Notes on Theory of Computation

William Schultz

February 9, 2022

Decidability

A language L is a set of strings $w \in \Sigma^*$, where Σ is some finite alphabet i.e. some finite set of symbols. We say that a language is *decidable* if there exists a Turing machine M such that M accepts every input $w \in L$ and always halts. We say that a language is *recognizable* if there is a Turing machine M that accepts every input $w \in L$ but it may reject or loop forever on inputs $w \notin L$. Note that for any Turing machine on a given input, there are three possible outcomes: accept, reject, or loop forever. A basic undecidable problem is the A_{TM} problem, which asks, given a Turing machine M and input w, does machine M accept on w? Note that the class of recognizable languages is more general than the class of undecidable languages. That is, any language that is decidable is recognizable, but the converse is not necessarily true.

P and NP

The class P is the set of languages for which there exists a Turing machine (i.e. an algorithm) that can decide P in polynomial time $O(n^k)$, for some positive constant k. The class NP is defined as the set of languages for which there exists a polynomial time *verifier*. Formally, a language L is in NP if there exists a Turing machine V (a verifier) such that, for any input $w \in L$ there exists a certificate c such that V accepts $\langle w, c \rangle$ and runs in polynomial time. Note that the size of the certificate can only be polynomial in the size of the input w, since the verifier machine only has a polynomial time run-time budget. At a high level, NP is the class of languages for which there exists an efficient way to verify solutions to a given problem instance. There may not be an efficient algorithm to decide the answer to a given instance though. It is not known for certain, however, whether there are (or aren't) polynomial time algorithms for solving problems in NP. This is the famous P vs. NP problem. It is believed that problems in NP do not have efficient (polynomial time) algorithms, but this is not formally proven.

NP-Completeness

There are some problems in the class NP that are the "hardest" problems in NP. We call these problems NP-complete. Formally, we say that a language A is NP-complete if $A \in NP$ and every language $B \in NP$ is polynomial time reducible to A. For two languages A and B, we say that A is polynomial time reducible to B if there is a polynomial time converter $R_{A \to B}$ that converts an input w_A to an input $w_B = R_{A \to B}(w_A)$ so that $w_B \in B \iff w_A \in A$. So, if a problem is NP-complete, it means that every problem in NP can be reduced to it i.e. we can take an input of any problem in NP, convert it to an input for the NP-complete problem in a way that preserves correctness. So, this means that if we solve one NP complete problem in an efficient (polynomial time) algorithm, then all NP problems are efficiently solvable.

The canonical NP-complete problem is the SAT problem i.e. checking whether a boolean formula is satisfiable. The Cook-Levin theorem shows that SAT is NP-complete. With this knowledge, we can prove other problems NP-complete. Since we know that any $B \in NP$ can be reduced to an NP-complete problem A, we can show that B is NP-complete by showing that A is reducible to B. That is, we establish that A is reducible to B and B is reducible to A, so the problems are "equivalently" hard. Note that there are some problems in NP, however, that are not NP-complete. Ladner's theorem establishes this i.e. it proves the existence of problems that are in NP but not in P and are not NP-complete. This class is called NP-intermediate. The construction used in this theorem is complicated, though, and not necessarily "natural". There are problems that are suspected to be in NP-intermediate i.e. they are in NP but have not been shown to be in P or be NP-complete e.g. integer factorization.

1 Upper and Lower Bounds

For any decision problem, we can establish both *upper bounds* and *lower bounds* on its complexity. Recall that a decision problem is formulated in terms of a language L, consisting of a set of strings. The decision problem for a given language L is to determine whether $w \in L$ for some given string w.

An upper bound makes a statement about the maximum hardness/complexity of the problem, and a lower bound makes a statement about the minimum easiness of the problem. Establishing an upper bound is typically much easier, since you only need to provide a concrete algorithm that solves the problem in some worst case running time (i.e. show there exists an algorithm). Establishing a lower bound is generally much harder, since you need to show that there exists no algorithm that can solve the problem more efficiently than a certain complexity class (i.e. show no algorithm exists).

For example, one of the best known algorithms for 3-SAT as of 2019 has a numerical upper bound of something around $O(1.307^n)$ [**?**]. There exists no known, general algorithm that can solve 3-SAT in polynomial time. But, it has also not been proven that such an algorithm doesn't exist. It seems that the best known lower bounds for SAT sit somewhere in the polynomial range of $n^{1.801}$, though this has some other caveats about "time-space tradeoffs" which I don't fully understand [**?**]. Proving that SAT, for example, had an exponential (or even super polynomial) lower bound would, of course, establish that $P \neq NP$, since SAT is NP-complete, and this would serve to separate P from NP. Of course, one could also prove $P = NP$ by simply giving a polynomial time algorithm for SAT i.e. by dropping the upper bound from exponential to polynomial. This might be "easier", in the sense that you would only have to find a single algorithm, but it may be "harder" in the sense that P=NP may not actually be true!