Midterm test study sheet for CS6902

Turing machines and decidability.

- A Turing machine is a finite automaton with an infinite memory (tape). Formally, a Turing machine is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. Here, Q is a finite set of states as before, with three special states q_0 (start state), q_{accept} and q_{reject}. The last two are called the halting states, and they cannot be equal. Σ is a finite input alphabet. Γ is a tape alphabet which includes all symbols from Σ and a special symbol for blank, $\sqcup$. Finally, the transition function is $\delta : Q. \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ 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 algorithm) 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_{accept}, with every configuration in the sequence resulting from a previous one by a transition in δ 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_{accept} or q_{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 $Check_A(M, w, y)$, which verifies that y is a transcript of an accepting computation of M on w. $Check_R$ and $Check_H$ 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}, $Halt_B$, 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 $Time(f(n))$ are sets of languages decidable in worst-case time $f(n)$. Similarly for $Space(f(n))$ and non-deterministic time $NTime(f(n))$. For non-deterministic time, the bound $f(n)$ must hold for all branches of the computation.
- A language L is in the complexity class `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 $\mathtt{P} = \bigcup_{k \geq 0} Time(n^k)$ is believed to capture the notion of efficient algorithms.
- A language L is in the class `NP` if there exists a *polynomial-time verifier*, that is, a relation $R(x, y)$ computable in polynomial time such that $\forall x, x \in L \iff \exists y, |y| \leq c|x|^d \wedge R(x, y)$. Here, c and d are fixed constants, specific for the language.
- A different, equivalent, definition of `NP` is a class of languages accepted by polynomial-time *non-deterministic* Turing machines. The name `NP` stands for "Non-deterministic Polynomial-time".
- $Time(f(n)) \subseteq NTime(f(n)) \subseteq Space(f(n)) \subseteq Time(2^{O(f(n))})$. In particular, $\mathtt{P} \subseteq \mathtt{NP} \subseteq \mathtt{EXP}$, where `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 `PH`: $\mathtt{P} \subseteq \mathtt{NP} \cap \mathtt{coNP} \subseteq \mathtt{NP} \cup \mathtt{coNP} \subseteq \mathtt{PH} \subseteq \mathtt{PSPACE} \subseteq \mathtt{EXP} \subseteq \mathtt{NEXP}$.
- Time hierarchy theorem: $Time(f(n)) \subsetneq Time(f(n)/\log n)$. Space hierarchy theorem: $Space(f(n)) \subsetneq Space(o(f(n)))$. In particular, $\mathtt{P} \subsetneq \mathtt{EXP}$ and $\mathtt{LogSpace} \subsetneq \mathtt{PSPACE}$.
- By *padding*, equalities between complexity classes translate upward and inequalities downward. So if $\mathtt{P} = \mathtt{NP}$ then $\mathtt{EXP} = \mathtt{NEXP}$.
- Examples of languages in `P`: connected graphs, relatively prime pairs of numbers (and, quite recently, prime numbers), palindromes,etc. In `NP`: all languages in P, Clique, Hamiltonian Path, SAT, etc. Technically, functions computing an output other than yes/no are not in `NP` since they are not languages. Maximizers such as LargestClique are not known to be in `NP`.
- Major Open Problem: is $\mathtt{P} = \mathtt{NP}$? 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 `N`-hard if every language in `NP` reduces to L. A language is `NP`-complete it is both in `NP` and `NP`-hard.
- Steps of proving `NP`-completeness of a given language L:
  1. Show that $L \in \mathtt{NP}$ by giving respective R, c, d and explaining how y encodes a solution.
  2. Show that L is `NP`-hard via a reduction as follows:
     (a) Find a suitable known `NP`-complete language L' such as $3SAT, Partition, IndSet$.
     (b) Describe a polynomial-time reduction f from this `NP`-complete language to your L, $L' \leq_p L$, for example $3SAT \leq_p L$.
     (c) Show that $x \in 3SAT \to f(x) \in L$ (or $x \in L' \to f(x) \in L$ if $L' \neq 3SAT$)
     (d) Show that $f(x) \in L \to x \in 3SAT$ (or $f(x) \in L \to x \in L'$)
     (e) Briefly explain why f is polynomial-time computable.