

Some notes on Decidability and Semi-Decidability

Informal Definition 1. An *idealized computer* is like a real modern computer except that it has an unlimited amount of memory.

Informal Definition 2. Let Σ be a finite alphabet. An *algorithm on Σ* is a computer program, M , (written in your favorite language) which runs on an idealized computer and does the following:

- (a) M takes as input a string w on Σ .
- (b) Given any input string w , M does one of the following three things:
 - (i) M eventually **halts**, declares that “ w is accepted,” and outputs some string u on Σ ; or
 - (ii) M eventually **halts** and declares that “ w is rejected”; or
 - (ii) M **never halts**, i.e. M enters an “infinite loop.”

Note that in (i) and (ii) above there is no restriction on the amount of time it takes before M halts.

Remark 0.1. At first we will be interested only in whether M accepts the string w . For these considerations, the output string u is irrelevant and will be ignored. We will be interested in u for other reasons later. Notice that there are two ways for M to not accept w . Given the input w , M might halt and explicitly reject w ; or M might fail to accept w because it never halts.

The two informal definitions above are designed to give you an intuitive feel for what is going on. Later we will replace the informal notion “ M is an algorithm on Σ ” with the completely formal notion “ M is a *Turing Machine* on Σ .” Turing Machines are just one way of making the notion of an algorithm precise. In the definitions and theorems below, I will use the word “algorithm.” If this word is replaced by the words “Turing Machine,” then the definitions and theorems will be perfectly mathematically rigorous.

Let M be an algorithm on Σ . There is an obvious way to associate with M a certain *language*, $L(M)$.

$$L(M) = \{ w \in \Sigma^* : M \text{ accepts } w \}.$$

Notice that the relationship between M and $L(M)$ here is a bit weaker than it was in the situation where M was a dfa. Since our M here might sometimes not halt, instead of calling M an *acceptor* we will call it a *semi-acceptor*. The following definition makes this precise.

Definition 0.2. Let L be a language on Σ and let M be an algorithm on Σ . We say that M is a *semi-acceptor* for L iff given any string $w \in \Sigma^*$ as input M does the following:

- If $w \in L$ then M eventually halts and accepts w .
- If $w \notin L$ then either
 - M eventually halts and rejects w , or
 - M never halts.

NOTICE: M is a semi-acceptor for L iff $L = L(M)$.

Here is the interesting thing about semi-acceptors. Suppose you input the string w to M and you have been waiting for a very long time for M to halt and so far it has not. What can you conclude? The answer is: nothing. It may be that you have waited a very long time because M is never going to halt. In this case $w \notin L(M)$. Or it may be that in the next minute M is going to halt and accept w . In this case $w \in L(M)$. There is no way for you to tell whether your waiting will ever pay off!

If M is a dfa, then given any input string w , M always halts and either accepts or rejects w . When this happens we say that M is an acceptor.

Definition 0.3. Let L be a language on Σ and let M be an algorithm on Σ . We say that M is an *acceptor* for L iff given any string $w \in \Sigma^*$ as input M does the following:

- If $w \in L$ then M eventually halts and accepts w .
- If $w \notin L$ then M eventually halts and rejects w .

NOTICE: M is an acceptor for L just in case: (i) $L = L(M)$; and (ii) M always halts on all inputs w .

Remark 0.4. So M is always a semi-acceptor for $L(M)$, and M may or may not be an acceptor for $L(M)$, depending on whether or not M always halts.

Definition 0.5. A language L is called *semi-decidable* iff it has some semi-acceptor M . That is, if and only if there is some M such that $L = L(M)$. L is called *decidable* iff it has some acceptor M . That is, if and only if there is some M such that: (i) $L = L(M)$; and (ii) M always halts on all inputs w .

Notice that if L is decidable then L is also semi-decidable. Later we will show that there are some languages which are semi-decidable but not decidable.

Remark 0.6. Another term for decidable that you may see is *recursive*. The term recursive is a bit confusing because it has other meanings, so we will not use it. Another term for semi-decidable that you may see is *recursively enumerable* or R.E.. We will talk later about the use of the word “enumerable.”

EXAMPLE Let $L = \{ a^n : n \text{ is prime } \}$. We know that L is not context-free. We will show here that L is decidable. To see this we must find an acceptor M for L . But how do we describe M ? Later when we talk about Turing Machines, you will be able to write down a Turing Machine M which is an acceptor for L . But this is usually quite tedious. In general for problems in which you are asked to show that a language L is decidable, it will be sufficient for you to write down an algorithm M which is an acceptor for L . But how do you write down an algorithm? As I have defined it above, an algorithm is a computer program in your favorite language. So to “give an algorithm” is just to give a program in any language. For convenience, I will always express my algorithms in an informal “pseudo-code.” So here is a pseudo-code algorithm M which is an acceptor for the language $L = \{ a^n : n \text{ is prime } \}$.

- | | |
|---|--|
| <p>(1) Input(w);</p> <p>(2) Let $n := \text{length}(w)$;</p> <p>(3) If ($n = 0$ or $n = 1$) then reject and quit;</p> | <p>(4) For $k := 2$ to $n - 1$
 For $s := 2$ to k
 If ($s * k = n$) then reject and quit;</p> <p>(5) accept and quit.</p> |
|---|--|

1 The Language Family Hierarchy Theorem

In this section we will prove the following theorem.

Theorem 1.1. Fix an alphabet Σ . Let \mathcal{R} be the family of regular languages on Σ . Let \mathcal{C} be the family of context-free languages on Σ . Let \mathcal{D} be the family of decidable languages on Σ . Let \mathcal{S} be the family of semi-decidable languages on Σ . Let \mathcal{U} be the family of all languages on Σ . Then:

$$\mathcal{R} \subsetneq \mathcal{C} \subsetneq \mathcal{D} \subsetneq \mathcal{S} \subsetneq \mathcal{U}.$$

In the above theorem $\mathcal{R} \subsetneq \mathcal{C}$ means that \mathcal{R} is a proper subfamily of \mathcal{C} . That is, every regular language is context free, but there are some context-free languages which are not regular.

We now turn towards a proof of the above theorem. We prove the theorem in a series of lemmas.

Lemma 1.2. $\mathcal{R} \subsetneq \mathcal{C}$.

Proof. Every regular language is context-free because a regular grammar is also a context-free grammar. The language $\{a^n b^n : n \geq 0\}$ is an example of a language that is context-free but not regular. \square

Lemma 1.3. $\mathcal{C} \subsetneq \mathcal{D}$.

Proof. The language $\{a^n : n \text{ is prime.}\}$ is an example of a language that is decidable but not context-free. Now we must show that every context-free language is decidable. At first this may seem trivial, but actually it is not. The problem is that an npda cannot be interpreted directly as an algorithm because of its non-determinism. (A dfa can be interpreted directly as an algorithm, and so it is trivially true that every regular language is decidable.)

Let L be a context-free language. We will describe an algorithm M which is an acceptor for L . Let $L' = L - \{\lambda\}$. (λ may or may not be an element of L ; if it is not then $L = L'$.) We will need to use the following fact:

FACT. There is a grammar G for L' that has no λ -productions and no unit productions.

A λ -production is a production rule of the form $A \rightarrow \lambda$. A unit-production is a production rule of the form $A \rightarrow B$, where A and B are both auxiliary symbols. The FACT is the content of Theorems 6.3 and 6.4 of our textbook. You are not responsible for knowing how to prove these. If you are interested, read pages 163 through 166.

Let G be a grammar as in the FACT above. The significance of the FACT is the following observation.

OBSERVATION. Let $w \in L'$, let $n = |w|$, and let $S \Rightarrow^* w$ be a derivation of w from G . Then the length of the derivation, that is the number of steps in the derivation, is less than $2n$.

The reason this is true is the following. Notice that every production rule in G is of one of the following two forms: (i) $A \rightarrow u$ where $|u| \geq 2$; or (ii) $A \rightarrow a$ where a is a main symbol. Since G has no λ -productions, no step of the derivation decreases the size of the sentential form. There can be at most $n - 1$ steps of the derivation of form (i) because each such step increases the size of the sentential form by at least 1. There can be at most n steps of the derivation of form (ii) because each such step adds a symbol to the final sentence w .

Using the OBSERVATION I will now describe an algorithm M which is an acceptor for L . Let LAMBDA be a Boolean constant which is equal to TRUE if $\lambda \in L$ and equal to FALSE if $\lambda \notin L$.

- (1) input w ;
- (2) if $w = \lambda$ and LAMBDA then accept and halt;
- (3) if $w = \lambda$ and NOT LAMBDA then reject and halt;
- (4) Let $X := \{S\}$;
- (5) For $i := 1$ to $2n - 1$
 - For R ranges over all of the production rules in G
 - For u ranges over all of the elements of X such that the left-most auxiliary symbol of u is the same as the left-hand side of rule R
 - (5a) Let u' be the result of applying rule R to u ;
 - (5b) If $u' = w$ then accept and halt;
 - (5c) If u' contains auxiliary symbols then let $X := X \cup \{u'\}$;
- (6) reject and halt

Note that it is the OBSERVATION on the previous page that allows me to have the upper bound of $2n - 1$ on the outer loop in step (5) above. Since this algorithm always halts it is an acceptor for L . Thus L is decidable. □

Continuing with the proof of Theorem 1.1, notice that it is trivial that every decidable language is semi-decidable. So to complete the proof of the theorem we need to prove two more lemmas:

Lemma 1.4. *There is a language which is semi-decidable but not decidable.*

Lemma 1.5. *There is a language which is not semi-decidable.*

The proofs of these two lemmas are a bit more sophisticated than the earlier material, and we must make a small diversion before we can get to them. One of the main ideas is the following:

IDEA 1. Let Σ be an alphabet. Let M be an algorithm on Σ . Then M can be thought of as a string on Σ .

Why is this true? Well, what is an algorithm? The way we have defined it, an algorithm is simply a computer program. Well a computer program is something you can write down using a certain set of symbols, such as for instance: letters of the alphabet and parentheses and semi-colons and white-space symbols. So a computer program is a string on some alphabet Σ' . Now if all of the symbols we need to write down the program are already symbols of Σ , then we are done and we see that IDEA 1 is true. Of course in general, Σ will not contain all of the symbols we need to write down a computer program. In that case we must use a form of CODING. We must code each symbol of Σ' by a string of symbols from Σ . For example, if $\Sigma = \{a, b\}$ and we need a semi-colon symbol to write down our programs, we could decide to code the semi-colon symbol as: aba . The details of this coding are uninteresting. It should not be hard to convince yourself that it can be done. Once this coding has been done, then we see that IDEA 1 is true.

Using IDEA 1 we can *enumerate* all algorithms on Σ .

Definition 1.6. Fix an alphabet Σ . For all strings $w \in \Sigma^*$ we now define an algorithm on Σ called M_w . The way we do this is as follows. We look at the string w and ask ourselves whether or not w codes a well-formed computer program on Σ using the coding scheme described above. If w does code a well-formed computer program on Σ then we let M_w be this program. Otherwise we let M_w be the dummy algorithm; that is the algorithm which takes input but does nothing and immediately rejects its input and halts.

Remark 1.7. The set $\{M_w : w \in \Sigma^*\}$ includes every possible algorithm on Σ .

Using the above definition we can now define a *universal algorithm*. As you will see in the following definition, a universal algorithm is just like a programmable computer.

Definition 1.8. Let Σ be a finite alphabet. A *universal algorithm on Σ* is a computer program, M , (written in your favorite language) which runs on an idealized computer and does the following:

- (a) M takes as input two strings on Σ : w and v .
- (b) M then does whatever M_v does when M_v is given the input w .

It is not hard to see that there really is such a thing as a universal algorithm. To actually write down a universal algorithm in any given programming language would be quite a difficult task though. Essentially, you would be writing a compiler/interpreter for the given language. The universal algorithm would have to perform the following steps:

- (1) decode v to get the actual text of the computer program coded by v
- (2) simulate a run of that computer program on input w .

Now we can prove Lemma 1.5, which states that there is a language which is not semi-decidable. In fact we will give an example of a language which is not semi-decidable. The example is:

$$L = \{w : w \notin L(M_w)\}.$$

I claim that L is not semi-decidable.

Proof. Suppose towards a contradiction that L is semi-decidable. Then there is an algorithm M such that $L(M) = L$. By Remark 1.7 above, there is some string w such that $M = M_w$. Fix such a string w . To get our contradiction we ask the question: Is w in L ? We find that:

$$w \in L \Rightarrow w \in L(M_w) \Rightarrow w \notin L; \text{ and}$$
$$w \notin L \Rightarrow w \notin L(M_w) \Rightarrow w \in L.$$

Either way this is a contradiction. So there must be no such w . So there must be no M such that $L(M) = L$. So L is not semi-decidable. \square

There is one last thing to do before we have finished with the proof of Theorem 1.1. We must prove Lemma 1.4 which states that there is a language L which is semi-decidable but not decidable. Towards this end we first discuss another interesting fact:

Lemma 1.9. *If L is decidable then \bar{L} is decidable.*

Here \bar{L} means the complement of L .

Proof. The proof is exactly the same as the corresponding proof for regular languages. Let M be an algorithm which is an acceptor for L . We will now describe an algorithm \bar{M} which is an acceptor for \bar{L} :

- (1) input(w);
- (2) run the algorithm M as a subroutine on the input w ;
- (3) if M accepts w
 then reject and halt
 else accept and halt;

\square

Remark 1.10. Notice that in step (2) above, since M is an acceptor, we know that it will eventually halt and return control back to the main algorithm. Notice also that this argument does not work if we only assume that M is a semi-acceptor.

Using the previous lemma we can now prove Lemma 1.4 which states that there is a language L which is semi-decidable but not decidable. In fact we will give an example of such a language: The example is:

$$L_1 = \{ w : w \in L(M_w) \}.$$

Let also

$$L = \{ w : w \notin L(M_w) \}.$$

Recall that above we proved that L is not semi-decidable. Notice that L_1 is the compliment of L . I claim that L_1 is semi-decidable but not decidable.

Proof. First let's see that L_1 is not decidable. Well, if L_1 were decidable then by Lemma 1.9 L would be decidable. But L is not even semi-decidable!

Now let's see that L_1 is semi-decidable. We will describe an algorithm M_1 which is a semi-acceptor for L_1 . Let M be a universal algorithm.

- (1) input(w);
- (2) run M as a subroutine on the inputs (w, w) ;
- (3) if M halts and accepts (w, w) [this means that M_w halts and accepts w , so $w \in L(M_w)$, so $w \in L_1$] then accept and halt;
- (3) if M halts and rejects (w, w) [this means that M_w halts and rejects w , so $w \notin L(M_w)$, so $w \notin L_1$] then reject and halt;

There is a subtlety in the above algorithm that you should notice. In step (2), the universal algorithm M is not guaranteed to ever halt. If M does not halt then control will never be returned to the main algorithm and so the main algorithm will not halt. But notice that this is ok. If M does not halt on input (w, w) this means that M_w does not halt on input w . But this means that $w \notin L(M_w)$ and so $w \notin L_1$, and so it is ok for our algorithm (which is only a semi-acceptor) to not halt. \square

Well this finishes the proof of the Language Family Hierarchy Theorem. Now we turn to ...

2 Other Interesting Matters

This next lemma tells us an interesting relationship between decidable and semi-decidable.

Lemma 2.1. *Let L be a language. Then the following are equivalent:*

- (a) L is decidable.
- (b) L is semi-decidable and \bar{L} is semi-decidable.

Proof. Assume (a). By the Lemma 1.9, \bar{L} is also decidable. Since L is decidable, L is semi-decidable. Since \bar{L} is decidable, \bar{L} is semi-decidable. So (b) is true.

Now assume (b). Let M_1 and M_2 be algorithms such that $L(M_1) = L$ and $L(M_2) = \bar{L}$. We will now describe an algorithm M which is an acceptor for L :

- (1) input(w);
- (2) run M_1 as a subroutine on the input w , but only for 1 step;
- (3) if M_1 accepts w in its first step then accept and halt;
- (4) run M_2 as a subroutine on the input w , but only for 1 step;
- (5) if M_2 accepts w in its first step then reject and halt;
- (6) run the next step of algorithm M_1 on the input w ;
- (7) if M_1 accepts w in this next step then accept and halt;
- (8) run the next step of algorithm M_2 on the input w ;
- (9) if M_2 accepts w in this next step then reject and halt;
- (10) goto (6);

Notice that if $w \in L$ then M_1 will eventually accept w and so the test in step (7) will eventually evaluate to true and so M will eventually accept w and halt. On the other hand, if $w \in \bar{L}$ then M_2 will eventually accept w and so the test in step (9) will eventually evaluate to true and so M will eventually reject w and halt. So M is an acceptor for L . So L is decidable. So (a) is true. \square

Remark 2.2. A more concise way to describe the algorithm in the previous proof is the following:

Run M_1 and M_2 *simultaneously* on the input w . Eventually one of the two will accept w . If it is M_1 then accept and halt. If it is M_2 then reject and halt.

What we have done in the longer more explicit algorithm in the proof above is to simulate parallel processing using time-slicing.

When we studied regular languages and context-free languages we showed that these families could be characterized in terms of certain types of *grammars*. Well we can extend this idea to the family of semi-decidable languages.

Lemma 2.3. *A language is semi-decidable iff there is some grammar that generates it.*

Sketch of Proof. Let L be a language. First suppose that there is a grammar G such that $L(G) = L$. It is not difficult to come up with an algorithm which is a semi-acceptor for L . This algorithm will be very much like the algorithm on the top of page 4, but without the stopping condition of $2n - 1$ on the outer loop.

The other direction is a lot more technical. We need to take any algorithm and convert it into an equivalent grammar. In order to make this task manageable, we must fix a simple language in which to express our algorithms, and we must be more precise about exactly what an algorithm is. One way to do this is using the notion of a Turing Machine. If we take the step of re-defining an algorithm to be a Turing Machine, then in order to prove the lemma we must show how to convert any Turing Machine into an equivalent grammar. This is done in Section 11.2 of our textbook. You will not be responsible for this proof, but read the section if you are interested. \square