CMPSCI 501: Formal Language Theory **Spring 2021**

Final Review Sheet

Rik Sengupta

Here are some topics that we covered in class. This list is by no means exhaustive, but may be useful as an exam prep tool.

List of topics

Section	*Problem or Topic*	*Description or Technique*
Regular Languages	DFAs	
	NFAs and equivalence with DFAs	The subset construction
	Regular expressions	Inductive definition
	DFA-recognizable $\equiv$ regular	Inductive proof using NFAs
	Closure properties	Most set-theoretic / group-theoretic operations
	The Pumping Lemma	Proving a language is non-regular
	L-equivalence and L-distinguishability	Proves lower bounds on size of DFA state space
	The Myhill-Nerode Theorem	Alternative proof of non-regularity
Context-Free Languages	CFGs	
	Ambiguity	Leftmost derivations to fix this
	Chomsky Normal Form	Establishes upper bounds
	PDAs	*Must* be nondeterministic
	PDA-recognizable $\equiv$ CFL	
	Closure properties	Some set-theoretic / group-theoretic operations
	The Pumping Lemma	Proving a language is non-CF
Turing Machines	Definitions	
	Computation histories	Low-level / representations
Decidability	A_{DFA}, A_{NFA}, A_{LBA}	Decidable (simulation)
	E_{DFA}, E_{NFA}, E_{CFG}	Decidable (graph search)
	EQ_{DFA}	Decidable (symmetric difference)
	A_{CFG}	Decidable (derive from CNF)
	any CFL	Decidable (use the A_{CFG}-decider)
Undecidability	A_{TM}	Undecidable (diagonalization)
	$HALT_{TM}$, E_{TM}, $REGULAR_{TM}$	Undecidable (reduce from A_{TM})
	$\overline{A_{TM}}$	Unrecognizable
	E_{LBA}, ALL_{CFG}, PCP, $EQ_{2DIM\text{-}DFA}$	Undecidable (computation histories)
	EQ_{CFG}	Undecidable (reduce from ALL_{CFG})
	Computable functions, mapping reductions	
	EQ_{TM}	Unrecognizable, co-unrecognizable
	Rice's Theorem	Determining a *nontrivial* property is undecidable
Miscellaneous Topics	Enumerators	TD and TR characterizations
	The Post Correspondence Problem	Proofs with computation histories
	LBAs	
	The Arithmetic Hierarchy	What is the "quantifier depth" of a problem?
	The Recursion Theorem	A TM can obtain its own description
	Kolmogorov complexity	String compressions and associated results

List of topics		
Section	*Problem or Topic*	*Description or Technique*
Time Complexity	P, NP, coNP, NP-complete, NP-hard	
	PATH, *RELPRIME*, etc.	$\in$ P, because of poly-time *algorithms*
	HAMPATH, *COMPOSITES*, etc.	$\in$ NP, because of poly-time *verifiers* (or *certificates*)
	SAT, *3SAT*	NP-complete (Cook-Levin)
	Polynomial time reductions	
	CLIQUE, *VERTEX-COVER*, *SUBSET-SUM*, *HAMPATH*, *UHAMPATH*, *3COLOR*	NP-complete (reduce from SAT or 3SAT)
	MAX-CUT, *DOMINATING-SET*, *SET-COVER*, *HITTING-SET*	NP-complete (reduce from various other problems)
Space Complexity	PSPACE, L, NL, PSPACE-/NL-complete	Examples of canonical problems
	SAT, and almost all standard problems	$\in$ PSPACE, because of poly-space *algorithms*
	Savitch's Theorem	PSPACE = NPSPACE
	TQBF	PSPACE-complete (recursive calls)
	FORMULA-GAME, *GG*	PSPACE-complete (reduce from TQBF)
	Log-space reductions	
	PATH	NL-complete (accepting configurations)
	NL = coNL	Immerman-Szelepcsényi (show $\overline{PATH} \in$ NL)
Advanced Topics	Time/Space Hierarchy Theorems	(Statements only)
	Circuits	size, depth, complexity, gates, fan-in
	PRAM, NC^i, AC^i, TC^i, branching programs	Hierarchies and canonical problems
	Alternation	Space-time tradeoff for alternation
Miscellaneous Topics	The Polynomial Hierarchy	PH $\subseteq$ PSPACE
	The Alternation Theorem	AP = PSPACE, AL = P, APSPACE = EXPTIME
	P-completeness	CVP

Some useful definitions and terminology

1. **Turing Machines and Strings**

   - *Dovetailing* is the concept of running a single machine in parallel on many different inputs, as if it were running simultaneously on all of them. Typically, this comes up in computation theory proofs, where we can simulate this behavior on an ordinary TM M as follows: let $\sigma_1, \sigma_2, \ldots$ be a lexicographical enumeration of all strings in Σ^*. For $i = 1$ to ∞, run M for i steps on machines $\sigma_1, \ldots, \sigma_i$. This is called dovetailing, as eventually we are guaranteed to simulate any specific move by the TM move on every possible finite input.
   - *Lexicographical ordering* in this course is the ordering where we sort by length *first*, and then alphabetically within strings of the same length.

2. **Graphs**

   - An undirected graph $G = (V, E)$ is *connected* when there is a path in G between any pair $u, v \in V$.
   - For $G = (V, E)$, a subgraph H is *spanning* if it includes all vertices in V. A *spanning tree* of G is a connected acyclic subgraph that spans G.
   - The complement of a graph G, denoted $\overline{G}$, is typically defined as the graph obtained by taking the same vertices, and flipping the edges and non-edges. Formally, if $G = (V, E)$, then $\bar{G} = (V, \binom{V}{2} - E)$.
   - A graph G is *bipartite* if its vertices can be partitioned into two subsets U and V such that all edges in G go from U to V (i.e. there is no edge internal to U or V). A graph G is *k-partite* if its vertices can be partitioned into k subsets $U_1, \ldots, U_k$ such that all edges in G go from U_i to U_j for $i \neq j$.
   - A *complete graph* on n vertices, denoted K_n, is the graph on n vertices such that any two of these vertices has an edge between them. A *complete bipartite graph* $K_{m,n}$ consists of m vertices U and n vertices V such that there is an edge between any $u \in U$ and $v \in V$. It is clear that K_n has $\binom{n}{2}$ edges, while $K_{m,n}$ has mn edges.
   - A *clique* of size k in a graph G is a K_k-subgraph in G. An independent set of size k consists of k distinct vertices in G that are pairwise *not* connected by an edge. It is clear that an independent set of size k corresponds to an induced $\bar{K_k}$ in G.

- A *proper k-coloring* of a graph G is a way to assign k or fewer colors to the vertices of G such that any pair of adjacent vertices receives different colors. Bipartite graphs correspond precisely to 2-colorable graphs; k-partite graphs correspond to k-colorable graphs; K_n is n-colorable but not n'-colorable for any $n' < n$. $K_{m,n}$ is 2-colorable.
- For a graph G, a *vertex cover* is a subset S of vertices such that every edge of G is incident on at least one vertex in S. It can be proved that a vertex cover in G corresponds bijectively to an independent set in G (why?).

3. **Boolean Formulas**

- A *variable* is defined in the usual way: it is a symbol that takes a Boolean value, either 0 or 1. A *literal* is either a variable or its negation. The formula $(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3)$ has three variables and four literals. Note that a formula on a set of variables does not need to contain all the variables.
- A *clause* consists of either a single literal, or a maximal disjunction (ORs) of literals. The formula above has two clauses.
- A formula is in *conjunctive normal form* or CNF if it is either a single clause, or a conjunction (AND) of clauses. The formula above is in CNF.

4. **Miscellaneous**

- A *certificate* (or a *witness*) is a string that certifies the answer to a computation, or provides a "proof" of membership of a word in a given language. In this course, we have used certificates typically for proving membership in NP and NL.

Some random thoughts and notes

1. Typically, showing that a statement is *true* needs a *proof* (usually an argument); showing it is *false* needs a *counterexample* (usually a construction).
2. Remember that coNP is *not* the complement of NP. By definition, a problem X is in coNP if and only if $\overline{X}$ is in NP. Why does this not imply coNP and NP are not complements?
3. For each complexity class (e.g. undecidable, unrecognizable, un-corecognizable, P, NP, NL, etc), you should know a **canonical** example of a *complete* problem for that class, for ease of reductions.
4. Remember the generic template for showing a problem X is not in a particular class (e.g. decidable). First, pick a canonical problem Y that is known to also be in that class. Take an *arbitrary* instance of the problem Y, and *use* that instance to construct an instance of the problem X with the guarantee that the X-instance is in X if and only if the Y-instance is in Y (i.e. you are using a hypothetical "black box" procedure for X to conclude that problem Y cannot be too much harder than X, which is a contradiction, proving such a black-box procedure cannot exist for X. That's the tl;dr version of a *reduction.*
5. Be careful about which results or statements apply to directed graphs, and which ones to undirected graphs (and which ones apply to both). The *HAMPATH* problem is NP-complete both for directed graphs *and* for undirected graphs. The *PATH* problem is NL-complete for directed graphs, but for undirected graphs it can be shown to be in L.
6. Be careful about which results or statements apply to positive integers, and which ones to all integers, and other similar assumptions. The *SUBSET-SUM* problem can be shown to work with all integers, not just for positive ones. We often state it for multisets, but it makes no difference if we defined it as a set instead. Be aware of these assumptions.
7. In particular, note that coNP $\cap$ NP *contains* P. We do not know if it equals P, however. Note the difference with Recognizable $\cap$ co-Recognizable = Decidable.
8. To show a problem is *complete* for a complexity class X, you need to show TWO things: the problem is *in* X (an upper bound), and the problem is X-hard (a lower bound). This last condition means all problems in X are *reducible* (in the meaningful sense, depending on the context) to X.
9. Be careful about which class contains which other class. We know the chain L $\subseteq$ NL $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subseteq$ EXPSPACE $\subseteq$ DECIDABLE $\subseteq$ RECOGNIZABLE. By the Hierarchy Theorems, we know some of these containments are strict, but we also do not know about some. It is very important to remember some strict containments, noting that the Hierarchy Theorems only apply to classes of the same *type* (space or time). For instance, we know $L \subsetneq PSPACE \subsetneq EXPSPACE$, and $P \subsetneq EXPTIME$.

10. Remember that the theorem NL = coNL is *extremely* useful. It means in order to show membership in NL, it suffices to show membership in coNL (and vice versa).

11. Remember all log-space reductions are polynomial-time reductions as well, but we do not know if the converse is true.

12. For each complexity class, you should know a **canonical** example of a problem in that complexity class. For instance, you should know $MAJORITY$ is in NC^1, and that *COMPOSITES* $\in$ NP. Most importantly, you should understand why.

13. For many complexity classes, you should understand the notion of being complete for that class in terms of reductions. The same notion of reductions will not in general hold for all complexity classes: for instance, it makes no sense to use polynomial time reductions for NL-completeness.

14. You should know which standard operations a class is closed under (keeping in mind that we do not know the answer to many of these). Given any complexity class, is it closed under union? Concatenation? Intersection? Complementation? Kleene star? It is a good idea to see which of these is obvious. For instance, any *deterministic* complexity class is closed under complementation. This is not necessarily true of nondeterministic classes.

15. Keep in mind many polynomial-time algorithms you are familiar with can actually be implemented in smaller complexity classes. For instance, from algorithms courses, we are used to thinking about the undirected PATH problem as an $O(m+n)$ algorithm (BFS or DFS), so PATH is clearly in P. However, as we know from class, the undirected PATH problem is a canonical example of a problem in L as well.

16. Finally, one nice big-picture takeaway from this course is that there are three models of complexity that are all in some sense equivalent: Turing Machines, circuits, and logical descriptions. There are notions of complexity for each of them that turn out to be equivalent, and so they all divide, sort, and neatly categorize the world of problems into the so-called *complexity zoo*, where any problem can be slotted into its appropriate place. There is much we do not know about the zoo, and it is constantly expanding as new complexity classes are defined every week, but the zoo itself forms the underlying structure of all computational problems, and rigorously defines what it means to be an intractable problem, what constitutes an easy one, and what the limits are to what we can do.