Exam study sheet for CS6902, winter 2015

Turing machines and decidability.

- A Turing machine is a finite automaton with an infinite memory (tape). Formally, a Turing machine is a 6-tuple \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}) \). Here, \( Q \) is a finite set of states as before, with three special states \( q_0 \) (start state), \( q_{\text{accept}} \) and \( q_{\text{reject}} \). The last two are called the halting states, and they cannot be equal. \( \Sigma \) is a finite input alphabet. \( \Gamma \) is a tape alphabet which includes all symbols from \( \Sigma \) and a special symbol for blank, \( \sqcup \). Finally, the transition function is \( \delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \) where \( L, R \) mean move left or right one step on the tape. Also know encoding languages and Turing machines as binary strings.

- Equivalent (not necessarily efficiently) variants of Turing machines: two-way vs. one-way infinite tape, multi-tape, non-deterministic, oblivious.

- PAL is decidable in linear time on a two-tape machine, but in quadratic time on one-tape.

- *Church-Turing Thesis* Anything computable by an algorithm of any kind (our intuitive notion of Church-Turing Thesis) is computable by a Turing machine.

- A Turing machine \( M \) accepts a string \( w \) if there is an accepting computation of \( M \) on \( w \), that is, there is a sequence of configurations (state, non-blank memory, head position) starting from \( q_0w \) and ending in a configuration containing \( q_{\text{accept}} \), with every configuration in the sequence resulting from a previous one by a transition in \( \delta \) of \( M \). A Turing machine \( M \) recognizes a language \( L \) if it accepts all and only strings in \( L \): that is, \( \forall x \in \Sigma^* \), \( M \) accepts \( x \) iff \( x \in L \). As before, we write \( \mathcal{L}(M) \) for the language accepted by \( M \).

- A language \( L \) is called Turing-recognizable (also recursively enumerable, r.e, or semi-decidable) if \( \exists \) a Turing machine \( M \) such that \( \mathcal{L}(M) = L \). A language \( L \) is called decidable (or recursive) if \( \exists \) a Turing machine \( M \) such that \( \mathcal{L}(M) = L \), and additionally, \( M \) halts on all inputs \( x \in \Sigma^* \). That is, on every string \( M \) either enters the state \( q_{\text{accept}} \) or \( q_{\text{reject}} \) in some point in computation. A language is called co-semi-decidable if its complement is semi-decidable. Semi-decidable languages can be described using unbounded \( \exists \) quantifier over a decidable relation; co-semi-decidable using unbounded \( \forall \) quantifier.

- There are languages that are higher in the arithmetic hierarchy than semi- and co-semi-decidable; they are described using mixture of \( \exists \) and \( \forall \) quantifiers and then number of alternation of quantifiers is the level in the hierarchy. An example of such decidable relation can be \( \text{Check}_A(M, w, y) \), which verifies that \( y \) is a transcript of an accepting computation of \( M \) on \( w \). \( \text{Check}_R \) and \( \text{Check}_H \) can be defined similarly for rejecting and halting computations.

- Decidable languages are closed under intersection, union, complementation, Kleene star, etc. Semi-decidable languages are not closed under complementation, but closed under intersection and union. If a language is both semi-decidable and co-semi-decidable, then it is decidable.

- Universal language \( A_{TM} = \{\langle M, w \rangle \mid w \in \mathcal{L}(M)\} \). Undecidability; proof by diagonalization and getting the paradox. \( A_{TM} \) is undecidable.

- A many-one reduction: \( A \leq_m B \) if exists a computable function \( f \) such that \( \forall x \in \Sigma^A, x \in A \iff f(x) \in B \). To prove that \( B \) is undecidable, (not semi-decidable, not co-semi-decidable) pick \( A \) which is undecidable (not semi, not co-semi.) and reduce \( A \) to \( B \). To prove that a language is in the class (e.g., semi-decidable), give an algorithm.

- Know how to do reductions and place languages in the corresponding classes, similar to the assignment (both easiness and hardness directions, where applicable).

- Examples of undecidable languages: \( A_{TM}, \text{Halt}_B, NE, Total, All \), Know which are semi-decidable, which co-semi-decidable and which neither.
Complexity theory, NP-completeness

- A Turing machine \( M \) runs in time \( t(n) \) if for any input of length \( n \) the number of steps of \( M \) is at most \( t(n) \) (worst-case running time).
- Time complexity classes \( \text{Time}(f(n)) \) are sets of languages decidable in worst-case time \( f(n) \). Similarly for \( \text{Space}(f(n)) \) and non-deterministic time \( \text{NTime}(f(n)) \). For non-deterministic time, the bound \( f(n) \) must hold for all branches of the computation.
- A language \( L \) is in the complexity class \( \text{P} \) (stands for Polynomial time) if there exists a Turing machine \( M \), \( \mathcal{L}(M) = L \) and \( M \) runs in time \( O(n^c) \) for some fixed constant \( c \). The class \( \text{P} = \bigcup_{k \geq 0} \text{Time}(n^k) \) is believed to capture the notion of efficient algorithms.
- A language \( L \) is in the class \( \text{NP} \) if there exists a polynomial-time verifier, that is, a relation \( R(x,y) \) computable in polynomial time such that \( \forall x, x \in L \iff \exists y, |y| \leq c|x|^d \land R(x,y) \). Here, \( c \) and \( d \) are fixed constants, specific for the language.
- A different, equivalent, definition of \( \text{NP} \) is a class of languages accepted by polynomial-time non-deterministic Turing machines. The name \( \text{NP} \) stands for “Non-deterministic Polynomial-time”.
- \( \text{Time}(f(n)) \subseteq \text{NTime}(f(n)) \subseteq \text{Space}(f(n)) \subseteq \text{Time}(2^{O(f(n))}) \). In particular, \( \text{P} \subseteq \text{NP} \subseteq \text{EXP} \), where \( \text{EXP} \) is the class of languages computable in time exponential in the length of the input. All of them are decidable. Alternating quantifiers, get polynomial-time hierarchy \( \text{PH} : \text{P} \subseteq \text{NP} \cap \text{coNP} \subseteq \text{NP} \cup \text{coNP} \subseteq \text{PH} \subseteq \text{PSPACE} \subseteq \text{EXP} \subseteq \text{NEXP}. \) By padding, equalities between complexity classes translate upward and inequalities downward. So if \( \text{P} = \text{NP} \) then \( \text{EXP} = \text{NEXP} \).
- Time hierarchy theorem: \( \text{Time}(f(n)) \subseteq \text{Time}(f(n)/ \log n) \). Space hierarchy theorem: \( \text{Space}(f(n)) \subseteq \text{Space}(o(f(n))). \) In particular, \( \text{P} \subseteq \text{EXP} \) and \( \text{LogSpace} \subseteq \text{PSPACE} \).
- Examples of languages in \( \text{P} \): connected graphs, relatively prime pairs of numbers (and, quite recently, prime numbers), palindromes, etc. In \( \text{NP} \): all languages in \( \text{P} \), \( \text{Clique} \), Hamiltonian Path, \( \text{SAT} \), etc. Technically, functions computing an output other than yes/no are not in \( \text{NP} \) since they are not languages. Maximizers such as \( \text{LargestClique} \) are not known to be in \( \text{NP} \).
- Major Open Problem: is \( \text{P} = \text{NP} \)? Widely believed that not, weird consequences if they were, including breaking all modern cryptography and automating creativity.
- Polynomial-time reducibility: \( A \leq_p B \) if there exists a polynomial-time computable function \( f \) such that \( \forall x \in \Sigma, x \in A \iff f(x) \in B. \)
- A language \( L \) is \( \text{NP} \)-hard if every language in \( \text{NP} \) reduces to \( L \). A language is \( \text{NP} \)-complete if it is both in \( \text{NP} \) and \( \text{NP} \)-hard.
- Steps of proving \( \text{NP} \)-completeness of a given language \( L \):
  1. Show that \( L \in \text{NP} \) by giving respective \( R, c, d \) and explaining how \( y \) encodes a solution.
  2. Show that \( L \) is \( \text{NP} \)-hard via a reduction as follows:
     (a) Find a suitable known \( \text{NP} \)-complete language \( L' \) such as \( 3\text{SAT}, \text{Partition}, \text{IndSet} \).
     (b) Describe a polynomial-time reduction \( f \) from this \( \text{NP} \)-complete language to your \( L, L' \leq_p L \), for example \( 3\text{SAT} \leq_p L \).
     (c) Show that \( x \in 3\text{SAT} \rightarrow f(x) \in L \) (or \( x \in L' \rightarrow f(x) \in L \) if \( L' \neq 3\text{SAT} \)).
     (d) Show that \( f(x) \in L \rightarrow x \in 3\text{SAT} \) (or \( f(x) \in L \rightarrow x \in L' \)).
     (e) Briefly explain why \( f \) is polynomial-time computable.
Cook-Levin Theorem states that SAT is NP-complete. The rest of NP-completeness proofs we saw are by reducing SAT (3SAT) to the other problems (also mentioned a direct proof for CircuitSAT in the notes).

Fagin theorem is a variant of Cook-Levin theorem that existential second-order logic captures NP.

If P = NP, then can compute witness y in polynomial time. Same idea as search-to-decision reductions.

Search-to-decision reductions: given an “oracle” with yes/no answers to the language membership (decision) problem in NP, can compute the solution in polynomial time with polynomially many yes/no queries. Similar idea to computing a witness if P = NP.

Ladner’s theorem: if P ≠ NP then there are problems that are neither in P nor NP-complete.

Schaefer’s theorem: all satisfiability (constraint satisfaction) problems defined as conjunctions of clauses of a certain type (Horn, 2CNF, etc) are either trivial, or complete for L, NL, ⊕L, P or NP.

Complexity theory, other topics

Boolean circuits: binary inputs, AND/OR/NOT gates. Parameters: size (# of gates) and depth (longest path from input to output) as a function of input size.

Non-uniform model (every input size can have a very different circuit). Thus, can do unary Halting problem (undecidable).

P/poly: polynomial-size circuits, corresponds to Turing machines with advice.

A Boolean function is hard if it is not computable by polynomial-size (P/poly) family of circuits. Most functions are hard by counting; big open problem if there is a hard function computable in P or even in EXP.

Parallel computation hierarchy: AC^i for log^i depth unbounded fan-in polynomial size, NC^i for log^i depth bounded fan-in circuits. AC^i \subseteq NC^{i+1} \subseteq AC^{i+1}; whole hierarchy is called NC. NC \subseteq P/poly.

AC^0 cannot do Parity; for the rest no lower bounds known. Open whether NP \subseteq P/poly.

Uniform version: require all circuits in a family to be generated by a Turing machine (e.g., logspace or polytime). Then, AC^0 \subseteq NC^1 \subseteq LogSpace \subseteq NL \subseteq AC^1 \subseteq NC^2 \subseteq \cdots \subseteq NC \subseteq P \subseteq P/poly.

Randomized computation: algorithms can use (polynomially many) random bits: M(x, r), where x is input, r is randomness.

Class RP (randomized polytime): L \in RP if x \in L, Pr_r[M(x, r) accepts] ≥ 1/2, and if x \notin L, Pr_r[M(x, r) accepts] = 0. RP \subseteq NP.

Class BPP (bounded-error probabilistic polytime): L \in BPP if x \in L, Pr_r[M(x, r) accepts] ≥ 2/3, and if x \notin L, Pr_r[M(x, r) accepts] ≤ 1/3. RP \subseteq BPP, but BPP vs NP is unknown, though BPP is in \Sigma_2^p \cap \Pi_2^p.

If there is an (exponentially) hard Boolean function in EXP, then BPP = P.