Exam study sheet for CS3719

Regular languages and finite automata

- An alphabet is a finite set of symbols. Set of all finite strings over an alphabet \( \Sigma \) is denoted \( \Sigma^* \). A language is a subset of \( \Sigma^* \). Empty string is called \( \epsilon \) (epsilon).

- Regular expressions are built recursively starting from \( \emptyset, \epsilon \) and symbols from \( \Sigma \) and closing under Union \( (R_1 \cup R_2) \), Concatenation \( (R_1 \circ R_2) \) and Kleene Star \( (R^* \text{ denoting 0 or more repetitions of } R) \) 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, \( \Sigma \) 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.

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

- Theorem: 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 \( Q, \Sigma, q_0 \) and \( F \) are as in the case of DFA, but the transition function \( \delta \) 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 \( w = w_1 \ldots w_m \) if there exists a sequence of states \( r_0, \ldots r_m \) 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 ) \( \epsilon \)-transitions. The start state of the DFA is the set of all states reachable from \( q_0 \) by following possibly multiple \( \epsilon \)-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 construction of a regular expression from a DFA first converts DFA into a Generalized NFA with regular expressions on the transitions, a single distinct accept state and transitions (possibly \( \emptyset \)) between every two states. The proof proceeds inductively eliminating states until only the start and accept states are left.

- Lemma The pumping lemma for regular languages states that for every regular language \( A \) there is a pumping length \( p \) such that \( \forall s \in A, \text{if } |s| > p \text{ then } s = xyz \text{ such that 1) } \forall i \geq 0, xy^iz \in A. \text{ 2) } |y| > 0 \text{ 3) } |xy| < p. \) 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 \( \{0^n1^n\} \), \( \{wwx^r\} \) 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, \( \Sigma \) the input alphabet, \( \Gamma \) the stack alphabet, \( q_0 \) the start state, \( F \) is the set of finite states and the transition function \( \delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\})). \)
• A context-free grammar (CFG) is a 4-tuple \((V, \Sigma, R, S)\), where \(V\) is a finite set of variables, with \(S \in V\) the start variable, \(\Sigma\) 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 \(A \rightarrow w\) 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 \(uvw\) (we say \(uAv\) yields \(uvw\), denoted \(uAv \Rightarrow uvw\)). 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 \(u \Rightarrow^* v\)). 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: \(L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}\). 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).

• 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) \(\forall i \geq 0, uv^ixyz \in A\). 2) \(|vy| > 0\) \(|vx| < p\). This lemma is used to show that languages such as \(\{a^n b^n c^n\}\), \(\{ww\}\) 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).

• Theorem There are context-free languages not recognized by any deterministic PDA.

Turing machines and decidability.

• A Turing machine is a finite automaton plus an infinite read/write 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 \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) 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.

• 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 \(q_{\text{accept}}\), 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 \(L(M)\) for the language accepted by \(M\).
A language \( L \) is called \emph{Turing-recognizable} (also \emph{recursively enumerable}, \emph{r.e.}, or \emph{semi-decidable}) if \( \exists \) a Turing machine \( M \) such that \( \mathcal{L}(M) = L \). A language \( L \) is called \emph{decidable} (or \emph{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 \emph{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; the number of alternations of quantifiers is the level in the hierarchy. In particular, the decidable predicate can be \( V_A(M, w, y) \) which is true iff \( y \) encodes an accepting computation of \( M \) on \( w \). \( V_R \) and \( V_H \) are defined similarly for \( y \) a rejecting and a halting computation, respectively.

- If a language is both semi-decidable and co-semi-decidable, then it is decidable.
- Universal language \( A_{TM} = \{ (M, w) \mid w \in \mathcal{L}(M) \} = \{ (M, w) \mid \exists y V_A(M, w, y) \} \). \( A_{TM} \) is undecidable: proof by contradiction. Examples of undecidable languages: \( A_{TM}, \text{Halt}_B, \text{NE} \) (semi-decidable), \( \text{Empty} \) (co-semi-decidable), \( L = \{ (M_1, w_1, M_2, w_2) \mid w_1 \in \mathcal{L}(M_1) \text{ and } w_2 \notin \mathcal{L}(M_2) \} \) Total (neither), three languages from the assignment.

A \emph{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 \( L \) is in a class (e.g., semi-decidable), give an algorithm (e.g, \( M_L \)).

**Complexity theory, \( \text{NP} \)-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 \( \mathcal{P} \) (stands for \emph{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 \( \mathcal{P} \) is believed to capture the notion of efficient algorithms.
- A language \( L \) is in the class \( \text{NP} \) if there exists a \emph{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 \emph{non-deterministic} Turing machines. The name \( \text{NP} \) stands for “Non-deterministic Polynomial-time”.
- \( \mathcal{P} \subseteq \text{NP} \subseteq \text{EXP} \), where \( \text{EXP} \) is the class of languages computable in time exponential in the length of the input. It is known that \( \mathcal{P} \subseteq \text{EXP} \). All of them are decidable. Alternating quantifiers, get polynomial-time hierarchy \( \text{PH} : \mathcal{P} \subseteq \text{NP} \cap \text{coNP} \subseteq \text{NP} \cup \text{coNP} \subseteq \text{PH} \subseteq \text{PSPACE} \subseteq \text{EXP} \).
- Examples of languages in \( \mathcal{P} \): connected graphs, relatively prime pairs of numbers (and, quite recently, prime numbers), palindromes, etc.
- Examples of languages in \( \text{NP} \): all languages in \( \mathcal{P} \), Clique, Hamiltonian Path, SAT, etc. Technically, functions computing an output other than yes/no are not in \( \text{NP} \) since they are not languages.
- Examples of languages not known to be in \( \text{NP} \): LargestClique, TrueQuantifiedBooleanFormulas.
- Major Open Problem: is \( \mathcal{P} = \text{NP} \)? Widely believed that not, weird consequences if they were, including breaking all modern cryptography and automating creativity.
• If \( P = \text{NP} \), then can compute witness \( y \) in polynomial time. Same idea as search-to-decision reductions.

• **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 it is both in \( \text{NP} \) and \( \text{NP} \)-hard.

• Cook-Levin Theorem states that \( \text{SAT} \) is \( \text{NP} \)-complete. The rest of \( \text{NP} \)-completeness proofs we saw are by reducing \( \text{SAT} \) (3SAT) to the other problems (also mentioned a direct proof for CircuitSAT in the notes).

• Examples of \( \text{NP} \)-complete problems with the reduction chain:
  - \( \text{SAT} \leq_p 3\text{SAT} \)
  - \( 3\text{SAT} \leq_p \text{IndSet} \leq_p \text{Clique} \)
  - \( \text{Partition} \leq_p \text{SubsetSum} \leq_p \text{GKP} \) (skipped \( 3\text{SAT} \leq \text{SubsetSum} \); see the book.)

Algorithm design and analysis

• **Greedy algorithms** (chapter 12-13) 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.) Examples of greedy algorithms: 2-approximation for Simple Knapsack, Fractional knapsack, Kruskal’s algorithm for Minimal Spanning Tree, Dijkstra’s algorithm, scheduling with deadlines and profits.

• **Dynamic programming** (chapter 12-13) 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. 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). 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 \( 0 \leq t \leq \text{maxd}_i \). Final answer value is \( A(n, \text{maxd}_n) \).
  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 \( \text{PrintOpt}() \) procedure to retrace decisions).

• Dynamic programming does not necessarily produces polynomial-time algorithms (e.g. Scheduling, Knapsack, SubsetSum, where the value of a binary variable is an array dimension).

• For some problem, dynamic programming allows to get arbitrary good approximation (trading off array size vs. accuracy).
• **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.

• **Network flows**
  
  - *Flow network*: a directed graph with (non-negative) weights on edges called capacities $c_e$, and two special vertices *source* $s$ and *sink* $t$.
  
  - *Flow* in a flow network is a function $f : E \to \mathbb{R}^+$ from edges to numbers satisfying
    1. *Capacity constraints*: For every edge $e \in E$ $0 \leq f(e) \leq c_e$. That is, flow on an edge never exceeds the edge’s capacity.
    2. *Flow conservation*: For every vertex $v$ other than $s$ and $t$, $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$.
      
      That is, for all intermediate vertices everything that goes into $v$ has to get out.

  - A *cut* in a flow network is partition of vertices into a set $S$ containing $s$, and a set $T$ containing $t$. The capacity of a cut is the sum of capacities of all edges from vertices in $S$ to vertices in $T$.

  - *Min cut max flow theorem*: The maximum possible flow in a flow network equals capacity of the smallest-capacity cut.

  - Given a flow network $G$ with some flow $f$, the *residual network* $G_f$ contains 1) all edges of $G$ with remaining capacity $c_e - f(e) > 0$, with capacities $c_e - f(e)$, 2) For each edge $(u, v)$ with flow $f(e) > 0$, a backward edge $(v, u)$ with capacity $f(e)$.

  - An *augmenting path* is a path from $s$ to $t$ in the residual network. Its capacity is the minimum capacity over edges on the path.

  - *Ford-Fulkerson-Edmonds-Karp algorithm*: Start with the original flow network as residual network. While $t$ is reachable from $s$ in the residual network, select the shortest augmenting path, update the residual network with flow along this augmenting path equal to its capacity.

  - When there are no more augmenting paths, a minimum-capacity cut consists of $S =$ all vertices reachable from $s$, $T$ the rest.

  - To solve maximum matching in a bipartite graph problem, add new vertices $s$ and $t$ to the graph, with edges from $s$ to one side of the bipartite graph and from the other side to $t$. Make all capacities 1. Compute the max. flow. Now, all edges between two sides of the graph with non-zero flow form a maximum matching.