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 7-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 $q_0 w$ 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 `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 `P` 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".
- `P` $\subseteq$ `NP` $\subseteq$ `EXP`, where `EXP` is the class of languages computable in time exponential in the length of the input.
- Examples of languages in `P`: connected graphs, relatively prime pairs of numbers (and, quite recently, prime numbers), etc.
- Examples of languages 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.
- Major Open Problem: is `P` = `NP`? Widely believed that not, weird consequences if they were, including breaking all modern cryptography and automating creativity.
- If `P` = `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 `NP`-hard if every language in `NP` reduces to L. A language is `NP`-complete it is both in `NP` and `NP`-hard.
- 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).
- Examples of `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 `NP`, can compute the solution in polynomial time with polynomially many yes/no queries. Similar idea to computing a witness if `P` = `NP`.

Algorithm design techniques

- *Greedy algorithms* and *dynamic programming* were the main design paradigms we looked at (we also talked about backtracking, but just as a case when dynamic programming does not work).
- Greedy algorithms work when there is a way to pick a locally optimal choices that lead to a globally optimal solution. In the proof, this is described as having a "promising solution" after each step of the algorithm: the partial solution constructed so far can be extended to the optimal using only (a subset of) remaining elements. That is, $S_{i+1} \subseteq S_{opt} \subseteq S_{i+1} \cup \{i+2, \ldots, n\}$. This property is proven by induction on i for a correctness proof of the algorithm.
  - Base case: $\exists S_{opt}$ such that $S_0 \subseteq S_{opt} \subseteq \{1, \ldots, n\}$: there is some set (possibly empty) satisfying the constraints, so some set satisfying the constraints is optimal, and any set extends empty set S_0.
  - Induction step: Assume that the algorithm worked correctly for the first i steps, that is, a partial solution S_i is promising: $S_i \subseteq S_{opt} \subseteq S_i \cup \{i+1, \ldots, n\}$. Now, show that there exists an optimal solution S'_{opt}, $S_{i+1} \subseteq S'_{opt} \subseteq S_{i+1} \cup \{i+2, \ldots, n\}$. Consider cases: the algorithm pick/does not pick the element $i+1$, and S_{opt} already has/does not have $i+1$. In case of discrepancy, show how to modify S_{opt} to obtain an equivalent solution containing $i+1$ ("exchange lemma").
  - Conclude that S_n must be optimal, since $S_n \subseteq S_{opt} \subseteq S_n \cup \emptyset$.
- Examples of greedy algorithms: Kruskal's, Fractional Knapsack, Scheduling with deadlines and profits (no durations) ; 1/2 approximation for monitors on links, 1/2 approximation for Knapsack.
- Often, running time of a greedy algorithm is $O(n \log n)$, dominated by the sorting part.
- Examples of dynamic programming algorithms: Floyd-Warshall, Longest Common Subsequence, scheduling with deadlines, profits and durations.
- Steps of designing a dynamic programming algorithm:
  1. Define the array, specifying the ranges of the variables and stating how to get the value of the best solution from the array (e.g., "$A(i,j)$ contains the longest common subsequence of x[1..i] and y[1..j]).
  2. Give a recurrence computing the values in the array, including initialization (e.g., $A(i,j) = \ldots$)
  3. Give pseudocode for the algorithm (filling the array and extracting the value of the answer)
  4. Describe how to compute the actual solution, not just the value: $PrintOpt()$.
- For the test, concentrate on Step 1 and Step 2. Be able to fill an array for a given example.
- Running time of a dynamic programming is usually comparable to the size of the table (if there is a max or min over non-constant number of choices, then times that number of choices). For the case of scheduling and knapsack, that could result in non-polynomial running time $O(nd)$, where d is the maximal deadline/capacity.
- Know scheduling algorithms (notes) and be able to say in which case which algorithm works (e.g., that dynamic programming is not polynomial time if the deadlines are more than exponentially larger than the number of jobs, and greedy works when there are no durations), and what you'd do in that case (backtracking). Know Knapsack problem and its variants (Fractional Knapsack can be solved by greedy, but Simple Knapsack is already NP-complete and can be reduced to scheduling, but can be approximated to within the factor of 2 by a greedy algorithm).

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, $\delta : Q \times \Sigma \to Q$ 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.
- 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 $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$. 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.
- **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 $->$ 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 $u = u_1, u_2, \ldots u_k = v$ such that for all i, $1 \leq i < k$, $u_i \Rightarrow u_{i+1}$ then we say that u derives v (denoted $v \stackrel{*}{\Rightarrow} v$.) 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: $\mathcal{L}(G) = \{w \in \Sigma^* | S \stackrel{*}{\Rightarrow} w\}$. 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.
- **Theorem** Every CFL (and, thus, every regular language) membership problem is solvable in polynomial time. That is, given a string and a grammar, it is possible to determine in polynomial time whether this grammar generates this string.
- There is a "pumping lemma" for CFLs, analogous to pumping lemma for regular languages, but slightly more complicated. It can be used to show that $a^n b^n c^n$ is not CFL.
- **Theorem** There are context-free languages not recognized by any deterministic PDA. (No proof given in class).