CS 154 – Introduction to Automata and Complexity Theory

Final Review Notes

Kevin Lee[1], Spring Quarter 1999 - 2000

Decision problem Given an alphabet Σ, consider a subset L of Σ^*. Our decision procedure is one which takes a string $w \in \Sigma^*$ and decides whether or not $w \in L$. This essentially represents the subset as a yes/no decision made on a case-by-case basis. Keep in mind that we will attempt to frame all problems as questions of this form.

One comment: make sure you understand infinite sets. In particular, be careful to avoid fallacies of the following form: one infinite set is "bigger" than another infinite set, there is a "largest string" in an infinite set, etc.

Proof techniques Look over sections 1.2 - 1.4 of the course reader for a good survey of proof techniques. In particular, make sure you are very familiar with induction; one of the most common errors in problem sets was an incomplete induction proof.

Regular sets You should know the following:

- ϵ transitions
- Subset construction (NFA $\rightarrow$ DFA)
- State elimination (DFA $\rightarrow$ RE)
- ϵ-NFA construction from regular expression
- Closure properties
- Pumping lemma
- Decision properties: emptiness, membership, equivalence

Context free languages You should know the following:

- CFGs: derivations, sentential forms, and parse trees
- Ambiguity in CFG's
- Chomsky normal form for CFG's
- PDA accepting: final state, empty stack
- Determinism: DPDA's are not equivalent to NPDA's

[1]with help from Aris Gionis

- Closure properties
- Pumping lemma
- Decision properties: emptiness, membership

Turing machines You should know the following:

- Acceptance by halting and final state
- Nondeterminism
- Extensions (don't memorize details, just understand/be familiar with)
- Recursively enumerable: class of languages which can be described by Turing machines
- Recursive: class of languages which can be described by Turing machines which halt on all inputs

Reductions A restatement of the basic definition of reduction: Given a problem P which we want to prove is not C (where C is a level of complexity; e.g., recursively enumerable, recursive/decidable, etc.), take another problem K which we know is not C. Then assume that P is C, which implies that a C-level solution for P exists. Using that C-level solution for P, build a C-level solution for K. This is a reduction from K to P.

What this reduction shows is that P is at least as complex as K. Specifically, if K is not C, then P cannot be C either. The reduction described above is essentially a proof by contradiction. We start by assuming that P is C, and use that assumption to build a C-level solution for K. Since we choose a K which we know is not C, no such solution can exist, so by contradiction, our assumption must be false - P cannot be C.

The above explanation is somewhat confusing, so here is a more concise version: to reduce P_1 to P_2, build a solution to P_1 using a black-box solution to P_2. This "reduces" the problem of solving P_1 to that of solving P_2 - once we have a solution to P_2, we can solve P_1 using our reduction.

Make sure that you have the correct direction for your reduction; otherwise, it doesn't prove anything at all. Also keep in mind the distinction between running a Turing machine and examining its encoding. This distinction plays a crucial role in the L_{ne} proof provided in the course reader, which you should be familiar with.

Decidability A decidable problem is one which can be stated as a recursive language. That is, there exists a Turing machine which given any input $w \in \Sigma^*$, is guaranteed to eventually halt. Note that this definition assumes that the Turing machine accepts by final state rather than by halting - otherwise, it would accept all strings.

Generally speaking, we are concerned here with three distinct classes of languages: recursive, recursively enumerable (but not recursive), and not recursively enumerable. Look at the course reader and book and be familiar with examples of each.

You should know:

- Complement properties of RE and recursive languages
- Rice's theorem
- Examples of non-RE languages: L_d, L_e
- Examples of undecidable problems: L_u, L_{ne}, CFG ambiguity

Tractability Here, we make a distinction between problems which can be solved in polynomial time and those which cannot. $\mathcal{P}$ denotes the class of problems solvable in polynomial time by deterministic Turing machines, and $\mathcal{NP}$ denotes the class of problems solvable in polynomial time by nondeterministic Turing machines.

We are generally concerned with NP-complete problems - that is, problems which are in $\mathcal{NP}$ but are not provably in $\mathcal{P}$. This is a gross oversimplification, and you should see the course reader and/or the textbook for further explanation. For now, just note that the class of NP-complete problems is a subset of those in NP.

The diagram provided on the last page of the course reader summarizes reductions among NP-complete problems. You should read through this chapter and familiarize yourself with the different NP-complete problems, and understand the reductions which transform these problems back and forth.

One important thing to note is that when dealing with tractability problems, we must make sure that our reductions are polynomial-time as well. If we provide an exponential-time reduction from a one polynomial-time algorithm to another, we have not really shown anything - the end result is still an exponential-time algorithm.