CS301: Computability and Complexity Theory (CC)

Lecture 7: Reducibility

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

November 17, 2023

Table of contents

Section 1

Previously on CS301

Some languages are not Turing-recognizable

Theorem

Some languages are not Turing-recognizable

This theorem has an important application to the theory of computation. It shows that some languages are not decidable or even Turing-recognizable, for the reason that there are uncountably many languages yet only countably many Turing machines. Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognized by any Turing machine. Such languages are not Turing-recognizable, as we state in the following theorem:

Reduction

- A **reduction** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem. Such reducibilities come up often in everyday life, even if we don't usually refer to them in this way.
- Reducibility plays an important role in classifying problems by decidability, and later in complexity theory as well
- When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A
- In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable
- Equivalently, if A is undecidable and reducible to B, B is undecidable. This is key to proving that various problems are undecidable
- In short, our method for proving that a problem is undecidable will be to show that some other problem already known to be undecidable reduces to it

Section 2

Context setup

Context setup

Corresponding to Sipser 5.2 & 5.3 & 6.3 & 6.4

Context setup

- We talk brieffly about **computation history** and **Post Correspondence Problem**
- We continue studying the primary method for proving that problems are computationally unsolvable: **reducibility**
- Next, we delve into deeper aspects of computability theory: **turing reducibility** and **descriptive complexity**
- Each topic is independet of the others

Section 3

Computation history

Computation history

The computation history method is an important technique for proving that A_{TM} is reducible to certain languages. This method is often useful when the problem to be shown undecidable involves testing for the existence of something. For example, this method is used to show the undecidability of Hilbert's tenth problem, testing for the existence of integral roots in a polynomial. The computation history for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input. It is a complete record of the computation of this machine.

Computation history

Definition

Let M be a Turing machine and w an input string. An *accepting computation history* for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. A *rejecting computation history* for M on w is defined similarly, except that C_l is a rejecting configuration.

Computation histories are finite sequences. If M doesn't halt on w, no accepting or rejecting computation history exists for M on w. Deterministic machines have at most one computation history on any given input. Nondeterministic machines may have many computation histories on a single input, corresponding to the various computation branches.

Computation history

Our first undecidability proof using the computation history method concerns a type of machine called a linear bounded automaton

Definition

A **linear bounded automaton** is a restricted type of TM wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is—in the same way that the head will not move off the left-hand end of an ordinary TM's tape.

A linear bounded automaton is a TM with a limited amount of memory. It can only solve problems requiring memory that can fit within the tape used for the input. Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor. Hence we say that for an input of length n, the amount of memory available is linear in n—thus the name of this model

A_{LBA} is decidable

Despite their memory constraint, linear bounded automata (LBAs) are quite powerful. For example, the deciders for A_{DFA} , A_{CFG} , E_{DFA} , and E_{CFG} all are LBAs. Let

$$A_{LBA} = \{< M, w > \mid M \textit{ is an LBA that accepts string } w\}$$

Theorem

A_{LBA} is decidable

A_{LBA} is decidable

Lemma

Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n

Proof.

Recall that a configuration of M is like a snapshot in the middle of its computation. A configuration consists of the state of the control, position of the head, and contents of the tape. Here, M has q states. The length of its tape is n, so the head can be in one of n positions, and g^n possible strings of tape symbols appear on the tape. The product of these three quantities is the total number of different configurations of M with a tape of length n. $\square$

A_{LBA} is decidable

Proof idea: In order to decide whether LBA M accepts input w, we simulate M on w. During the course of the simulation, if M halts and accepts or rejects, we accept or reject accordingly. The difficulty occurs if M loops on w. We need to be able to detect looping so that we can halt and reject.
The idea for detecting when M is looping is that as M computes on w, it goes from configuration to configuration. If M ever repeats a configuration, it would go on to repeat this configuration over and over again and thus be in a loop. Because M is an LBA, the amount of tape available to it is limited. By previous lemma, M can be in only a limited number of configurations on this amount of tape. Therefore, only a limited amount of time is available to M before it will enter some configuration that it has previously entered. Detecting that M is looping is possible by simulating M for the number of steps given by previous lemma. If M has not halted by then, it must be looping.

A_{LBA} is decidable

Theorem

A_{LBA} is decidable

Proof.

The algorithm that decides A_{LBA} is as follows:

$L =$ On ipunt $< M, w >$, where M is a LBA and w is a string:

1. Simulate M on w for qng^n steps or until it halts
2. If M has haled; *accept* if it has accepted or *reject* if it has rejected. If has not halted, *rejext*

If M on w has not halted within qng^n steps, it must be repeating a configuration according to previous lemma and therefore looping. That is why our algorithm rejects in this instance. □

Section 4

Post Correspondence Problem

PCP is undecidable

Next, we show that the phenomenon of undecidability is not confined to problems concerning automata. We study a problem that can easily be described as a puzzle and it is called **Post Correspondence Problem** (PCP).
We begin with a collection of dominos, each containing two strings, one on each side. An individual domino looks like

$$\left[\frac{a}{b}\right]$$

and a collection of dominos looks like

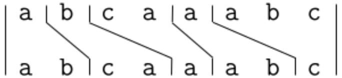
$$\left\{\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right]\right\}$$

PCP is undecidable

The task is to make a list of these dominos (repetitions permitted) so that the string we get by reading off the symbols on the top is the same as the string of symbols on the bottom. This list is called a *match*. For example, the following list is a match for this puzzle

$$\left[\frac{a}{ab}\right], \left[\frac{b}{ca}\right], \left[\frac{ca}{a}\right], \left[\frac{a}{ab}\right], \left[\frac{abc}{c}\right]$$

Reading off the top string we get *abcaaabc*, which is the same as reading off the bottom. We can also depict this match by deforming the dominos so that the corresponding symbols from top and bottom line up



PCP is undecidable

For some collections of dominos, finding a match may not be possible. For example, the collection

$$\left\{ \left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{ba}\right] \right\}$$

cannot contain a match because every top string is longer than the corresponding bottom string.

PCP is undecidable

We state the problem precisely and then express it as a language. An instance of the PCP is a collection P of dominos

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

and a match is a sequence $i_1, i_2, \ldots, i_l$, where $t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_1} \ldots t_{i_l}$. The problem is to determine if P has a match. Let

$$PCP = \{< P > | P \text{ is an instance of the Post Correspondence Problem with a match}\}$$

Theorem

PCP is undecidable

PCP is undecidable

Proof idea: Conceptually this proof is simple, though it involves many details. The main technique is reduction from A_{TM} via accepting computation histories. We show that from any TM M and input w, we can construct an instance P where a match is an accepting computation history for M on w. If we could determine whether the instance has a match, we would be able to determine whether M accepts w.
How can we construct P so that a match is an accepting computation history for M on w?
We choose the dominos in P so that making a match forces a simulation of M to occur. In the match, each domino links a position or positions in one configuration with the corresponding one(s) in the next configuration.

Proof.

homework :) $\square$

Section 5

Mapping Reducibility

Mapping Reducibility

- We have shown how to use the reducibility technique to prove that various problems are undecidable.
- Next, we formalize the notion of reducibility. Doing so allows us to use reducibility in more refined ways, such as for proving that certain languages are not Turing-recognizable and for applications in complexity theory.
- We concentrate on a simple type of reducibility called **mapping reducibility**
- Being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B
- If we have such a conversion function, called a *reduction*, we can solve A with a solver for B
- The reason is that any instance of A can be solved by first using the reduction to convert it to an instance of B and then applying the solver for B

Computable functions

A TM computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a **computable function** if some TM M, on every input w, halts with just $f(w)$ on its tape.

Example: All usual, arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $< m, n >$ and returns $m + n$, the sum of m and n.

Mapping Reducibility

Next, we define mapping reducibility. As usual, we represent computational problems by languages.

Definition

Language A is mapping reducible to language B, written $A \leq_m B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w

$$w \in A \iff f(w) \in B$$

the function f is called the **reduction** from A to B.

Mapping Reducibility

A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. The term mapping reduction comes from the function or mapping that provides the means of doing the reduction.

If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem. We capture this idea in next theorem

Theorem

If $A \leq_m B$ and B is decidable, then A is decidable.

Mapping Reducibility

Proof.

We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

```
$N$ = On input $w$:
  1. Compute $f(w)$
  2. Run $M$ on input $f(w)$ and output whatever $M$ outputs
```

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Therefore, N works as desired. □

Corollary

If $A \leq_m B$ and B is undecidable, then B is undecidable.

Example I

We used a reduction from A_{TM} to prove that $HALT_{TM}$ is undecidable. This reduction showed how a decider for $HALT_{TM}$ could be used to give a decider for A_{TM}. We can demonstrate a mapping reducibility from A_{TM} to $HALT_{TM}$ as follows. To do so, we must present a computable function f that takes input on the form $< M, w >$ and returns putput $< M', w' >$, where

$$< M, w > \in A_{TM} \iff < M', w' > \in HALT_{TM}$$

Example I

The following machine F computes a reduction f

```
$F$ = On iput $< M, w >$:
  1. Construct the following machine $M'$
     M' = On input $x$:
       1. Run $M$ on $x$
       2. If $M$ accepts, accept
       3. If $M$ rejects, enter a loop
  2. Output $< M', w >$
```

A minor issue arises here concerning improperly formed input strings. If TM F determines that its input is not of the correct form as specified in the input line "On input $< M, w >$:" and hence that the input is not in A_{TM}, the TM outputs a string not in $HALT_{TM}$. Any string not in HALT TM will do. In general, when we describe a TM that computes a reduction from A to B, improperly formed inputs are assumed to map to strings outside of B.

Example II

A mapping reduction from E_{TM} to EQ_{TM} is function f that maps the input $< M >$ to the output $< M, M_1 >$, where M_1 is the machine that rejects all inputs.

The proof of theorem showing that E_{TM} is undecidable illustrates the difference between the formal notion of mapping reducibility that we have defined in this section and the informal notion of reducibility that we used earlier. The proof shows that E_{TM} is undecidable by reducing A_{TM} to it. Let's see whether we can convert this reduction to a mapping reduction. From the original reduction, we may easily construct a function f that takes input $< M, w >$ and produces output M_1, where M_1 is the TM described in that proof. But M accepts w iff $L(M_1)$ is not empty so f is a mapping reduction from A_{TM} to E_{TM}. It still shows that E_{TM} is undecidable because decidability is not affected by complementation, but it doesn't give a mapping reduction from A_{TM} to E_{TM}. In fact, no such reduction exists.

Section 6

Turing Reducibility

Turing Reducibility

- We already introduced the reducibility as a way of using a solution to one problem to solve other problems
- Thus, if A is reducible to B, and we find a solution to B, we can obtain a solution to A
- Subsequently, we described mapping reducibility, a specific form of reducibility
- But does mapping reducibility capture our intuitive concept of reducibility in a more general way?
- Let us consider A_{TM} and $\overline{A_{TM}}$
- Intuitively, they are reducible to one another because a solution to either could be used to solve the other by simply reversing the answer
- However, we know that $\overline{A_{TM}}$ is not mapping reducible to A_{TM} because A_{TM} is Turing-recognizable but $\overline{A_{TM}}$ is not
- We present a very general form of reducibility called **Turing reducibility**

Oracle TMs

Definition

An **oracle** for a language B is an external device that is capable of reporting whether any string w is a member of B. An **oracle TM** is a modified TM that has the additional capability of querying an oracle. We write M^B to describe an oracle TM that has an oracle for language B.

We aren't concerned with the way the oracle determines its responses. We use the term oracle to connote a magical ability and consider oracles for languagesthat aren't decidable by ordinary algorithms.

Example I

Consider an oracle for A_{TM}. An oracle TM with an oracle for A_{TM} can decide more languages than an ordinary TM can. Such a machine can (obviously) decide A_{TM} itself, by querying the oracle about the input. It can also decide E_{TM}, the emptiness testing problem for TMs with the following procedure called $T^{A_{TM}}$.

```
$T^{A_{TM}}$ = On input $< M >$, where $M$ is a TM:
  1. Construct the following TM $N$
     $N$ = On any input:
       1. Run $M$ in parallel on all strings in $\Sigma^*$
       2. If $M$ accepts any of this strings, accept
  2. Query the oracle to determine whether $< N, 0 > \in A_{TM}$
  3. If the oracle answer No, accept; if Yes, reject
```

Turing reducibility

If M's language isn't empty, N will accept every input and, in particular, input 0. Hence the oracle will answer YES, and $T^{A_{TM}}$ will reject. Conversely, if M's language is empty, $T^{A_{TM}}$ will accept. Thus $T^{A_{TM}}$ decides E_{TM}. We say that E_{TM} is **decidable relative** to A_{TM}. That brings us to the definition of Turing reducibility:

Definition

Language A is Turing reducible to language B, written $A \leq_T B$, if A is decidable relative to B

Turing reducibility

Theorem

If $A \leq_T B$ and B is decidable, then A is decidable

Proof.

If B is decidable, then we may replace the oracle for B by an actual procedure that decides B. Thus, we may replace the oracle Turing machine that decides A by an ordinary Turing machine that decides A. $\square$

Turing reducibility

Turing reducibility is a generalization of mapping reducibility. If $A \leq_M B$, then $A \leq_T B$ because the mapping reduction may be used to give an oracle Turing machine that decides A relative to B.

An oracle Turing machine with an oracle for A_{TM} is very powerful. It can solve many problems that are not solvable by ordinary Turing machines. But even such a powerful machine cannot decide all languages

Section 7

A Definition of Information

Definition of Information

- The concepts algorithm and information are fundamental in computer science. While the Church–Turing thesis gives a universally applicable definition of algorithm, no equally comprehensive definition of information is known
- Instead of a single, universal definition of information, several definitions are used, depending upon the application. Next we discuss one way of defining information, using computability theory

Definition of Information

We start with an example. Consider the information content of the following two binary sequences

$$
\begin{aligned}
A &= 0101010101010101010101010101010101010101 \\
B &= 1110010110100011101010000111010011010111
\end{aligned}
$$

Intuitively, sequence A contains little information because it is merely a repetition of the pattern 01 twenty times. In contrast, sequence B appears to contain more information.

Definition of Information

- We can use previous simple example to illustrate the idea behind the definition of information that we present
- We define the quantity of information contained in an object to be the size of that object's smallest representation or description
- By a description of an object, we mean a precise and unambiguous characterization of the object so that we may recreate it from the description alone
- Sequence A contains little information because it has a small description, whereas sequence B apparently contains more information because it seems to have no concise description

Definition of Information

- Why do we consider only the shortest description when determining an object's quantity of information?
- We may always describe an object, such as a string, by placing a copy of the object directly into the description
- We can obviously describe the preceding string B with a table that is 40 bits long containing a copy of B. This type of description is never shorter than the object itself and doesn't tell us anything about its information quantity
- However, a description that is significantly shorter than the object implies that the information contained within it can be compressed into a small volume, and so the amount of information can't be very large
- Hence **the size of the shortest description determines the amount of information**
- Next, we formalize this intuitive idea. First, we restrict our attention to objects that are binary strings. Second, we consider only descriptions that are themselves binary strings. By imposing this requirement, we may easily compare the length of the object with the length of its description

Minimal length

- Many types of description language can be used to define information. Selecting which language to use affects the characteristics of the definition. Our description language is based on algorithms
- One way to use algorithms to describe strings is to construct a TM that prints out the string when it is started on a blank tape and then represent that TM itself as a string
- A drawback to this approach is that a TM cannot represent a table of information concisely with its transition function. To represent a string of n bits, you might use n states and n rows in the transition function table
- Instead we describe a binary string x with a Turing machine M and a binary input w to M. The length of the description is the combined length of representing M and w
- We write this description with our usual notation for encoding several objects into a single binary string $< M, w >$
- We define the string $< M, w >$ to be $< M >, w$, where we simply concatenate the binary string w onto the end of the binary encoding of M. The encoding $< M >$ of M may be done in any standard way

Descriptive complexity

Definition

Let x be a binary string. The minimal description of x, written $d(x)$, is the shortest string $< M, w >$ where TM M on input w halts with x on its tape. If several such strings exist, select the lexicographically first among them. The **descriptive complexity** of x, written $K(x)$, is

$$K(x) = |d(x)|$$

In other words, $K(x)$ is the length of the minimal description of x. The definition of $K(x)$ is intended to capture our intuition for the amount of information in the string x. Descriptive complexity is also known as **Kolmogorov complexity**.

Next we establish some simple results about descriptive complexity.