TU Dresden, Fakultät Informatik
Daniel Borchmann

Summer Term 2016

Advanced Topics in Complexity Theory

Exercise 1: General Revision

2016-04-05

This exercise sheet is meant as a general revision of the fundamental notions of complexity theory. We start with the definition of a deterministic Turing machine.

Definition 1.1 A deterministic *Turing machine* $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ consists of

- a finite set Q of *states*,
- an *input alphabet* Σ not containing $\square$,
- a *tape alphabet* Γ such that $\Gamma \supseteq \Sigma \cup \{ \square \}$.
- a *transition function* $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{ \mathsf{L}, \mathsf{R} \}$
- an *initial state* $q_0 \in Q$,
- an *accepting state* $q_{\text{accept}} \in Q$, and
- an *rejecting state* $q_{\text{reject}} \in Q$ such that $q_{\text{accept}} \neq q_{\text{reject}}$. ♢

The current *configuration* of a Turing machine M is completely determined by the current state $q \in Q$, the current word w of the tape, and the position of the head. If $w = w_1 w_2$ is such that in the current configuration of M the head is placed right at the first letter of w_2, then we can denote the current configuration by $w_1 q w_2$. A configuration is called *accepting* of $q = q_{\mathsf{accept}}$ and called *rejecting* if $q = q_{\text{reject}}$. For some word $w \in \Sigma$ the *initial configuration* C_w of M on input w is the configuration $q_0 w$.

A configuration C_i *goes into* a configuration C_{i+1} of M, written $C_i \vdash_M C_{i+1}$, if the machine M can go from configuration C_i to configuration C_{i+1} by a step allowed by the transition function of M. A *computation* of M on input w is a sequence $C_0, \ldots, C_n$ of configurations satisfying

$$C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_{n-1} \vdash_M C_n$$

such that $C_0 = C_w$ and C_n is either an accepting or a rejecting configuration. The computation is called *accepting* if C_n is accepting, and rejecting otherwise. The length of such a computation is n.

The *language* accepted by a Turing machine M is the set of all words $w \in \Sigma^*$ such that the computation of M on input w is accepting.

Exercise 1.2 Show that there exist languages L that are not accepted by any Turing machine.

The definition of Turing machines can be adapted in many ways. Two important examples are *multi-tape* Turing machines and *non-deterministic* Turing machines. Multi-tape Turing machines have more than one tape to work on. Non-deterministic Turing machines have instead of a transition function δ a *transition relation* that specifies which transitions of the machine are valid. In general, a non-deterministic Turing machine can have more than one valid computation on a given input word. In contrast to deterministic Turing machines, a *rejecting* configuration of a non-deterministic Turing machine is a configuration that is either in state q_{reject} or for which no successor configuration exists. The language accepted by a non-deterministic Turing machine M is the set of all words w such that there exists an accepting computation of M on input w.

Let $f\colon \mathbb{N} \to \mathbb{N}$. A Turing machine M is called $f(n)$*-time bounded* if for each word $w \in \Sigma^*$ of length n the length of each computation of M on input w is bounded by $\mathcal{O}(f(n))$. A Turing machine M is called $f(n)$*-space bounded* if it halts and every input, has a designated read-only input tape, and for each word $w \in \Sigma^*$ of length n each tape in each configuration of each computation of M on w that is not the input tape never contains more than $\mathcal{O}(f(n))$ letters.

Exercise 1.3 Show that every language that can be accepted by an $f(n)$-space bounded deterministic Turing machine can also be accepted by an $2^{\mathcal{O}(f(n))}$-time bounded deterministic Turing machine.

The notion of time and space bounded Turing machines allows us to define our first complexity classes.

Definition 1.4 Let $f\colon \mathbb{N} \to \mathbb{N}$ and $k \in \mathbb{N}_+$. Then

- $\mathsf{DTime}_k(f(n))$ denotes the class of all languages accepted by an $f(n)$-time bounded k-tape deterministic Turing machine;
- $\mathsf{NTime}_k(f(n))$ denotes the class of all languages accepted by an $f(n)$-time bounded k-tape non-deterministic Turing machine;
- $\mathsf{DSpace}_k(f(n))$ denotes the class of all languages accepted by an $f(n)$-space bounded k-tape deterministic Turing machine;
- $\mathsf{NSpace}_k(f(n))$ denotes the class of all languages accepted by an $f(n)$-space bounded k-tape non-deterministic Turing machine.

$$\mathsf{DTime}(f(n)) \coloneqq \bigcup_{k\in\mathbb{N}_+} \mathsf{DTime}_k(f(n)), \qquad \mathsf{NTime}(f(n)) \coloneqq \bigcup_{k\in\mathbb{N}_+} \mathsf{NTime}_k(f(n)),$$
$$\mathsf{DSpace}(f(n)) \coloneqq \bigcup_{k\in\mathbb{N}_+} \mathsf{DSpace}_k(f(n)), \qquad \mathsf{NSpace}(f(n)) \coloneqq \bigcup_{k\in\mathbb{N}_+} \mathsf{NSpace}_k(f(n)) \quad \diamondsuit$$

Based on this definition, we are now able to introduce some classical complexity classes.

$$\mathsf{L} = \mathsf{DSpace}(\log n), \qquad \mathsf{NL} = \mathsf{NSpace}(\log n),$$
$$\mathsf{P} = \bigcup_{d\geq 1} \mathsf{DTime}(n^d), \qquad \mathsf{PSpace} = \bigcup_{d\geq 1} \mathsf{DSpace}(n^d),$$
$$\mathsf{NP} = \bigcup_{d\geq 1} \mathsf{NTime}(n^d), \qquad \mathsf{ExpTime} = \bigcup_{d\geq 1} \mathsf{DTime}(2^{n^d}).$$

Of these, the class NP will receive the most attention during the course of this lecture. Therefore, different characterizations of NP will play an important role.

Exercise 1.5 Show that $L \in \mathsf{NP}$ if and only if there exists a *deterministic polynomial-time verifier* for L, i.e., there exists a polynomial $p(n)$ and polynomial-time bounded deterministic Turing machine M such that

$$L = \{\, x \in \Sigma^* \mid \exists y \in \Sigma^* \colon |y| \leq p(|x|) \wedge (x, y) \in \mathcal{L}(M) \,\}.$$

It is unknown whether $\mathsf{P} = \mathsf{NP}$, and thus we do not know if each problem in NP can be solved in polynomial time. To still certify that certain problems in NP are "more difficult" than others we use the notion of *polynomial-time reductions*.

Definition 1.6 Let $L_1, L_2 \subseteq \Sigma^*$ and let $f \colon \Sigma^* \to \Sigma^*$. We say that f is a *reduction* from L_1 to L_2 if and only if for all $w \in \Sigma^*$ we have

$$w \in L_1 \iff f(w) \in L_2.$$

We say that f can be *computed in polynomial time* if there exists a deterministic polynomial-time bounded Turing machine with a designated output tape, such that on input w accepts with $f(w)$ written on this output tape.

Finally f is a *polynomial-time reduction* (*Karp-reduction*) from L_1 to L_2 if and only if f is a reduction from L_1 to L_2 that can be computed in polynomial time. We write $L_1 \leq_\mathsf{p} L_2$ if there exists a polynomial-time reduction from L_1 to L_2. ♢

Intuitively, $L_1 \leq_\mathsf{p} L_2$ means that solving L_1 cannot be "much more difficult" than solving L_2. Thus, a problem L such that $L' \leq_\mathsf{p} L$ for all $L' \in \mathsf{NP}$ is at least as difficult as every problem in NP. We call such problems L NP-*hard*. NP-hard problems that are contained in NP are called NP-*complete*. Therefore, NP-complete problems can be regarded as the most difficult problems in NP. Surprisingly, those problems exist, and showing this is not even difficult!

Exercise 1.7 Show that the word-problem of nondeterministic polynomial-time Turing machines is NP-complete.

Another prominent problem known to be NP-complete is SAT, the satisfiability of propositional formulas in conjunctive normal form.

Theorem 1.8 (Cook-Levin) SAT is NP-complete.

This problem remains NP-complete if we restrict our attention to propositional formulas in conjunctive normal form where each clause has at most three literals. This problem is called 3SAT.

Exercise 1.9 Show that 3SAT is NP-complete.

Note that the corresponding decision problem 2SAT where each clause contains at most 2 literals is in P.

Exercise 1.10 Show that the following problem is NP-complete:

Input:	A propositional formula φ in CNF
Question:	Does φ have at least 2 different satisfying assignments?

Exercise 1.11 (Small Challenge) Show that IndependentSet is NP-complete:

Input:	An undirected graph $G = (V, E)$ and some $k \in \mathbb{N}$.
Question:	Does there exists an *independent set* in G of size k or more?

(Recall that an independent set in a graph G is a set W of vertices such that no two vertices in W are connected via an edge in G.)

Based on this result, show that the problems of finding large cliques (i.e., complete subgraphs) and of finding small vertex covers (i.e., sets of vertices such that every edge is adjacent to at least one of these) in undirected graphs are also NP-complete.