Theory of Formal Languages (Decidability Theory & Decidable Languages)

Lecture 11: Feb. 9, 2019

Prof. K.R. Chowdhary : *Professor of CS*

11.1 Decidable Theory

In one of the celebrated developments in mathematical logic, *Alongo Church*, building on the work of *Kurt Gödel*, showed that no algorithm can decide in general, whether the statements in number theory are true or false. Formally, we write $(\mathbb{N}, +, \times)$ to be the model whose universe is the natural numbers with the usual $+$ and $\times$ relations. For example, $6 \in \mathbb{N}$, and $6 = 1+5, 6 = 4+2, 6 = 2 \times 3, 6 = 1 \times 6$, etc, and the relation is equality. Alongo Church showed that, the theory "$Th(\mathbb{N}, +, \times)$", of this model is *undecidable*.

Before, we embark on this undecidable theory, let us examine one theory that is *decidable*. Let $(\mathbb{N}, +)$ be the same model, without '$\times$'relation. Hence, its theory is, $Th(\mathbb{N}, +)$. For example, the statement – "For all x, there exists some y, such that $x + x = y$",

$$\forall x \; \exists y[x + x = y]$$

is true and therefore, it is a member of $Th(\mathbb{N}, +)$. However, the statement, "there exists a y, such that for all x, there is $x + x = y$",

$$\exists y \; \forall x[x + x = y]$$

is false, and therefore not a member.

Theorem 11.1 *$Th(\mathbb{N}, +)$ is decidable.*

Proof Idea. We give an algorithm that can determine whether its input, a sentence ϕ in the language of $(\mathbb{N}, +)$ is true in the model. Let,

$$\phi = Q_1 x_1 Q_2 x_2 \ldots Q_l x_l[\psi] \tag{11.1}$$

where $Q_1, \ldots, Q_l$ each represent either $\exists$ or $\forall$ quantifiers, and ψ is a formula without quantifiers that has variables $x_1, \ldots, x_l$. For each i from 0 to l, define ϕ_i to be,

$$\phi_i = Q_{i+1} x_{i+1} Q_{i+2} x_{i+2} \ldots Q_l x_l[\psi]. \tag{11.2}$$

Thus, $\phi_0 = \phi$, and $\phi_l = \psi$.

The formula ϕ_i has i free variables (i.e., constants, or variables which are not bound). For $a_1, \ldots, a_i \in \mathbb{N}$, write $\phi_i(a_1, \ldots, a_i)$ to be the sentence obtained by substituting the constants $a_1, \ldots, a_i$ for the variables $x_1, \ldots, x_i$ in ϕ_i.

For each i from 0 to l, the algorithm constructs a finite automaton A_i that recognizes the collection of strings representing i-tuples of numbers that make ϕ_i true.

The algorithm begins by constructing A_l directly using a generalization. Then for each i from l down to 1, it uses A_i for construct A_{i-1}. Finally, once the algorithm has A_0, it tests whether A_0 accepts the empty string. If it does, ϕ is true, and algorithm accepts.

11.2 Decidable Languages

In this section we give some examples of languages that are decidable by algorithms. We focus on languages concerning automata and grammars. For example, we can present an algorithm that tests whether a string is a member of a context-free language (CFL). This is a parsing algorithm. These languages are interesting for several reasons. First, certain problems of this kind are related to applications. The problem of testing whether a CFL generates a string, is related to the problem of recognizing and compiling programs in a programming language. Second, certain other problems concerning automata and grammars are not decidable by algorithms. Starting with examples where decidability is possible helps us to appreciate the undecidable problems.

11.2.1 Decidability of Grammar Problems

The close relationship among programming language syntax, context-free grammars and language, parsing, and compiling is well known. Many of the problems concerning programming languages, parsing, or compiling, that one wish to solve are equivalent to undecidable or computationally intractable context-free grammar or language problems. Several such problems are,

1. *Emptiness-of-intersection problem.* It is a problem of determining if the intersection of the languages generated by two grammars is empty.

2. *Grammar-class-membership problems.* That is, problems of determining, for a grammar G and grammar class C, whether G is an element o f C.

3. *Language-class-membership problems.* That is, problems of determining, for a grammar G and class of languages $\mathcal{L}$, whether the language generated by G is an element of $\mathcal{L}$.

4. *Grammatical-similarity-relation problems.* That is, problems of determining, for a grammar G, binary relation ρ, and grammar class C, whether there exists a grammar $H \in C$ such that $G \ \rho \ H$.

11.2.2 Decidable Properties of Regular Languages

We begin with certain computational problems concerning finite automata. We give algorithms for testing,

a. whether a finite automaton accepts a string,

b. whether the language of a finite automaton is empty, and

c. whether two finite automata are equivalent.

Note that we chose to represent various computational problems by languages. Doing so is convenient because we have already set up terminology for dealing with languages. For example, the acceptance problem for DFAs is of testing whether a particular deterministic finite automaton M accepts a given string w. It can be expressed as a language, we call as A_{DFA}. This language contains the encoding of all DFAs together with strings that the DFAs accept.

Consider the language,

$$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}. \tag{11.3}$$

The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether $\langle B, w\rangle$ is a member of the language A_{DFA}. Similarly, we can formulate other computational problems in terms of testing membership in a language. Hence, showing that the language is decidable is the same as showing that the computational problem is decidable.

In the following theorem we show that A_{DFA} is decidable. Hence this theorem shows that the problem of testing whether a given finite automaton accepts a given string is decidable.

Theorem 11.2 *A_{DFA} is decidable.*

Proof: We simply need to present a TM M that decides A_{DFA}, where M is described as follows:

The TM M receives input $\langle B, w\rangle$, where B is description/representation of a DFA and w is a string. Then TM M simulate DFA B with w as input to B. If in this simulation, B accepts w, then M accepts, else M rejects. The A_{DFA} (acceptability of DFA) is language whose elements are all such pairs $\langle B, w\rangle$. □ ■

In the similar line, it can be easily shown that A_{NFA} is decidable language. For this, first it is necessary to convert NFA into DFA, then run the above algorithm, such that if decidability of $\langle N, w\rangle$ is to be decided, where N is and NFA, and whether w is acceptable by N or not. Turing machine T_1 translates the NFA N into an equivalent DFA D, and a Turing machine T_2 decides membership of $\langle D, w\rangle$ by simulating DFA D on T_2. If T_2 accepts w, then NFA accepts else not.

Similarly, we can determine whether a regular expression R generates a given string w. Accordingly, we may define a language $\langle R, w\rangle$ such that R is a regular expression, and it generates string w, as,

$$A_{REX} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}. \tag{11.4}$$

Following theorem proves that A_{REX} is decidable language.

Theorem 11.3 *A_{REX} is a decidable language.*

Proof: Let P be a TM that decides A_{REX}. Let $\langle R, w\rangle$ is input to P, where R is regular expression, and w is some string. Follow these steps to decide about A_{REX}.

1. Run P to convert R into equivalent NFA A,
2. Input $\langle A, w\rangle$ to a TM N,
3. Run TM N on input $\langle A, w\rangle$,
4. If N accepts, then P *accepts*, else *rejects*.

■

Now we turn to a different kind of problem concerning finite automata: *emptiness testing* for the language of a finite automaton. In the preceding three theorems we had to determine whether a finite automaton accepts a particular string. In the next proof we must determine whether a finite automaton accepts any strings at all. Let

$$E_{DFA} = \{\langle A\rangle \mid A \text{ is DFA and } L(A) = \phi\}. \tag{11.5}$$

Example 11.4 *Show that decidable languages are closed under the operations of* union *and* intersection.

Let there be two decidable languages L_1 and L_2. Since they are decidable, let L_1 has decider M_1 (Turing machine) and L_2 has decider M_2, such that $L(M_1) = L_1$ and $L(M_2) = L_2$. We can prove that the union of these languages, $L_3 = L_1 \cup L_2$, is decidable by showing how we can build a decider M_3 for L_3 using the deciders for L_1 and L_2.

We would build decider for L_3 using a two-tape Turing machine which accepts any string in $L_1 \cup L_2$ as follows. We call tape of M_1 as tape-1 and tape of M_2 as tape-2.

1. Input string w is on tape-1,
2. *copy* input string w from tape-1 to tape-2, so that both tapes now contain w,
3. *run* M_1 on tape-1; if M_1 accepts w, then M_3 accepts; if M_1 rejects w then go to step-4.
4. *run* M_2 on tape-2; if M_2 accepts w, then M_3 accepts; if M_2 rejects w then M_3 rejects."

Since both M_1 and M_2 are deciders (i.e., they always halt) then M_3 is also a decider, it will never loop. Since, every multitape TM has an equivalent single tape TM, so we can manipulate the definition above and make it a single tape TM.

The intersection is done similarly, but the machine M_3 is defined as follows:

1. Input string w is on tape-1,
2. *copy* input string w to tape 2,
3. *run* M_1 on tape-1; if M_1 accepts w, then go to step-4; if M_1 rejects w then M_3 reject.
4. *run* M_2 on tape-2: if M_2 accepts w, then M_3 accepts; if M_2 rejects w then M_3 reject."

Theorem 11.5 *E_{DFA} is decidable language.*

Proof: A DFA accepts some string iff reaching an accept state from the start state by traveling along the edges of a transition graph is possible. To test this condition, we can design a TM T that uses a marking algorithm.

Let T has input $\langle A \rangle$, and A is a DFA. The steps are:

1. Mark the start of A.
2. Mark every state that has a transition coming into it from any state that is already marked.
3. If no accept state is marked, *accept*; otherwise, *reject*.

■

The next theorem states that determining whether two DFAs recognize the same language is decidable. The language of all pairs $\langle A, B \rangle$, where A and B are representation of two DFAs, which recognize the identical language is expressed as,

$$EQ_{DFA} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are } DFAs \text{ and } L(A) = L(B)\}. \tag{11.6}$$

Theorem 11.6 *EQ_{DFA} is decidable language.*

Proof: To prove this, we construct a DFA D from DFAs A and B, where D accepts only those strings that are accepted either by A or B, but not both. Thus, if A and B recognize the same language, D will accept nothing[1]. The language D is

$$L(D) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)}) \cap L(B)) \tag{11.7}$$

Once we have constructed D we can use theorem 11.5 to test whether $L(D)$ is empty. If empty, then $L(A)$ and $L(B)$ are equal. Let there is a TM F as defined below:

1. Input $\langle A, B \rangle$ to F,
2. Construct DFA D from these two DFAs A, B as per equation 11.7,
3. Run TM T (as defined in theorem 11.5) on input D,
4. If T accepts, then *accept*, else *reject*.

■

[1]If two sets are P and Q, then $P \oplus Q = (\overline{P} \cap Q) \cup (P \cap \overline{Q})$. If P and Q are equal, then as per the below given formula, $P \oplus Q$ is null. However, if they are not equal, the $P \oplus Q$ is not null, but some finite set.

11.2.3 Decidable Properties of Context-Free Languages

Similar to the regular languages and finite automata, there exist properties of context-free languages and pushdown automata, and these can be algorithmically determined. For example, to determine whether a context-free grammar G generates a string w, and to find out whether the language of a given CFG G is empty. We can represent a language A_{CFG} consisting of tuples $\langle G, w\rangle$ as elements, where CFG G generates the string w, as

$$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates } w\}. \tag{11.8}$$

We have the following theorem about this.

Theorem 11.7 *A_{CFG} is decidable language.*

Proof: Using an approach where CFG G will generate all its possible strings, and ten it remains to find if w is generated is not an efficient method, because the computation may work indefinitely. On the other side, if G does not generate w, this algorithm would never halt. This idea for TM is as *recognizer* but not as *decider* of A_{CFG}.

To work as decider, the algorithm should try only finite number of derivations. So, we take help of CNF (Chomsky Normal Form), where a string of length $|w| = n$ can be derived in $2n - 1$ steps. Thus, checking only $2n - 1$ steps would be sufficient. Thus, we have following steps to decide A_{CFG}, using a TM S, which works as follows.

1. Input S is $\langle G, w\rangle$, where G is grammar and w is string,
2. Convert G to equivalent CNF,
3. List all the derivations with $2n - 1$ steps, for $n = 0$, have derivations of single step,
4. If any of these derivations generate w, *accept*, else *reject*.

■

A Program is a *finite description*, e.g., it describes a procedure that tries to factor the number, and outputs "no" if it finds prime factors, and "yes" if it does not.

There are two ways of looking at sets: 1. From *Computability* point of view, and 2. From *Compressibility* point of view. The *tractability* is a notion which describes that a set is "nearly computable". Assume that we have some function f and we can compute it, i.e., there is an algorithm, but only with some external information that we need from a set A. We write this as, $f \leq_T A$ (f is reducible to A). Assume that, in addition, A is computable, i.e., there is an algorithm to compute A. We can build together both programs into one program that directly computes f. In other words: if A is computable then all f that are computable in A are computable, too.

11.3 Computable sets

In computability theory, a set of natural numbers is called *recursive*, *computable* or *decidable* if there is an algorithm which terminates after a finite amount of time and correctly decides whether a given natural number belongs to the set.

A more general class of sets consists of recursively enumerable sets, also called semidecidable sets. For these sets, it is only required that there is an algorithm that correctly decides when a number is in the set; the algorithm may give no answer (but not the wrong answer) for numbers not in the set. A set which is not computable is called noncomputable or undecidable.

Definition 11.8 Computable set. *A subset S of the natural numbers $\mathbb{N}$ is called recursive if there exists a total computable function f such that $f(x) = 1$ if $x \in S$ and $f(x) = 0$ if $x \notin S$. In other words, the set S is recursive if and only if the* indicator function[2] *1_S is computable.* □

The following sets are computable:

1. Every finite or co-finite subset of the natural numbers is computable.
2. The above can be modified to include these special cases:
   (a) The empty set is computable.
   (b) The entire set of natural numbers is computable.
   (c) Each natural number (as defined in standard set theory) is computable; that is, the set of natural numbers less than a given natural number is computable.
3. The set of prime numbers is computable.
4. A recursive language that is recursive subset of a formal language, is computable.
5. The set of stings that correspond to well-formed programs is computable.

Note that any finite object can be encoded as a natural numbers (e.g., an algorithm, a program, a large prime number, a large composite number, code of a Turing Machine, etc). Hence, as per the definition of computable set, all these objects are computable.

Given sets $A, B \subseteq \mathbb{N}$, we say that "$A$ is computable in B", which we write as $A \leq_T B$, if there is a computable procedure that can tell whether an element is in A or not using B as an *oracle*. We call this also as "A is Turing reducible to B". That is, problem A is polynomial time reducible to problem B. We say that A is Turing equivalent to B, and we write $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$. The following three sets are Turing equivalent.

a. The set of pairs (set-of-generators, relations), of non-trivial groups;
b. The set of finite triangulations of simply connected manifolds;

[2] An indicator function or a characteristic function is defined on a set X that indicates membership of an element in a subset A of X. It has value 1 for all elements of A and the value 0 for all elements of X not in A. That is, $1_A(x) = 1$, if $x \in A$, else 0 for $x \notin A$.

c. The set of programs that halt.

What is a computable set? We fix some model of computation (typically Turing machines), and also fix a set $A \subseteq \mathbb{N}$, and the TM can answer Yes if a given integer $x \in A$, else the answer is No.

Example 11.9 *Show that infinite set T_∞ of Turing machines is computable.*

Turing machines have definite representation (see Universal TMs, in chapter **??**). For any x, if $x \in T_\infty$, we first choose format of x, if it is not as per the representation of TM, we reject it. Otherwise, we check for membership of x step by step in T_∞. The T_∞ is arranged lexicographically. In the search process of x, if $|x| > |T_i|$, where $T_i \in T_\infty$, then reject x, else keep on searching it in T_∞. When found, accept x.

□

There also exists non-computable sets. Following are the examples of non-computable sets:

- *The word problem:* Consider the groups that can be constructed with a finite set of generators and a finite set of relations between the generators. The set of pairs $\langle$*set-of-generators, relations*$\rangle$, of *non-trivial* groups is not computable. For example, the generators can be grammars' set, and relations can be acceptability / rejection relation between grammar and strings.

- *Simply connected manifolds:* The set of finite triangulations of *simply connected* manifolds is not computable.

- *The Halting problem:* The set of programs that halt, and do not run for ever, is not computable.