List of potential questions and answers for INFOF408 Computability and Complexity (2019/2020)

December 27, 2019

Contents

Remarks

Sonia to future students : These answers are not necessarily there to provide a better explanation of the course (maybe later when I myself would have had time to understand better). It is simply a collection of the answers to the questions to avoid looking everywhere to find them. If it was easily found in the course reference book, I added what it says. If not, I searched in other summaries, in my own course notes or on the internet.

1 Chapter 3: The Church-Turing Thesis

1.1 Explain how decision problems can be formalized by the notion of language of finite words. Develop an example of a language that formalizes a problem

cf : [1]

First, we need to define these concepts :

Decision problems : the set of inputs for which the answer is yes. In other words, an instance of a decision problem is whether or not an input A is part of the solution package. It is important to distinguish between an instance and a problem (the former refers to a specific input whereas the second takes as a reference all possible inputs).

Language of finite words: The language of an M machine is defined as : L(M) = A. If A is the set of all the words (strings) that the M machine accepts. We can say that:

- M accepts A
- M recognizes A

add examples

1.2 Define the notion of configuration of a Turing machine. Define the notion of computation of a deterministic Turing machine M on a word w.

cf Sipser, pp. 140-141

A *configuration* of a TM consists of three items — the current state, tape contents and head location. For a state q and two strings u and v over the tape alphabet Γ we could write a configuration uqv where : uv is the current content of the tape, q is the current state and the head location is the first symbol of v.

$1011q_701111$ represents a configuration when the tape is 101101111, the current state is q_7 and the head is on the second 0 (given by the position of q_7 in $1011q_701111$.)

1.3 Are nondeterministic Turing machines more powerful than deterministic Turing machines as deciders ?Justify your answer by proving that every nondeterministic Turing machine has an equivalent deterministic Turing machine

cf Sipser, pp. 150

A nondeterministic Turing machine is defined in the expected way. At any point in a computation the machine may proceed according to several possibilities. The transition function for a nondeterministic Turing machine has the form add equation

The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine.

add schema

If some branch of the computation leads to the accept state, the machine accepts its input.

Now we show that nondeterminism does not affect the power of the Turing machine model.

THEOREM 3.16 *Every nondeterministic Turing machine has an equivalent deterministic Turing machine.*

PROOF IDEA We can simulate any nondeterministic TM N with a deterministic TM D. The idea behind the simulation is to have D try all possible branches of N's nondeterministic computation. If D ever finds the accept state on one of these branches, D accepts. Otherwise, D's simulation will not terminate. We view N's computation on an input w as a tree. Each branch of the tree represents one of the branches of the nondeterminism. Each node of the tree is a configuration of N. The root of the tree is the start configuration. The TM D searches this tree for an accepting configuration. Conducting this search carefully is crucial lest D fail to visit the entire tree. A tempting, though bad, idea is to have D explore the tree by using depth-first search. The depth-first search strategy goes all the way down one branch before backing up to explore other branches. If D were to explore the tree in this manner, D could go forever down one infinite branch and miss an accepting configuration on some other branch. Hence we design D to explore the tree by using breadth first search instead. This strategy explores all branches to the same depth before going on to explore any branch to the next depth. This method guarantees that D will visit every node in the tree until it encounters an accepting configuration.

PROOF The simulating deterministic TM D has three tapes. By Theorem 3.13 this arrangement is equivalent to having a single tape. The machine D uses its three tapes in a particular way, as illustrated in the following figure. Tape 1 always contains the input string and is never altered. Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation. Tape 3 keeps track of D's location in N's nondeterministic computation tree.

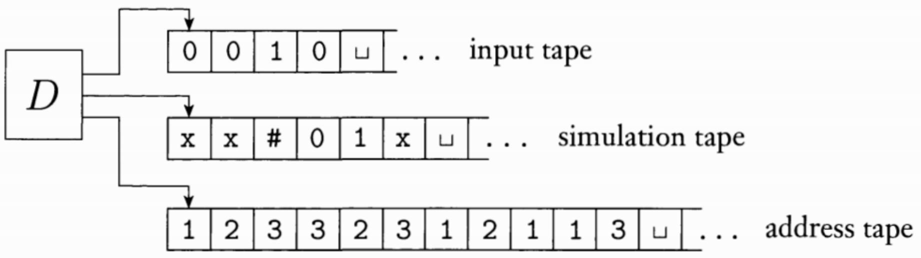
D
0 0 1 0 ⊔ . . . input tape
x x # 0 1 x ⊔ . . . simulation tape
1 2 3 3 2 3 1 2 1 1 3 ⊔ . . . address tape

FIGURE 3.17
Deterministic TM D simulating nondeterministic TM N

Let's first consider the data representation on tape 3. Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function. To every node in the tree we assign an address that is a string over the alphabet $\sum_b = \{1, 2, ..., b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's lst child. Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case the address is invalid and doesn't correspond to any node. Tape 3 contains a string over . It represents the branch of N's computation from the root to the node addressed by that string, unless the address is invalid. The empty string is the address of the root of the tree. Now we are ready to describe D.

1 Initially tape 1 contains the input w, and tapes 2 and 3 are empty.

2 Copy tape 1 to tape 2.

3 Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, accept the input.

4 Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of N's computation by going to stage 2.

1.4 Given a Turing machine M and a word w, give the computation of M on w. Does M accept w

cf Sipser, pp. 141

The start configuration of M on input w is the configuration q_0 w, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape. In an accepting configuration the state of the configuration is q_{accept}. In a rejecting configuration the state of the configuration is q_{reject}. Accepting and rejecting configurations are halting configurations and do not yield further configurations. Because the machine is defined to halt when in the states q_{accept} and q_{reject}, we equivalently could have defined the transition function to have the more complicated form

add equation

where Q' is Q without q_{accept} and q_{reject}.

A Turing machine M **accepts** input w if a sequence of configurations $C_1, C_2, ..., C_k$ exists, where

1 C_1 is the start configuration of M on input w

2 each C_i yields C_{i+1}, and

3 C_k is an accepting configuration

1.5 Define when a Turing machine recognizes a language L and when it decides a language L. What is the fundamental difference between those two notions ? Why do we use deciders to formalize the intuitive notion of algorithm

cf Sipser, pp. 142

The collection of strings that M accepts is the language of M, or the language recognized by M, denoted $L(M)$.

Definition

Call a language Turing-recognizable if some Turing machine recognizes it.

When we start a Turing machine on an input, three outcomes are possible. The machine may accept, reject, or loop. By loop we mean that the machine simply does not halt. Looping may entail any simple or complex behaviour that never leads to a halting state. A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult.

For this reason we prefer Turing machines that halt on all inputs; such machines never loop. These machines are called deciders because they always make a decision to accept or reject. A decider that recognizes some language also is said to decide that language.

Definition

Call a language Turing-decidable or simply decidable if some Turing machine decides it.

1.6 Define the notion of enumerator for a language. Prove that a language is Turing-recognizable if and only if it has an enumerator

cf Sipser, pp. 152

Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer. The following figure depicts a schematic of this model.

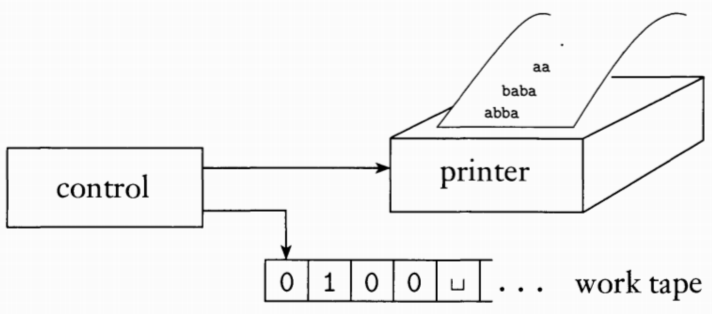

FIGURE 3.20
Schematic of an enumerator

An enumerator E starts with a blank input tape. If the enumerator doesn't halt, it may print an infinite list of strings. The language enumerated by E is the collection of all the strings that it eventually prints out. Moreover, E may generate the strings of the language in any order, possibly with repetitions. Now we are ready to develop the connection between enumerators and Turing-recognizable languages.

THEOREM 3.2 *A language is Turing-recognizable if and only if some enumerator enumerates it.*

PROOF First we show that if we have an enumerator E that enumerates a language A, a TM M recognizes A. The TM M works in the following way. $M =$ "On input w:

1 Run E. Every time that E outputs a string, compare it with w.

2 If w ever appears in the output of E, accept"

Clearly, M accepts those strings that appear on E's list. Now we do the other direction. If TM M recognizes a language A, we can construct the following enumerator E for A. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in $\sum^*$.

$E =$ "Ignore the input.

1 Repeat the following for $i = 1, 2, 3, \ldots$.

2 Run M for i steps on each input. $s_1, s_2, \ldots, s_i$

3 If any computations accept, print out the corresponding s_j."

If M accepts a particular string s, eventually it will appear on the list generated by E. In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1. This procedure gives the effect of running M in parallel on all possible input strings.

1.7 Define the notion of multitape Turing machine. Prove that every multitape Turing machine has an equivalent single tape Turing machine

cf Sipser, pp. 148

A multitape Turing machine is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially the input appears on tape 1, and the others start out blank. The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously. Formally, it is

add equation

where k is the number of tapes. The expression

add equation

means that, if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes symbols b_1 through b_k, and directs each head to move left or right, or to stay put, as specified. Multitape Turing machines appear to be more powerful than ordinary Turing machines, but we can show that they are equivalent in power. Recall that two machines are equivalent if they recognize the same language.

THEOREM 3.13 *Every multitape Turing machine has an equivalent single-tape Turing machine.*

PROOF We show how to convert a multitape TM M to an equivalent singletape TM S. The key idea is to show how to simulate M with S. Say that M has k tapes. Then S simulates the effect of k tapes by storing their information on its single tape. It uses the

new symbol # as a delimiter to separate the contents of the different tapes. In addition to the contents of these tapes, S must keep track of the locations of the heads. It does so by writing a tape symbol with a dot above it to mark the place where the head on that tape would be. Think of these as "virtual" tapes and heads. As before, the "dotted" tape symbols are simply new symbols that have been added to the tape alphabet.
The following figure illustrates how one tape can be used to represent three tapes.

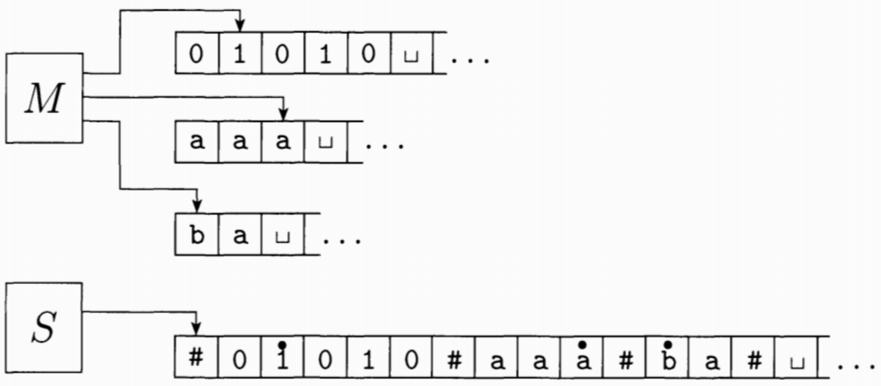

FIGURE 3.14
Representing three tapes with one

$S =$ "On input $w = w_1, ..., w_n$:

1 Firsts S puts its tape into the format that represent all k tapes of M. The formatted tape contains

Add equation

2 To simulate a single move, S scans its tape t from the first #, which marks the left-hand end, to the $(k+1)$st #, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M's transition function dictates.

3 If at any point S moves one of the virtual heads to the right onto a #, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost #, one unit to the right. Then it continues the simulation as before."

1.8 What is the Church-Turing thesis ? Give a list of arguments in favor of this thesis ? Why is it not a theorem

cf Sipser, pp. 155

HILBERT'S PROBLEMS

In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris. In his lecture, he identified twenty-three mathematical problems and posed them as a challenge for the coming century. The tenth problem on his list concerned algorithms. Before describing that problem, let's briefly discuss polynomials. A polynomial is a sum of terms, where each term is a product of certain variables and a constant called a coefficient. For example,

add equation

is a term with coefficient 6, and

add equation

is a polynomial with four terms over the variables x, y and z For this discussion, we consider only coefficients that are integers. A root of a polynomial is an assignment of values to its variables so that the value of the polynomial is 0. This polynomial has a root at $x = 5$, $y = 3$, and $z = 0$. This root is an integral root because all the variables are assigned integer values. Some polynomials have an integral root and some do not.
Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root. He did not use the term algorithm but rather

> *"a process according to which it can be determined by a finite number of operations."*

Interestingly, in the way he phrased this problem, Hilbert explicitly asked that an algorithm be "devised." Thus he apparently assumed that such an algorithm must exist, someone need only find it.
As we now know, no algorithm exists for this task; it is algorithmically unsolvable. For mathematicians of that period to come to this conclusion with their intuitive concept of algorithm would have been virtually impossible. The intuitive concept may have been adequate for giving algorithms for certain tasks, but it was useless for showing that no algorithm exists for a particular task. Proving that an algorithm does not exist requires having a clear definition of algorithm. Progress on the tenth problem had to wait for that definition.

The **definition** came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the λ-calculus to define algorithms. Turing did it with his "machines." These two definitions were shown to be equivalent. This connection between the informal notion of algorithm and the precise definition has come to be called the Church-Turing thesis. The Church-Turing thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem. In 1970, Yuri Matijasevic, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots.

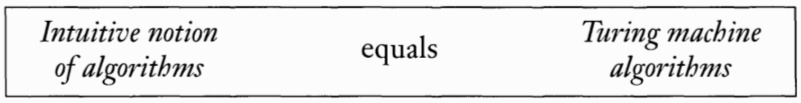

FIGURE 3.22
The Church–Turing Thesis

Need more info

1.9 Why don't we use the notions of finite automata or context-free grammars to formalize algorithms

cf Sipser, pp. 138

First, here are the differences between a Turing machine and a finite automata :

1. A Turing machine can both write on the tape and read from it.
2. The read-write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting take effect immediately.

Not in the book, need research

2 Chapter 4: Decidability

The notion of *countable* is used several times here, I invite you to read the book on pages 174 to 178 to understand this notion. The answers will be more readable after that

2.1 Why is the set of Turing machines countable

cf Sipser, pp. 178

PROOF To show that the set of all Turing machines is countable we first observe that the set of all strings $\sum^*$ is countable, for any alphabet $\sum$. With only finitely many strings of each length, we may form a list of $\sum^*$ by writing down all strings of length 0, length 1, length 2, and so on. The set of all Turing machines is countable because each Turing machine M has an encoding into a string $< M >$. If we simply omit those strings that are not legal encoding of Turing machines, we can obtain a list of all Turing machines.

2.2 Prove that the set of subsets of an infinite countable set is not countable

Maybe in page 175, maybe in the TP?

2.3 Prove that the language $L_0 = \{w_i | w_i \notin L(M_i)$ is not Turing recognizable

Every decidable language is Turing-recognizable.

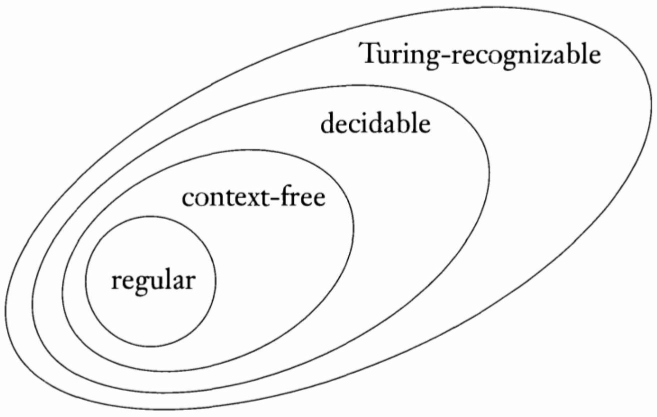

FIGURE 4.10
The relationship among classes of languages

Don't know... Does I refer to automaton or turing machine?

2.4 Prove that the language $A_{TM} = \{(M, w) | M$ is a TM and M accepts $w\}$ is undecidable

cf Sipser, pp. 174

The problem of determining whether a Turing machine accepts a given input string. We call it A_{TM} by analogy with A_{DFA} and A_{CFG}.

$$A_{TM} = \{\langle M, w\rangle |\ M \text{ is a TM and } M \text{ accepts } w\}.$$

THEOREM 4.11 A_{TM} *is undecidable.*

Before we get to the proof, let's first observe that A_{TM} is Turing-recognizable. Thus this theorem shows that recognizers are more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize.

The following Turing machine U recognizes A_{TM}.

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate W on input w.
2. If M ever enters its accept state, accept; if M ever enters its reject state, reject."

Note that this machine loops on input $\langle M, w\rangle$ if M loops on w, which is why this machine does not decide A_{TM}. If the algorithm had some way to determine that M was not halting on w, it could reject. Hence A_{TM} is sometimes called the halting problem. As we demonstrate, an algorithm has no way to make this determination.
The Turing machine U is interesting in its own right. It is an example of the universal Turing machine first proposed by Turing. This machine is called universal because it is capable of simulating any other Turing machine from the description of that machine. The universal Turing machine played an important early role in stimulating the development of stored-program computers.

THE DIAGONALIZATION METHOD
The proof of the undecidability of the halting problem uses a technique called diagonalization, discovered by mathematician Georg Cantor in 1873. Cantor was concerned with the problem of measuring the sizes of infinite sets. If we have two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size? For finite sets, of course, answering these questions is easy. We simply count the elements in a finite set, and the resulting number is its size. But, if we try to count the elements of an infinite set, we will never finish! So we can't use the counting method to determine the relative sizes of infinite sets.

For example, take the set of even integers and the set of all strings over 0,1. Both sets are infinite and thus larger than any finite set, but is one of the two larger than the other? How can we compare their relative size? Cantor proposed a rather nice solution to this problem. He observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set. This method compares the sizes without resorting to counting. We can extend this idea to infinite sets. Let's see what it means more precisely.

Definition

Assume that we have sets A and B and a function f from A to B. Say that f is one-to-one if it never maps two different elements to the same place-that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that f is onto if it hits every element of B—that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that A and B are the same size if there is a one-to-one, onto function $f : A \longrightarrow B$. A function that is both one-to-one and onto is called a correspondence. In a correspondence every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B.

There is a lot of page explaining theorem before atteining the ATM one, the theorem is from page 179 to page 181...

2.5 Explain why proving A_{TM} undecidable establishes that R $\neq$ RE

R and RE are already defined in question 1.5

- R = recursive language or **Turing-recognizable**
- RE = recursively enumerable language or **decidable**

So we can rephrase the question as "Why proving A_{TM} undecidable establishes that Turing-recognizable $\neq$ decidable ?"

Tired, will continue after

2.6 Prove that a language A $\in$ R iff A is Turing-recognizable and co-Turing-recognizable

Page 181

3 Chapter 5: Reducibility

3.1 Explain the technique of "the reduction" with the proof that $\textbf{HALT}_{\textbf{TM}} = \{(M, w) | M$ is a TM and M halts on $w\}$ is undecidable

cf Sipser, pp. 188 - 189

Let's consider a related problem, $\mathrm{HALT}_{\mathrm{TM}}$, the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input. We use the undecidability of A_{TM} to prove the undecidability of $\mathrm{HALT}_{\mathrm{TM}}$ by reducing A_{TM} to $\mathrm{HALT}_{\mathrm{TM}}$.

PROOF IDEA This proof is by contradiction. We assume that $\mathrm{HALT}_{\mathrm{TM}}$ is decidable and use that assumption to show that A_{TM} is decidable, contradicting Theorem 4.11 (cf section 2.4). The key idea is to show that A_{TM} is reducible to $\mathrm{HALT}_{\mathrm{TM}}$.

Let's assume that we have a TM R that decides $\mathrm{HALT}_{\mathrm{TM}}$. Then we use R to construct S, a TM that decides A_{TM}. To get a feel for the way to construct S, pretend that you are S. Your task is to decide ATM. You are given an input of the form (M, w). You must output accept if *M accepts w*, and you must output *reject* if M loops or rejects on w. Try simulating M on w. If it accepts or rejects, do the same. But you may not be able to determine whether M is looping, and in that case your simulation will not terminate. That's bad, because you are a decider and thus never permitted to loop. So this idea, by itself, does not work.

Instead, use the assumption that you have TM R that decides $\mathrm{HALT}_{\mathrm{TM}}$. With R, you can test whether M halts on w. If R indicates that M doesn't halt on w, reject because (M, w) isn't in A_{TM}. However, if R indicates that M does halt on w, you can do the simulation without any danger of looping.

Thus, if TM R exists, we can decide A_{TM}, but we know that A_{TM} is undecidable. By virtue of this contradiction we can conclude that R does not exist. Therefore $\mathrm{HALT}_{\mathrm{TM}}$ is undecidable.

PROOF Let's assume for the purposes of obtaining a contradiction that TM R decides $\mathrm{HALT}_{\mathrm{TM}}$. We construct TM S to decide $\mathrm{HALT}_{\mathrm{TM}}$, with S operating as follows.

$S =$ "On input (M, w), an encoding of a TM M and a string w:

1. Run TM R on input (M, w).
2. If R rejects, *reject.*
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, accept; if M has rejected, *reject.*

Clearly, if R decides $\mathrm{HALT}_{\mathrm{TM}}$, then S decides ATM. Because A_{TM} is undecidable, $\mathrm{HALT}_{\mathrm{TM}}$ also must be undecidable.

3.2 Explain the technique of "the reduction" with the proof that $\textsf{REGULAR}_{\textsf{TM}} = \{(M) | M$ is a TM and $L(M)$ is a regular language $\}$ is undecidable

cf Sipser, pp. 191

PROOF IDEA As usual for undecidability theorems, this proof is by reduction from A_{TM}. We assume that REGULAR_{TM} is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM}. Less obvious now is how to use R's ability to assist S in its task. Nonetheless we can do so.

The idea is for S to take its input (M, w) and modify M so that the resulting TM recognizes a regular language if and only if M accepts w. We call the modified machine M_2 . We design M_2 to recognize the nonregular language $\{0^n1^n | n \geq 0\}$ if M does not accept w, and to recognize the regular language Σ^* if M accepts w. We must specify how S can construct such an M_2 from M and w. Here, M_2 works by automatically accepting all strings in $\{0^n1^n | n \geq 0\}$. In addition, if M accepts w, M_2 accepts all other strings.

PROOF We let R be a TM that decides REGULAR_{TM} decide A_{TM}. Then S works in the following manner.

$S =$ "On input (M, w), where M is a TM and w is a string:

1. Construct the following TM M_2.

   $M_2 =$ "On input x:

   a) If x has the form 0^n1^n, *accept*

   b) If x does not have this form, run M on input w and *accept* if M accepts w."

2. Run R on input (M_2).

3. If R accepts, *accept*; if R rejects, *reject*."

3.3 Let $A_{\textsf{LBA}} = \{(B, w) | B$ is a linear bounded automata and B accepts $w\}$, explain why this language is decidable

cf Sipser, pp. 194 -

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 Lemma 5.8, 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 Lemma 5.8. If M has not halted by then, it must be looping.

PROOF The algorithm that decides A_{LBA} is as follows. L = "On input (M, w), where M is an LBA and w is a string:

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

If M on w has not halted within qng^n steps, it must be repeating a configuration according to Lemma 5.8 and therefore looping. That is why our algorithm rejects in this instance.

3.4 State Rice's Theorem and prove it. Give two problems that can be proved undecidable by applying Rice's Theorem

cf Sipser, pp. 213

Definition

Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable. In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial – it contains some, but not all, TM descriptions. Second, P is a property of the TM's language-whenever $L(M_1) = L(M_2)$, we have $(M_1) \in P$ iff $(M2) \in P$. Here, M_1 and M_2 are any TMs. Prove that P is an undecidable language.

cf Sipser, pp. 215

PROOF Assume for the sake of contradiction that P is a decidable language satisfying the properties and let R_p be a TM that decides P. We show how to decide A_{TM} using R_p by constructing TM S. First let $T_\emptyset$ be a TM that always rejects, so $L(T_\emptyset) = \emptyset$. You may assume that $(T_\emptyset) \notin P$ without loss of generality, because you could proceed with $\bar{P}$ instead of P if $(T_\emptyset) \in P$. Because P is not trivial, there exists a TM T with $(T) \in P$. Design S to decide A_{TM} using R_p's ability to distinguish between $T_\emptyset$ and T. S = "On input (M, w):

1. Use M and w to construct the following TM M_w.

   M_w, = "On input x:

   a) Simulate M on w. If it halts and rejects, *reject*.

If it accepts, proceed to stage 2

   b) Simulate T on x. If it accepts, *accept*."

2. Use TM R_p to determine whether $(M_w) \in P$. If YES, *accept*.

   If NO, *reject*."

TM M_w simulates T if M accepts w. Hence $L(M_w)$ equals $L(T)$ if M *accepts* w and $\emptyset$ otherwise. Therefore $(M, w) \in P$ iff M accepts w.

3.5 Define the notion of "reduction function" $f : \Sigma^* \to \Sigma^*$ from a problem A to a problem B and prove the following theorem: "If $A \leq B$ and B is decidable then A is decidable" where $A \leq B$ reads "A is reducible to B"

cf Slides ch5

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a reduction from problem $A \subseteq \Sigma*$ to problem $B \subseteq \Sigma*$ if the following conditions hold:

1. f is a computable function, i.e. there is an algorithm (termination is guaranteed) that given a finite word w, it computes the finite word $f(w)$.
2. $w \in A$ iff $f(w) \in B$, this ensures that positive instances of the problem A are translated into positive instances of the problem B, and negative instances of the problem A are translated into negative instances of the problem B.

Clearly, if there exists a reduction from A to B and B is decidable then A must be decidable. Conversely, if there exists a reduction from A to B and A is undecidable then B must be undecidable.

NB: When a language A is mapping reducible to B, we note it $A \leq_m B$.

THEOREM *If $A \leq_m B$ and B is decidable then A is decidable.*

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 $w \in A$ iff $f(w) \in B$ because f is a reduction from A to B. Then M accepts $f(w)$ iff $w \in A$. So N decides A.

Corollary *If $A \leq_m B$ and A is undecidable then B is undecidable.*

Also:

THEOREM *If $A \leq_m B$ and B is Turing recognizable then A is Turing recognizable.*

Corollary *If $A \leq_m B$ and A is not Turing recognizable then B is not Turing recognizable.*

4 Chapter 7: Time Complexity

4.1 Define the notion of time complexity TIME(f (n)). Explain why we can say that this is a worst-case measure of complexity ? Why do we use big O notation when we reason on the time complexity of a Turing machine

4.2 Define the big O notation and the small o notation. Give examples and explain the relation that exists between those two notions

4.3 Why are deterministic polynomial time Turing machines suitable to study the class P

4.4 Define the running time of a Turing machine that is a decider. If M is a nondeterministic machine that decides the language L in $O(t(n))$, how can we bound the running time of a deterministic Turing machine that decides L

4.5 Define the class P and explain why it is an important complexity class

4.6 Given two examples of problems and their natural encodings. Suggest other encodings that are not reasonable and explain why

4.7 Give an example of a problem which is in NP and prove its membership NP using the notion of certificate

4.8 We have given two definitions of the class NP. One uses "certificates" and one uses nondeterministic Turing machines. Recall the two definitions and prove that they are equivalent

4.9 Explain why NP $\subseteq$ ExpTime

4.10 Define the notion of NP complete problem and explain why this notion is important

4.11 Define the notion of "polynomial time mapping reducible". Show that if A is polynomial time reducible to B and $B \in P$ then $A \in P$

4.12 Prove that 3SAT is polynomial time reducible to CLIQUE

4.13 Prove the following two statements: (i) if $B \in$ NP-Complete and $B \in$ P, then P = NP, (ii) if $B \in$ NP-Complete and $B \leq_P C$, and $C \in$ NP then C is NP-Complete

4.14 Give the main constructions and arguments underlying the proof of the Cook-Levin Theorem that establishes that SAT is NP-Complete

4.15 Prove that 3SAT $\leq_P$ VERTEX-COVER

4.16 Define the HAMILTONIAN-PATH problem and prove it is NP-complete

4.17 Give a dynamic programming algorithm for the PARTITION problem and explain why it does not imply P = NP

4.18 Give a dynamic programming algorithm for the SUBSET-SUM problem and explain why it does not imply P = NP

5 Chapter 8: Space Complexity

5.1 Define the notion of space complexity. Illustrate the difference between time complexity and space complexity by showing an example of problem that can be solved in $O(t(n))$ space but we believe cannot be solved in $O(t(n))$ time

5.2 If a language $L \in$ **SPACE**$(O(t(n))$ with $t(n) \geq n$, what can we say about

the time complexity of L

5.3 State and prove Savitch's theorem

5.4 Define the classes PSPACE, NPSPACE, coPSPACE and coNPSPACE. Explain why all those classes are equal

5.5 Define the notion of PSPACE-Completness and give two examples of problems that are PSPACE-Complete

5.6 Explain why TQBF can be solved in polynomial space

5.7 Give the main arguments and constructions underlying the proof that TQBF is PSPACE-Complete. Explain why the proof of Cook-Levin theorem cannot be applied directly to show that TQBF is PSPACE-Complete

5.8 Prove that Generalized Geography is PSPACE-Complete

5.9 Define the class L and NL. In those definitions we use Turing machines with a read only input tape and read/write working tape, explain why

5.10 Explain why PATH is in NL, and why we believe that it is not in L

5.11 Explain how we can bound the time complexity of a Turing machine which decides a language in L

6 Chapter 9: Intractability

6.1 Define the notion of space constructible function $f : \mathbb{N} \rightarrow \mathbb{N}$. Give an example of such a function and explain why it is so

6.2 Knowing that the equivalence between extended regular expressions is ExpSpace-Complete, show by applying the space hierarchy theorem that this problem can not be solved in deterministic polynomial time, nor in nondeterministic polynomial time

6.3 Define the notion of Turing machine with oracle. Define the set of lan guages that can be solved in $\mathsf{P}^{\mathsf{SAT}}$, define a language that belongs to this set and which is not believed to be in NP (explain why)

6.4 What do we know about the relations between the complexity classes P, NP, PSPACE, EXPTIME, and EXPSPACE? Explain

References

[1] *Answers to exam questions (en/fr) January 2018.* URL: `https://dochub.be/documents/638221`.

[2] *Answers to questions from 2016 - 2017 (in English).* URL: `https://dochub.be/documents/63528`.

[3] *Réponses aux questions d'examen.* URL: `https://dochub.be/documents/63309`.