. This gate has a single qubit input and a single qubit output. The box labelled by U_f realises the transformation with the matrix U_f corresponding (see Section 1) to the black box function f. This gate has a two-qubit input and a two-qubit output. Finally, M is the "gate" denoting the measuring operation of the first qubit in a two-qubit.

In terms of matrices, the algorithm from the diagram first computes $H\begin{pmatrix}1\\0\end{pmatrix}\otimes H\begin{pmatrix}0\\1\end{pmatrix}$, which, by a tensor product formula from LN 7, equals to

$$(H\otimes H)\left(\begin{pmatrix}1\\0\end{pmatrix}\otimes\begin{pmatrix}0\\1\end{pmatrix}\right)=(H\otimes H)\begin{pmatrix}0\\1\\0\\0\end{pmatrix}.$$

Then, according to the diagram, the resulting two-qubit is put through the gate of the matrix U_f (this is a black box operation):

$$U_f(H\otimes H)\begin{pmatrix}0\\1\\0\\0\end{pmatrix}. \qquad (A)$$

Then we put the first qubit of the resulting two-qubit through the gate with the Hadamard matrix:

$$(H \otimes I)U_f(H \otimes H) \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \qquad (B)$$

where I is the (2×2)-identity matrix. It remains to measure the first qubit of the resulting two-qubit, which is a particular classical bit (recall that measuring a qubit always results in a classical bit). We now prove that if the output bit is $|\mathbf{0}\rangle$ (with probability 1), then f is not a bijection, otherwise, if it is $|\mathbf{1}\rangle$ (with probability 1), then f is a bijection.

Let us compute the value of the expression (B) step by step. The first step is:

$$(H \otimes H) \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1/2 \\ -1/2 \\ 1/2 \\ -1/2 \end{pmatrix}.$$

Now we compute the value of expression (A) separately for each matrix U_f. For matrix U_f of the type (1) the output coincides with the input

$$1/2 \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

since in this case $U_f = I$ is the identity matrix. For matrix U_f of the type (2), we get

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1/2 \\ -1/2 \\ 1/2 \\ -1/2 \end{pmatrix} = 1/2 \begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix}.$$

For matrix U_f of the type (3):

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1/2 \\ -1/2 \\ 1/2 \\ -1/2 \end{pmatrix} = 1/2 \begin{pmatrix} -1 \\ 1 \\ 1 \\ -1 \end{pmatrix}.$$

Finally, for the matrix U_f of the type (4):

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1/2 \\ -1/2 \\ 1/2 \\ -1/2 \end{pmatrix} = 1/2 \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \end{pmatrix}.$$

We notice that outputs on this stage for functions of types (1) and (4) differ only in sign. The same can be said about functions of types (2) and (3). Taking

this into account, we will perform the last matrix multiplication, by the matrix $H \otimes I$, explicitly only for functions of types (1) and (2), the outputs for (3) and (4) can be obtained from that multiplying by -1. First compute

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}.$$

Applying this transformation to (A) for the function f of the type (1), we get:

$$\left(\frac{1}{\sqrt{2}}\right)\left(\frac{1}{2}\right) \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \end{pmatrix}.$$

Doing the same for f of the type (2):

$$\left(\frac{1}{\sqrt{2}}\right)\left(\frac{1}{2}\right) \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 0 \\ 1 \\ -1 \end{pmatrix}.$$

According to the remark above, the outputs for functions of types (3) and (4) are

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 0 \\ -1 \\ 1 \end{pmatrix} \quad \text{and} \quad \frac{1}{\sqrt{2}} \begin{pmatrix} -1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

respectively.

The last step of the algorithm is to measure the first qubit in the obtained two-qubit. According to the general rule of measurement (see Section 2, LN 8), for f of types (1) and (4) the first qubit will collapse to $|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ with probability 1, and therefore to $|\mathbf{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ with probability 0. Similarly, for f of types (2) and (3) the first qubit will collapse to $|\mathbf{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ with probability 1, and therefore to $|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ with probability 0.* Now we recall that functions f of

* Note that since the two-qubit, that we obtain before measuring, is not entangled, it can be represented as a tensor product of two qubits. For example, for f of type (1) it will be:

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix}.$$

From this, measuring the first qubit is obvious.

types (2) and (3) are bijections, while types (1) and (2) are not. Thus, measuring of two-qubits solves the problem of deciding whether or not the black box function f is bijective. This was all accomplished with only one function evaluation as opposed to the two evaluations that the classical algorithm demands.

Chapter 10

Lecture Notes 10

1. Balancing problem

We now consider a generalization of the computational problem considered in Lecture Notes 9, which provides an example of a more spectacular quantum speedup.

Definition. *A function* $f : \{0,1\}^n \to \{0,1\}$ *is called* balanced *if exactly half of the inputs goes to* 0 *(and another half goes to* 1*). It is called* constant *if all values of* f *are* 0 *or all values are* 1.

Note that when $n = 1$, balanced functions coincide with bijective functions.

Computational problem. *Given a black-box function* $f : \{0,1\}^n \to \{0,1\}$*, which is either balanced or a constant, decide which of the two it is.*

A classical (non-quantum) algorithm starts evaluating f on different inputs. In (computationally) best case, the first and the second evaluations will produce different values, hence f is *not* constant, so it has to be balanced. But in the worst case we might need to continue evaluations until we evaluate more than the half of all possible inputs. Since the number of all $0-1$-vectors in $\{0,1\}^n$ is 2^n, we might need to perform $2^n/2 + 1 = 2^{n-1} + 1$ evaluations. However, using quantum computations, we can solve the problem with just *one* evaluation of f. This is an exponential speedup!

2. The Deutsch-Jozsa algorithm

The "flow chart" for the algorithm is essentially the same as the chart for the Deutsch algorithm (see Lecture Notes 9), with slightly different interpretation of its elements, as follows.

1. The top input 0 is now understood as $\mathbf{0} = (0, 0, \ldots 0)$, the n-tuple of classical 0-bits. Writing each of these bits in the quantum fashion, as $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, and taking

the tensor product, we represent the top input as

$$\begin{pmatrix}1\\0\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\0\end{pmatrix}=\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}.$$

2. Two top H-gates represent multiplications by the matrix

$$H^{\otimes n}:=\underbrace{H\otimes\cdots\otimes H}_{n\text{ times}}$$

3. Function f is represented by a unitary matrix U_f using the control element added to the input (see Section 1, Lecture Notes 9).

4. The measuring "gate" M measures an n-qubit.

In terms of matrices, the computation (except measuring) can be written as follows:

$$(H^{\otimes n}\otimes I)U_f(H^{\otimes n}\otimes H)\left(\begin{pmatrix}1\\0\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\0\end{pmatrix}\otimes\begin{pmatrix}0\\1\end{pmatrix}\right).$$

One can calculate how elements of the matrix $H^{\otimes n}$ look like. Consider first the case $n=2$. Label rows and columns in $H^{\otimes 2}=H\otimes H$, in a natural way, by 2-words in alphabet $\{0,1\}$:

$$H^{\otimes 2}=\frac{1}{2}\begin{pmatrix}&00&01&10&11\\00&1&1&1&1\\01&1&-1&1&-1\\10&1&1&-1&-1\\11&1&-1&-1&1\end{pmatrix}.$$

Then it is easy to check that the element $H^{\otimes 2}[(i_1,j_1),(i_2,j_2)]$ of this matrix is equal to $\frac{1}{2}(-1)^{(i_1\wedge i_2)\oplus(j_1\wedge j_2)}$, where $\wedge$ is the logical conjunction and $\oplus$ is XOR. For example, $(0\wedge 0)\oplus(1\wedge 1)=0\oplus 1=1$, hence $H^{\otimes 2}[(0,1),(0,1)]=\frac{1}{2}(-1)$. In general, label rows and columns in $H^{\otimes n}$ by n-words in alphabet $\{0,1\}$. Let $\mathbf{i}=(i_1,\ldots,i_n)$ be a label of a row, while $\mathbf{j}=(j_1,\ldots,j_n)$ be a label of a column. Then

$$H^{\otimes n}[\mathbf{i},\mathbf{j}]=\frac{1}{\sqrt{2^n}}(-1)^{\langle\mathbf{i},\mathbf{j}\rangle},$$

where $\langle\mathbf{i},\mathbf{j}\rangle=(i_1\wedge j_1)\oplus(i_2\wedge j_2)\oplus\cdots\oplus(i_n\wedge j_n)$. *

* If you know what the 2-element field $\mathbf{Z}_2$ is, observe that $\langle\mathbf{i},\mathbf{j}\rangle$ is the dot product in the vector space $(\mathbf{Z}_2)^n$.

Straightforward calculations show that if f is the constant function $f \equiv 1$, then the top n-qubit before the measuring gate is

$$(-1)\begin{pmatrix}1\\0\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\0\end{pmatrix} = (-1)\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}.$$

If f is the constant function $f \equiv 0$, then the top n-qubit before the measuring gate is

$$\begin{pmatrix}1\\0\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}.$$

In both cases, after measuring the top n-qubit collapses to the vector of classical bits

$$\begin{pmatrix}1\\0\end{pmatrix},\cdots,\begin{pmatrix}1\\0\end{pmatrix}$$

with probability 1. If, after measuring, anything else is found, then f is balanced.

Bibliography

[BFH+21] Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *SAT Competition 2021: Solver and Benchmark Descriptions*, number B-2021-1. University of Helsinki Department of Computer Science, 2021.

[CO14] Amin Coja-Oghlan. The asymptotic k-sat threshold. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 804–813, 2014.

[Coo66] S.A. Cook. *On the minimum computation time of functions*. PhD thesis, Department of Mathematics Harvard University, 1966.

[Dav92] J.H. Davenport. Primality Testing Revisited. In P.S. Wang, editor, *Proceedings ISSAC 1992*, pages 123–129, 1992.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[FKRS17] T. Friedrich, A. Krohmer, R. Rothenberger, and A.M. Sutton. Phase Transitions for Scale-Free SAT Formulas. In *Proceedings AAAI 2017*, pages 3893–3899, 2017.

[Heu18] M.J.H. Heule. Schur number five. *AAAI'18/IAAI'18/EAAI'18: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference. and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence Article No.: 808*, pages 6598–6606, 2018.

[KS00] W. Küchlin and C. Sinz. Proving Consistency Assertions for Automotive Product Data Management. *J. Automated Reasoning*, 24:145–163, 2000.

[KSM11] M. Kaufmann and J. Strother Moore. How Can I Do That with ACL2? Recent Enhancements to ACL2. `http://arxiv.org/abs/1110.4673`, 2011.

[LSZ93] M. Luby, A. Sinclair, and D. Zuckerman. Optimal Speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings 38th Design Automation Conference*, pages 530–535, 2001.

[MSS96] J.P. Marques-Silva and K.A Sakallah. GRASP — A New Search Algorithm for Satisfiability. In *Proceedings ICCAD*, pages 220–227, 1996.

[Rab80] M.O. Rabin. Probabilistic Algorithm for Testing Primality. *J. Number Theory*, 12:128–138, 1980.

[Tse68] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic Part 2 (ed A. Slisenko)*, pages 115–125, 1968.

Index