CS301: Computability and Complexity Theory (CC)

Lecture 4: Decidability

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

October 27, 2023

Table of contents

Section 1

Previously on CS301

The Definition of Algorithm

Intuitive notion of algorithms	equals	*Turing machine algorithms*

Figure: The Church–Turing thesis

Describing TMs

There are three possibilities:

1. The first is the **formal description** that spells out in full the TM's states, transition function, ...
2. Second is a higher level of description, called the **implementation description**, in which we use English prose to describe the way that the TM moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function
3. The third is the **high-level description**, wherein we use English prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head

Describing TMs

- From now on we use **high-level description**
- The input to a TM is always a string
- If we want to provide an object other than a string as input, we must first represent that object as a string
- Our notation for the encoding of an object O into its representation as a string is $< O >$
- If we have several objects $O_1, O_2, \ldots, O_k$ representing string is $< O_1, O_2, \ldots, O_k >$

Describing TMs

- A TM may be programmed to decode the representation so that it can be interpreted in the way we intend
- We describe Turing machine algorithms with an indented segment of text
- We break the algorithm into stages, each usually involving many individual steps of the TM's computation
- The first line of the algorithm describes the input to the machine. If the input description is simply w, the input is taken to be a string
- If the input description is the encoding of an object as in $< A >$, the TM first implicitly tests whether the input properly encodes an object of the desired form and rejects it if it doesn't

Example

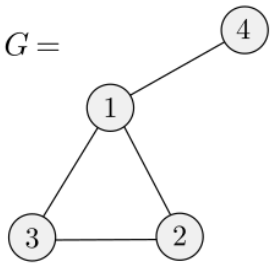
A graph is *connected* if every node can be reached from every other node by traveling along the edges of the graph. Let A be the language consisting of all strings representing undirected graphs that are connected.

$$A = \{< G > | G \text{ is a connected undirected graph}\}$$

M is a high-level description of a TM that decide A
$M =$ On input $< G >$, the encoding of a graph G:

1. Select the first node of G and mark it
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*

Example



$\langle G \rangle =$

`(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`

Figure: A graph G and its encoding $< G >$

Section 2

Context setup

Context setup

Corresponding to Sipser 4.1

Context setup

- Previously we introduced the TM as a model of a general purpose computer
- We have defined the notion of algorithm in terms of TM by means of the Church–Turing thesis
- We begin to investigate the power of algorithms to solve problems
- We demonstrate certain problems that can be solved algorithmically and others that cannot

Section 3

Decidable languages

Decidable languages

- We give some examples of languages that are decidable by algorithms
- We focus on languages concerning automata and grammars
- We focus on this languages because certain problems of this kind are related to specific applications
- While certain other problems concerning automata and grammars are not decidable by algorithms

Regular languages

- We start with certain computational problems concerning finite automata
- We chose to represent various computational problems by languages
- Reason being that we have already set up terminology for dealing with language
- Let us start with acceptance problem for DFAs

Regular languages

Bellow language contains the encodings of all DFAs together with strings that the DFAs accept

$$A_{DFA} = \{< B, w > | B \textit{ is a DFA that accepts input string } w\}$$

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
- Showing that the language is decidable is the same as showing that the computational problem is decidable

A_{DFA} is decidable

Theorem

A_{DFA} is decidable

A_{DFA} is decidable

Proof idea: All we need is to present a TM M that decides A_{DFA} $M =$ On input $< B, w >$, where B is a DFA and w is a string:

1. Simulate B on w
2. If simulation ends in accept state, *accept*. If ends in nonaccepting state, *reject*

A_{DFA} is decidable

Proof.

Let us examnine the input $< B, w >$. It is a representation of a DFA B together with a string w. One reasonable representation of B is simply a list of its five components: Q, Σ, δ, q_0 and F. When M receives its input, M first determines whether it properly represents a DFA B and a string w. If not, M rejects. Then M carries out the simulation directly. It keeps track of B's current state and B's current position in the input w by writing this information down on its tape. Initially, B's current state is q_0 and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When M finishes processing the last symbol of w, M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state $\square$

A_{NFA} is decidable

Bellow language contains the encodings of all NFAs together with strings that the NFAs accept

$$A_{NFA} = \{< B, w > | B \textit{ is a NFA that accepts input string } w\}$$

Theorem

A_{NFA} is decidable

A_{NFA} is decidable

Proof.

We present a TM N that decides A_{NFA} . We could design N to operate like M , simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: Have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.

$N =$ On input $< B, w >$, where B is a NFA and w is a string:

1. Convert NFA B to an equivalent DFA C
2. Run TM M from previous theorem on input $< C, w >$
3. If M accepts, *accept*; otherwise, *reject*

Running TM M in stage 2 means incorporating M into the design of N as a subprocedure □

A_{REX} is decidable

Bellow language contains the encodings of all REXs together with strings that the REXs generates

$$A_{REX} = \{< R, w > | R \textit{ is a regulaar expression that generates string } w\}$$

Theorem

A_{REX} is decidable

A_{REX} is decidable

Proof.

The following TM P decides A_{REX}

$P =$ On input $< R, w >$, where R is a regular expression and w is a string

1. Convert regular expression R to an equivalent NFA A
2. Run TM N on input $< A, w >$
3. If N accepts, *accept*; if N rejects, *reject*

□

Emptiness testing

- Now we turn to a different kind of problem concerning finite automata: *emptiness testing* for the language of a finite automaton
- We had to determine whether a finite automaton accepts a particular string

E_{DFA} is a decidable

The following language is the set of all languages that does not accepts any string

$$E_{DFA} = \{< A > | A \text{ is a DFA and } L(A) = \emptyset\}$$

Theorem

E_{DFA} is a decidable language

E_{DFA} is a decidable

Proof.

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

$T =$ on input $< A >$, where A is a DFA:

1. Mark the start state of A
2. Repeat until no new stages get marked:
3. Mark any state that has a transition coming into it from any state that is already marked
4. If no accept state is marked, *accept*; otherwise, *reject*

□

EQ_{DFA} is a decidable

Determining whether two DFAs recognize the same language is decidable. Let

$$EQ_{DFA} = \{< A, B > | A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

Theorem

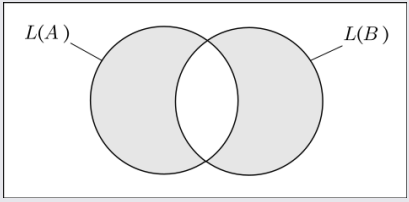
EQ_{DFA} is decidable language

EQ_{DFA} is a decidable

Proof.

To prove this theorem, we use previous theorem. We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = (L(A) \cup \overline{L(B)}) \cap (\overline{L(A)} \cap L(B))$$



EQ_{DFA} is a decidable

Proof.

Above expression is sometimes called the *symmetric difference* of $L(A)$ and $L(B)$. $\overline{L(A)}$ is the complement of $L(A)$. $\overline{L(A)}$ is the complement of $L(A)$. The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = LB(B)$. We can construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection. These constructions are algorithms that can be carried out by TMs. Once we have constructed C, we can use previous theorem to test whether $L(C)$ is empty. If it is empty then $L(A)$ and $L(B)$ must be equal.

$F =$ On input $< A, B >$ where A and B are DFSs:

1. Construct DFA C as described
2. run TM T (previous theorem) on input $< C >$
3. If T accepts, *accept*; otherwise, *reject*

□

A_{CFG} is a decidable

Next, we describe algorithms to determine whether a CFG generates a particular string and to determine whether the language of a CFG is empty. Let

$$A_{CFG} = \{< G, w > \mid G \textit{ is a CFG that generates string } w\}$$

Theorem

A_{CFG} is a decidable language

Proof idea: For CFG G and string w, we want to determine whether G generates w. One idea is to use G to go through all derivations to determine whether any is a derivation of w. This idea doesn't work, as infinitely many derivations may have to be tried. If G does not generate w, this algorithm would never halt. This idea gives a Turing machine that is a recognizer, but not a decider.

A_{CFG} is a decidable

To make this Turing machine into a decider, we need to ensure that the algorithm tries only finitely many derivation. We will show (seminar) that if G were in Chomsky normal form, any derivation of w has $2n-1$ steps, where n is the length of w. As result checking only derivations with $2n-1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using reviously studied conversion.

Proof.

The TM T for A_{CFG} is:
$S =$ On input $< G, w >$, where G is a CFG and w is a string

1. Convert G to an equivalent grammar in Chomsky normal form
2. List all derivations with $2n-1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step
3. If any of these derivations generate w, *accept*; otherwise, *reject*

□

CFG and PDA equivalence

Recall that we have given procedures for converting back and forth between CFGs and PDAs. Hence everything we say about the decidability of problems concerning CFGs applies equally well to PDAs.

E_{CFG} is a decidable

We can show that the problem of determining whether a CFG generates any strings at all is decidable. Let

$$E_{CFG} = \{< G > \mid G \text{ is a CFG and } L(G) = \emptyset\}$$

Theorem

E_{CFG} is a decidable language

E_{CFG} is a decidable

Proof idea: In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals. The algorithm does so by solving a more general problem. It determines for each variable whether that variable is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable

First, the algorithm marks all the terminal symbols in the grammar. Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols, all of which are already marked, the algorithm knows that this variable can be marked, too. The algorithm continues in this way until it cannot mark any additional variables.

E_{CFG} is a decidable

Proof.

R = On input $< G >$ where G is a CFG:

1. Mark all terminal symbols in G
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \rightarrow U_1 U_2 \ldots Uk_k$ and each symbol $U_1, \ldots, U_k$ has already been marked
4. If the start variable is not marked, *accept*; otherwise, *reject*

□

Context-free language are decidable

Next, we consider the problem of determining whether two context-free grammars generate the same language. Let

$$EQ_{CFG} = \{< G, H > \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$$

We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable. Because E_{CFG} also is decidable, you might think that we can use a similar strategy to prove that EQ_{CFG} is decidable. But something is wrong with this idea. The class of context-free languages is not closed under complementation or intersection. EQ_{CFG} is not decidable.