Midterm study sheet for CS3719

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 $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ where L, R mean move left or right one step on the tape.

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

- *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.

- 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.

- Encoding languages and Turing machines as binary strings.

- Undecidability; proof by diagonalization. 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.

- Know how to do reductions and place languages in the corresponding classes, similar to the assignment.

- Examples of undecidable languages: A_{TM}, $Halt_B$, NE, $Total$, All, $Halt0Loop1$.

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 $t(n)$.
- A language L is in the complexity class $\texttt{P}$ (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 $\texttt{P}$ is believed to capture the notion of efficient algorithms.
- A language L is in the class $\texttt{NP}$ 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 $\texttt{NP}$ is a class of languages accepted by polynomial-time *non-deterministic* Turing machines. The name $\texttt{NP}$ stands for "Non-deterministic Polynomial-time".
- $\texttt{P} \subseteq \texttt{NP} \subseteq \texttt{EXP}$, where $\texttt{EXP}$ is the class of languages computable in time exponential in the length of the input.
- Examples of languages in $\texttt{P}$: connected graphs, relatively prime pairs of numbers (and, quite recently, prime numbers), etc.
- Examples of languages in $\texttt{NP}$: all languages in P, Clique, Hamiltonian Path, SAT, etc. Technically, functions computing an output other than yes/no are not in $\texttt{NP}$ since they are not languages.
- Major Open Problem: is $\texttt{P} = \texttt{NP}$? Widely believed that not, weird consequences if they were, including breaking all modern cryptography and automating creativity.
- If $\texttt{P} = \texttt{NP}$, then can compute witness y in polynomial time.
- *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 $\texttt{N}$-hard if every language in $\texttt{NP}$ reduces to L. A language is $\texttt{NP}$-complete it is both in $\texttt{NP}$ and $\texttt{NP}$-hard.
- Cook-Levin Theorem states that SAT is $\texttt{NP}$-complete. The rest of $\texttt{NP}$-completeness proofs we saw are by reducing SAT (3SAT) to the other problems (also mentioned a direct proof for CircuitSAT in the notes).
- Examples of $\texttt{NP}$-complete problems with the reduction chain:
  - $SAT \leq_p 3SAT$
  - $3SAT \leq_p IndSet \leq_p Clique$
  - $HamCycle \leq_p TSP$ (skipped $3SAT \leq_p HamPath$; see the book.)
  - $3SAT \leq_p SubsetSum \leq_p Partition$ Reduction relies on numbers in binary; unary case solvable by dynamic programming in polynomial time.
- Search-to-decision reductions: given an "oracle" with yes/no answers to the language membership (decision) problem in $\texttt{NP}$, can compute the solution in polynomial time with polynomially many yes/no queries. Similar idea to computing a witness if $\texttt{P} = \texttt{NP}$.