CS301 Complexity of Algorithms Notes

Based on lectures by Dr. Matthias Englert
Notes by Alex J. Best

May 15, 2014

Introduction These are some rough notes put together for CS301 in 2014 to make it a little easier to revise. The headings correspond roughly to the contents of the module that is on the module webpage so hopefully these are fairly complete, however they are not guaranteed to be.

What is a problem?

Definition 1. A *problem* is a function

$$f\colon \{0,1\}^* \to \{0,1\}^*.$$

A *decision problem* is a function

$$f\colon \{0,1\}^* \to \{0,1\}.$$

We identify a decision problem f with the language

$$L_f = \{x : f(x) = 1\}.$$

and call the problem of computing f the problem of deciding the language L_f.

What is a computation? A set of fixed mechanical rules for computing a function for any input.

Definition of a Turing machine.

Definition 2. A *Turing machine* consists of an infinite tape with letters of the alphabet Γ (usually $= \{0, 1, \square\}$) written on it. At the start the input is written on the tape beginning at the head and the state is q_{start}. The transition function

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$$

dictates what to read, which state to switch to and which direction to move to after reading a symbol while in a given state. So the machine is given by (Γ, Q, δ). When the machine reaches the state q_{end} the computation halts and the output is the section of tape starting at the head until the first blank $\square$.

Definition 3. A Turing machine M computes a function $f\colon \{0,1\}^* \to \{0,1\}^*$ in time $T\colon \mathbb{N} \to \mathbb{N}$ if for every $x \in \{0,1\}^*$ the output of M when given x initially is $f(x)$, this computation must finish after at most $T(|x|)$ steps (this implies M halts on every input).

We say M computes f if it computes f in $T(n)$ time for any function T.

What happens if we increase the alphabet?

Claim 1. Let $f\colon \{0,1\}^* \to \{0,1\}^*$ and $T\colon \mathbb{N} \to \mathbb{N}$ be some functions. If f is computable in time $T(n)$ by a Turing machine M using alphabet Γ then f is computable in time

$$3\lceil \log |\Gamma| \rceil T(n)$$

by a Turing machine $\tilde{M}$ that uses only the alphabet $\{0,1,\square\}$.

To see this we can encode all the old symbols in terms of only $\{0,1,\square\}$ and create a new Turing machine that does what the old one would do, we have to move back an fourth along a strip of size $\log |\Gamma|$ at most 3 times to do this however.

k-tape Turing machines. We can also using Turing machines that have k tapes and k heads rather than simply one however this doesn't make much difference either.

Claim 2. Let $f\colon \{0,1\}^* \to \{0,1\}^*$ and $T\colon \mathbb{N} \to \mathbb{N}$ be some functions. If f is computable in time $T(n)$ by a k-tape Turing machine M using then f is computable in time

$$7kT(n)^2$$

by a single tape Turing machine $\tilde{M}$.

Church-Turing thesis. The Church-Turing thesis is the statement that

> Every physically realisable computation device (be it silicon-based, DNA-based, neuron based, ...) can be simulated by a Turing machine.

This is generally believed to be true.

Universal Turing machines. As a Turing machine is given by some finite amount of data we can represent it by some string encoding. We assume that we have picked an encoding so that every string represents a Turing machine and every Turing machine can be encoded by infinitely many strings. The Turing machine encoded by a string α is denoted M_α.

Theorem 1. There exists a Turing machine (a universal Turing machine) that when given a pair $\langle \alpha, x \rangle$ as input outputs the result of running the Turing machine encoded by α with input x. Moreover if the machine encoded by α halts within $T(|x|)$ steps on input x then the universal Turing machine halts within $C\,\mathrm{poly}(T(|x|))$ steps, where C is independent of $|x|$.

We assume we have fixed some encoding for Turing machines and a corresponding universal Turing machine, denoted $\mathcal{U}$.

Uncomputable functions. The set of all functions

$$f\colon \{0,1\}^* \to \{0,1\}$$

is uncountable. So as any Turing machine can be encoded as a finite string of bits under some fixed encoding, there are countably many Turing machines.

Corollary 1. So there are functions

$$f\colon \{0,1\}^* \to \{0,1\}$$

that are not computable.

Example 1. The function

$$f(\alpha) = \begin{cases} 0 & M_\alpha(\alpha) = 1, \\ 1 & \text{otherwise} \end{cases}$$

is not computable.

This is as if we had a Turing machine M that computed f then M halts for all x, with $M(x) = f(x)$. So if we let x be a string representing the Turing machine M then $M(x) = f(x)$, but this is a contradiction by the definition of f.

HALT is not computable. Another function that is not computable is

$$\text{HALT}(\langle \alpha, x \rangle) = \begin{cases} 1 & M_\alpha \text{ halts on input } x, \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Assume that M_{HALT} is a Turing machine that computes HALT, then we can design a Turing machine to compute the function f in example 1, thus deriving a contradiction.

The Turing machine for f would first run $M_{\text{HALT}}(\langle \alpha, \alpha \rangle)$ outputting 0 if this returned 1. Otherwise it would then use the universal Turing machine $\mathcal{U}$ to compute $M_\alpha(\alpha)$, outputting 1 if this returned 0 and 0 otherwise. □

There are more functions that are practically useful that are not computable. For example deciding if a Diophantine equation (a possibly multivariate polynomial with integer coefficients) has a solution in the integers is impossible in general.

The language

$$\{\alpha : M_\alpha \text{ halts on all inputs}\}$$

is also undecidable. This is as if we could compute this language we could compute HALT by taking a pair $\langle \alpha, x \rangle$ and forming a Turing machine the always computes $M_\alpha(x)$ no matter what input it is given. If we could decide if the new Turing machine halted on all inputs we could decide if M_α halts on input x.

Rice's Theorem. All Turing machines correspond to a function

$$f \colon \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$$

where $f(x) = \perp$ means that the Turing machine does not halt on input x. Not all of these functions correspond to Turing machines, but we can take $\mathcal{R}$ to be the set of all such functions that do correspond to Turing machines.

Theorem 2 (Rice's Theorem)**.** Let $\mathcal{C}$ be a non-empty proper subset of $\mathcal{R}$, then the language

$$\{\alpha : M_\alpha \text{ corresponds to a function } f \in \mathcal{C}\}$$

is undecidable.

Proof. □

The complexity class P. We now define some *complexity classes*, sets of functions that can be computed with some given resources.

Definition 4 (The class DTIME)**.** Let $T\colon \mathbb{N} \to \mathbb{N}$ be a function, then we let $\mathrm{DTIME}(T(n))$ be the set of boolean functions computable in $O(T(n))$ time.

Definition 5 (The class P)**.**

$$\mathrm{P} = \bigcup_{k \geq 1} \mathrm{DTIME}(n^k).$$

This class does not depend on the exact definition of Turing machine used. Problems in P are thought of as efficiently solvable and are a very natural model for this concept.

Strong Church-Turing thesis. The strong Church-Turing thesis is the statement that

> Every physically realisable computation device (be it silicon-based, DNA-based, neuron based, ...) can be simulated by a Turing machine *with only a polynomial overhead.*

This is more controversial than the normal Church-Turing thesis as either

a) quantum mechanics does not behave as we currently understand it to,

b) a classical computer can factor integer's in polynomial time or

c) the strong Church-Turing thesis is wrong.

Reductions.

Definition 6 (Karp reduction)**.** A language $L \subseteq \{0,1\}^*$ is *Karp reducible* to a language $L' \subseteq \{0,1\}^*$ if there exists a computable function

$$f\colon \{0,1\}^* \to \{0,1\}^*,$$

such that for all x

$$x \in L \iff f(x) \in L'.$$

Definition 7 (Polynomial time Karp reduction)**.** A language $L \subseteq \{0,1\}^*$ is *polynomial time* Karp reducible to a language $L' \subseteq \{0,1\}^*$ if there exists a *polynomial time* computable function

$$f\colon \{0,1\}^* \to \{0,1\}^*,$$

such that for all x

$$x \in L \iff f(x) \in L'.$$

We denote this relationship by

$$L \leq_p L'.$$

If $L \leq_p L'$ and $L' \leq_p L$ then we write $L \equiv_p L'$.

Vertex Cover and Independent Set are equivalent. INDEPENDENT SET: Given a graph $G = (V, E)$ and an integer k is there set $S \subset V$ of size at least k where each edge of G has at most one endpoint in S.
VERTEX COVER: Given a graph $G = (V, E)$ and an integer k is there set $S \subset V$ of size at most k where each edge of G has at least one endpoint in S.

Claim 3. VERTEX COVER $\equiv_p$ INDEPENDENT SET.

Proof. S is an independent set if and only if $V \setminus S$ is a vertex cover. □

Vertex Cover reduces to Set Cover. SET COVER: Given a set U, a collection $S_1, \ldots, S_m$ of subsets of U and an integer k, does there exist a collection of $\leq k$ of the subsets whose union is all of U.

Claim 4. VERTEX COVER $\leq_p$ SET COVER.

Proof. We create a set cover instance for a graph $G = (E, V)$ by letting $U = E$ and $S_v = \{e \in E : e \text{ is incident to } v\}$ be our collection of subsets. Then there exists a set cover of size at most k if and only if there exists a vertex cover of size at most k in the graph G. □

3-SAT reduces to Independent Set.

Definition 8. A *literal* is a boolean variable or its negation.
A *clause* is a disjunction (OR) of literals.
A propositional formula Φ is in *conjunctive normal form* if it is a conjunction (AND) of clauses.
The problem SAT asks if a given propositional formula Φ that is in CNF has a satisfying assignment of variables.
A special case of this is 3-SAT where we require that each clause be a disjunction of exactly three literals.

Claim 5. 3-SAT $\leq_p$ INDEPENDENT SET.

Proof. We use the given proposition formula to construct an instance of INDEPENDENT SET as follows. For each clause we add a triangle of vertices to a graph G, labelled by each literal. We then connect all literals appearing to all of their negations. Then G contains an independent set of size k if and only if Φ is satisfiable. □

Transitivity for the reducibility-relation. Reduction is transitive, i.e. if $X \leq_p Y$ and $Y \leq_p Z$ then $X \leq_p Z$. To see this we can think of composing the functions that provide the reductions from X to Y and Y to Z.

Example 2.

$$\text{3-SAT} \leq_p \text{INDEPENDENT-SET} \leq_p \text{VERTEX-COVER} \leq_p \text{SET-COVER}$$

Definition of NP. For comparison we give an equivalent definition of the class P to the one given above.

Definition 9 (Complexity class P)**.** A language L is in the class P if there exists a Turing machine M and polynomial T so that M terminates on input x after at most $T(|x|)$ steps and M accepts x if and only if $x \in L$.

Definition 10 (Complexity class NP)**.** A language L is in the class NP if there exists a Turing machine M and polynomials T and p so that M terminates on any input x after at most $T(|x|)$ steps. We also require that if $x \in L$ then there exists a certificate $t \in \{0,1\}^{p(|x|)}$ so that M accepts $\langle x, t\rangle$, conversely if $x \notin L$ we require that M rejects any pair $\langle x, t\rangle$ where $t \in \{0,1\}^{p(|x|)}$.

In the definition of NP above M is called a certifier or verifier and t a certificate or proof for x.

Claim 6. P $\subseteq$ NP

Proof. Take the certifier M to be the decider for the problem from P, ignoring the second element of the input pair. □

Example 3. The language

$$\text{COMPOSITES} = \{s \in \mathbb{N} : s \text{ is composite}\}$$

is in NP. This is as we can use a non-trivial proper factor as a certificate, and then the certifier just needs to check the factor is as claimed by checking bounds and dividing. Here $|t| \leq |s|$.

Example 4. The problem SAT is in NP as we can take a satisfying assignment of the n boolean variables in the formula as the certificate. The verifier then needs only run through the formula checking that each clause has at least one true literal.

Example 5. Another problem in NP is HAM-CYCLE which asks if a given graph $G = (V, E)$ has a simple cycle that visits every node. The certificate for this problem is a permutation of the nodes of G and so the certifier need only check that all nodes are in this permutation exactly once and that there are edges between consecutive nodes in the permutation.

Definition of NP-completeness.

Definition 11 (NP-completeness)**.** A (decision) problem Y is called *NP-complete* if it is in NP and has the property that every other problem X in NP reduces to it in polynomial time.

Theorem 3. Let Y be an NP-complete problem, then Y is solvable in polynomial time if and only if P = NP.

Proof. ($\Leftarrow$) If P = NP then Y can be solved in polynomial time as Y is in NP.
($\Rightarrow$) If Y can be solved in polynomial time then as any problem X in NP reduces to Y in polynomial time we can solve X is polynomial time. Hence NP $\subseteq$ P and so P = NP. □

An easy artificial NP-complete problem: TMSAT.

Alternative definition of NP using reductions and a representative (i.e. complete) problem of the class. Given a natural NP-complete problem we can equivalently define the complexity class NP to be all problems polynomial time reducible to that problem.

Cook-Levin theorem.

Theorem 4 (Cook, Levin)**.** SAT is NP-complete.

Proof. We know SAT is in NP, in order to show that all other problems in NP reduce to it we need some more machinery! □

Oblivious Turing machines.

Definition 12. A Turing machine is called *oblivious* is its head movements do not depend on the input x but only on the length $|x|$ of the input.

Theorem 5. Given any Turing machine M that decides a language in time $T_M(n)$ there exists an oblivious Turing machine that decides the same language in time $O(T_m(n)^2)$.

Proof. Exercise! □

Proof of the Cook-Levin theorem. We will solve the problem of certificate ex

Now we have established a natural NP-complete problem others are easier to do. To establish the NP-completeness of a given problem Y we can perform the following

Step 1 Show Y is in NP.

Step 2 Choose an appropriate NP-complete problem X.

Step 3 Prove $X \leq_p Y$.

A major problem of complexity theory is if P = NP, if it does then there are efficient algorithms for all NP-complete problems. If not no such algorithms are possible, and most computer scientists believe this to be the case. Most NP problems are known to be in P or NP-complete, factoring and graph isomorphism are exceptions.

3-SAT is NP-complete.

Theorem 6. 3-SAT is NP-complete.

Proof. It suffices to show that SAT $\leq_p$ 3-SAT since we know that 3-SAT is in NP.Let Φ be any SAT-formula and let $C = (L_1 \vee L_2 \vee \cdots \vee L_k)$ be a clause of size $k > 3$ ($L_i = x_i$ or $L_i = \overline{x_i}$). Introduce a new variable z and replace C with

$$C' = (L_1 \vee L_2 \vee \cdots \vee L_{k-2} \vee z) \text{ and } C'' = (L_{k-1} \vee L_k \vee \overline{z}).$$

Doing this will transform all clauses into clauses of at most 3 literals. Finally we turn the clauses of length < 3 into ones of length 3. □

Subset-Sum is NP-complete. SUBSET-SUM: Given natural numbers $w_1, \ldots, w_n$ and an integer W, is there a subset that adds up exactly to W?

Example 6.

$$\{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\},\ W = 3654.$$

is a yes instance

$$1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = 3754.$$

Remark 1. With arithmetic problems, input integers are encoded in binary. Polynomial reduction must be polynomial in *binary* encoding.

Claim 7. 3-SAT $\leq_p$ SUBSET-SUM.

Proof. Given an instance Φ of 3-SAT, we construct an instance of SUBSET-SUM that has a solution iff Φ is satisfiable.

Given a 3-SAT instance Φ with n variables and k clauses, form $2n + 2k$ decimal integers, each of $n + k$ digits. □

Scheduling With Release Times is NP-complete.

Hamiltonian cycle is NP-complete.

TSP is NP-complete.

3-Colouring is NP-complete.

Planar 3-Coloring is NP-complete.

Planar k-Coloring.

Oracle Turing Machines.

Cook-reductions.

Oracle Turing Machines.

Cook-reductions.

Self-Reducibility.

Gödel's first incompleteness Theorem.

Test your intuition - complexity of (Longest Path, Shortest Path, Perfect Matching, MaxCut, Halt within 4000 steps, Min spanning tree, Degree bounded min spanning tree).

Where does TAUTOLOGY fit?

Asymmetry of NP.

NP versus coNP.

Some properties of NP and coNP.

Well characterized problems.

PRIMES is in NP.

FACTOR is well characterized.

PSPACE.

QSAT is in PSPACE.

NP is a subset of PSPACE.

PSPACE is a subset of EXPTIME.

Competitive Facility Location is PSPACE-complete.

Sliding Blocks.

Randomization.

RP.

Probability Amplification.

coRP.

Polynomial equality.

Polynomial identity testing.

BPP.

Probability Amplification for BPP.

ZPP.

ZPP=RP $\cap$ coRP.

ZPP as expected poly-time.

IP.

IP protocol for Graph-Non-Isomorphism.

dIP=NP.

coNP is contained in IP, arithmetization, and the sumcheck protocol.

IP.

Program checking.

Zero knowledge proofs.

Zero knowledge proof protocol for graph isomorphism.

Communication Complexity.

Computing OR.

Computing MEDIAN.

Communication Complexity.

Computing MEDIAN.

Protocol Trees.

EQUALITY.

Deciding Palindromes requires quadratic time.

Communication Complexity.

Protocol trees and combinatorial rectangles.

EQUALITY (again).

DISJOINTNESS.

INNER PRODUCT and the rank technique.

Area-Time tradeoffs for VLSI chips.

NP-hard.

Approximation Algorithms.

2-approximation for Max Sat.

2-approximation for Load Balancing (List Scheduling).

3/2-approximation for Load Balancing (LPT).

$O(\log n)$**-approximation for Set Cover.**

2-approximation for Vertex Cover.

PTAS and FPTAS.

An FPTAS for Knapsack.

Linear Programming based approximation algorithm for Weighted Set Cover.