Final exam 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. 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.
- *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.
- 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, $Halt0Loop1$. Know which are semi-decidable, which co-semi-decidable and which neither.
- Rice's theorem: any non-trivial property of Turing machines is undecidable (property: if $\mathcal{L}(M_1) = \mathcal{L}(L_2)$, then either both M_1 and M_2 have the property, or neither does).

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).

- A language L is in the complexity class $\mathtt{P}$ (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 $\mathtt{P}$ is believed to capture the notion of efficient algorithms.

- A language L is in the class $\mathtt{NP}$ if there exists a *polynomial-time verifier*, that is, a relation $R(x, y)$ computable in polynomial time such that $\forall x, x \in L \iff \exists y, |y| \le c|x|^d \wedge R(x, y)$. Here, c and d are fixed constants, specific for the language.

- A different, equivalent, definition of $\mathtt{NP}$ is a class of languages accepted by polynomial-time *non-deterministic* Turing machines. The name $\mathtt{NP}$ stands for "Non-deterministic Polynomial-time".

- $\mathtt{P} \subseteq \mathtt{NP} \subseteq \mathtt{EXP}$, where $\mathtt{EXP}$ is the class of languages computable in time exponential in the length of the input. It is known that $\mathtt{P} \subsetneq \mathtt{EXP}$. All of them are decidable. Alternating quantifiers, get polynomial-time hierarchy $\mathtt{PH}$: $\mathtt{P} \subseteq \mathtt{NP} \cap \mathtt{coNP} \subseteq \mathtt{NP} \cup \mathtt{coNP} \subseteq \mathtt{PH} \subseteq \mathtt{PSPACE} \subseteq \mathtt{EXP}$.

- Examples of languages in $\mathtt{P}$: connected graphs, relatively prime pairs of numbers (and, quite recently, prime numbers), palindromes,etc.

- Examples of languages in $\mathtt{NP}$: all languages in P, Clique, Hamiltonian Path, SAT, etc. Technically, functions computing an output other than yes/no are not in $\mathtt{NP}$ since they are not languages.

- Examples of languages not known to be in $\mathtt{NP}$: LargestClique, TrueQuantifiedBooleanFormulas.

- 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.

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

- *Polynomial-time reducibility*: $A \le_p B$ 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 $\mathtt{N}$-hard if every language in $\mathtt{NP}$ reduces to L. A language is $\mathtt{NP}$-complete it is both in $\mathtt{NP}$ and $\mathtt{NP}$-hard.

- Cook-Levin Theorem states that SAT is $\mathtt{NP}$-complete. The rest of $\mathtt{NP}$-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 $\mathtt{NP}$-complete problems with the reduction chain:
  - $SAT \le_p 3SAT$
  - $3SAT \le_p IndSet \le_p Clique$
  - $HamCycle \le_p HamPath \le stHamPath$ (skipped $3SAT \le_p HamPath$ and $HamCycle \le_p TSP$; see the book.)
  - $Partition \le_p SubsetSum \le_p GKP$ (skipped $3SAT \le SubsetSum$; see the book.) .

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

Algorithm design for languages in $\mathtt{P}$

- *Greedy algorithms* Sort items then go through them either picking or ignoring each; never reverse a decision. Running time usually $O(n \log n)$ where n is the number of elements (depends on data structures used, too). Often does not work or only gives an approximation; when it works, correctness proof by induction on the number of steps (i.e., S_i is the solution set after considering i^{th} element in order.)
  - Base case: show $\exists S_{opt}$ such that $S_0 \subseteq S_{opt} \subseteq S_0 \cup \{1, \ldots, n\}$.
  - Induction hypothesis: assume $\exists S_{opt}$ such that $S_i \subseteq S_{opt} \subseteq S_i \cup \{i+1, \ldots, n\}$.
  - Induction step: show $\exists S'_{opt}$ such that $S_{i+1} \subseteq S'_{opt} \subseteq S_{i+1} \cup \{i+2, \ldots, n\}$.
    1. Element $i+1$ is not in S_{i+1}. Argue that S_{opt} does not have it either, then $S'_{opt} = S_{opt}$.
    2. Element $i+1$ is in S_{i+1}. Either S_{opt} has it (possibly in the different place – then switch things around to get S'_{opt}), or S_{opt} does not have it, then throw some element j out of S_{opt} and put $i+1$ instead for S'_{opt}; argue that your new solution is at least as good.
- Examples of greedy algorithms: Kruskal's algorithm for Minimal Spanning Tree, activity selection, scheduling with deadlines and profits, problem from the assignment...
- *Dynamic programming* Precompute partial solutions starting from the base cases, keep them in a table, compute the table from already precomputed cells (e.g., row by row, but can be different). Arrays can be 1,2, 3-dimensional (possibly more), depends on the problem. Steps of design:
  1. Define an array; that is, state what are the values being put in the cells, then what are the dimensions and where the value of the best solution is stored. E.g.: $A(i, t)$ stores the profit of the best schedule for jobs from 1 to i finishing by time t, where $1 \leq i \leq n$, and $0 \leq t \leq maxd_i$. Final answer value is $A(n, maxd_i)$.
  2. Give a recurrence to compute A from the previous cells in the array, including initialization. E.g. (longest common subsequence) $A(i,j) = \begin{cases} A(i-1, j-1) + 1 & x_i = y_j \\ \max\{A(i-1, j), A(i, j-1)\} & \text{otherwise} \end{cases}$
  3. Give pseudocode to compute the array (usually we omitted it in class).
  4. Explain how to recover the actual solution from the array (usually using a recursive $PrintOpt()$ procedure to retrace decisions).
- Running time a function of the size of the array – might be not polynomial (e.g., scheduling with very large deadlines)!
- Examples: Scheduling, Knapsack, Longest Common Subsequence, Longest Increasing Subsequence, All Pair Shortest Path (Floyd-Warshall).
- *Backtracking* Used when others don't work; usually exponential time, but faster than testing all possibilities. Make a decision tree of possibilities, go through the tree recursively, if some possibilities fail, backtrack.

Regular languages and finite automata:

- An *alphabet* is a finite set of symbols. Set of all finite strings over an alphabet Σ is denoted Σ^*. A *language* is a subset of Σ^*. Empty string is called ϵ (epsilon).
- *Regular expressions* are built recursively starting from $\emptyset, \epsilon$ and symbols from Σ and closing under Union ($R_1 \cup R_2$), Concatenation ($R_1 \circ R_2$) and Kleene Star (R^* denoting 0 or more repetitions of R) operations. These three operations are called regular operations.

- A Deterministic Finite Automaton (DFA) D is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, q_0 is the start state, and F is the set of accept states. A DFA accepts a string if there exists a sequence of states starting with $r_0 = q_0$ and ending with $r_n \in F$ such that $\forall i, 0 \leq i < n, \delta(r_i, w_i) = r_{i+1}$. The language of a DFA, denoted $\mathcal{L}(D)$ is the set of all and only strings that D accepts.

- Deterministic finite automata are used in string matching algorithms such as Knuth-Morris-Pratt algorithm.

- A language is called *regular* if it is recognized by some DFA.

- The class of regular languages is closed under union, concatenation and Kleene star operations.

- A *non-deterministic* finite automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0 and F are as in the case of DFA, but the transition function δ is $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$. Here, $\mathcal{P}(Q)$ is the powerset (set of all subsets) of Q. A non-deterministic finite automaton *accepts* a string $w = w_1 \dots w_m$ if there exists a sequence of states $r_0, \dots r_m$ such that $r_0 = q_0$, $r_m \in F$ and $\forall i, 0 \leq i < m, r_{i+1} \in \delta(r_i, w_i)$.

- **Theorem:** For every NFA there is a DFA recognizing the same language. The construction sets states of the DFA to be the powerset of states of NFA, and makes a (single) transition from every set of states to a set of states accessible from it in one step on a letter following with all states reachable by (a path of) ϵ-transitions. The start state of the DFA is the set of all states reachable from q_0 by following possibly multiple ϵ-transitions.

- **Theorem:** A language is recognized by a DFA if and only if it is generated by some regular expression. In the proof, the construction of DFA from a regular expression follows the closure proofs and recursive definition of the regular expression.

- The class of regular languages is polynomial-time decidable; moreover, it is possible to decide in polynomial time, given a description of a DFA or an NFA and a string, whether this DFA/NFA accepts this string. For DFAs, just simulate it; for NFA, need to keep the list of all states where the automaton could be at the moment and need reachability for ϵ-arrows.

- **Lemma** The *pumping lemma for regular languages* states that for every regular language A there is a pumping length p such that $\forall s \in A$, if $|s| > p$ then $s = xyz$ such that 1) $\forall i \geq 0, xy^iz \in A$. 2) $|y| > 0$ 3) $|xy| < p$. The proof proceeds by setting p to be the number of states of a DFA recognizing A, and showing how to eliminate or add the loops. This lemma is used to show that languages such as $\{0^n1^n\}, \{ww^r\}$ and so on are not regular.

Context-free languages and Pushdown automata.

- A *pushdown automaton* (PDA) is a "NFA with a stack"; more formally, a PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is the set of states, Σ the input alphabet, Γ the stack alphabet, q_0 the start state, F is the set of finite states and the transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$.

- A *context-free grammar* (CFG) is a 4-tuple (V, Σ, R, S), where V is a finite set of variables, with $S \in V$ the start variable, Σ is a finite set of `terminals` (disjoint from the set of variables), and R is a finite set of rules, with each rule consisting of a variable followed by $\to$ followed by a string of variables and terminals.

- Let $A \to w$ be a rule of the grammar, where w is a string of variables and terminals. Then A can be replaced in another rule by w: uAv in a body of another rule can be replaced by uwv (we say uAv yields uwv,denoted $uAv \Rightarrow uwv$). If there is a sequence $u = u_1, u_2, \dots u_k = v$ such that for all i, $1 \leq i < k$, $u_i \Rightarrow u_{i+1}$ then we say that u derives v (denoted $v \overset{*}{\Rightarrow} v$.) If G is a context-free grammar, then the language of G is the set of all strings of terminals that can be generated from the start variable: $\mathcal{L}(G) = \{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$. A *parse tree* of a string is a tree representation of a sequence of derivations; it is *leftmost* if at every step the first variable from the left was substituted. A grammar is called *ambiguous* if there is a string in a grammar with two different (leftmost) parse trees.

- A language is called a *context-free language* (CFL) if there exists a CFG generating it.

- **Theorem** Every regular language is context-free.

- **Theorem** A language is context-free iff some pushdown automaton recognizes it. The proof of one direction constructs a PDA from the grammar (by having a middle state with "loops" on rules; loops consist of as many states as needed to place all symbols in the rule on the stack). The proof for another direction constructs a grammar that for every pair of states has a variable and a rule generating strings for a sequence of steps between these states keeping stack content.

- A grammar is in Chomsky normal form if all rules are of the form $A \to BC$ or $A \to a$, only $S \to \epsilon$ allows to have ϵ, and S occurs only on the left side. Any grammar can be converted into Chomsky normal form; if the last rule in Sipser's construction is done first, then the resulting grammar is polynomial size.

- A derivation of a string w in a grammar in Chomsky normal form takes exactly $2|w| - 1$ steps.

- Context-free languages are decidable in polynomial time via a dynamic programming algorithm that, given a grammar in Chomsky normal form and a string, decided if this grammar derives the string.

- **Lemma** The *pumping lemma for context-free languages* states that for every CFL A there is a pumping length p such that $\forall s \in A$, if $|s| > p$ then $s = uvxyz$ such that 1) $\forall i \geq 0, uv^ixy^iz \in A$. 2) $|vy| > 0$ 3) $|vxy| < p$. The proof proceeds by analyzing repeated variables in large parse trees, setting the pumping length to $d^{|V|+1}$ where d is the length (number of symbols) of the longest rule. This lemma is used to show that languages such as $\{a^nb^nc^n\}, \{ww\}$ and so on are not regular.

- **Theorem** The class of CFLs is *not* closed under complementation and intersection (although it is closed under union, Kleene star and concatenation). Example: complement of $\{a^nb^nc^n\}$ is context-free, but $\{a^nb^nc^n\}$ is not.

- **Theorem** There are context-free languages not recognized by any deterministic PDA. Example: also complement of $\{a^nb^nc^n\}$, since deterministic PDA are closed under complementation.