CSCI3390-Lecture 7: Undecidability Proofs; Reductions

September 21,2018

1 Summary

- We prove the claims that all the problems about Turing machines listed in the last lecture are undecidable.
- The main tool for proving this is to *reduce* the problem that we want to show is undecidable to a problem we already know to be undecidable.

2 A collection of undecidable problems about Turing machines

Recall our list of problems.

2.1 Halting Problem, version 1

Input: A Turing machine $\mathcal{M}$ and a string $w \in \{0, 1\}^*$.
Output: Yes if $\mathcal{M}$ eventually halts when started on input w, No otherwise.

2.2 Halting Problem, version 2

Input: A Turing machine $\mathcal{M}$.
Output: Yes if $\mathcal{M}$ eventually halts regardless of the input string (*i.e.,* $\mathcal{M}$ is free of infinite loops), No otherwise (*i.e.,* if $\mathcal{M}$ runs forever on some input).

2.3 Nonemptiness

Input: A Turing machine $\mathcal{M}$.
Output: Yes if there is some $w \in \{0,1\}^*$ accepted by $\mathcal{M}$, no otherwise.

2.4 Finiteness.

Input: A Turing machine $\mathcal{M}$.
Output: Yes if the set of strings accepted by $\mathcal{M}$ is finite, No if $\mathcal{M}$ accepts infinitely many strings.

2.5 Equivalence

Input: Two Turing machines $\mathcal{M}, \mathcal{N}$.
Output: Yes $\mathcal{M}$ and $\mathcal{N}$ accept exactly the same strings, no otherwise.

2.6 Minimality

Input: A Turing machine $\mathcal{M}$.
Output: Yes if $\mathcal{M}$ is the smallest Turing machine that recognizes the set of strings it accepts, No if there is a smaller machine that recognizes the same language.

Here you have to say what exactly you mean by the size of a Turing machine. One way to do this is to fix an encoding scheme, and use the length of the encoding of $\mathcal{M}$ as the size of $\mathcal{M}$.

3 Reductions

The main tool for proving a given problem Y is undecidable is to *reduce* another problem X that we already know is undecidable to Y. In essence, this means that we show that if we had an algorithm to decide Y, then we would have such an algorithm for X (which we already know is not possible). We sometimes say that we prove Y is undecidable by a 'reduction from X'. Think of it this way—if we had a Y subroutine, we could use it to write an X program.

The actual form these reductions take is to provide an algorithm for transforming an instance u of problem X into an instance u' of problem Y such that u' is a positive instance (a Yes instance) of Y if and only if u is a Yes instance for X.

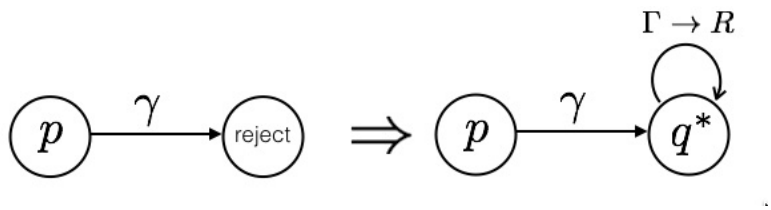

Figure 1: *Illustration of the reduction for the first version of the halting problem. Every transition of $\mathcal{M}$ into the rejecting state is treated this way. The new machine cycles endlessly in the new state q^* regardless of the tape symbols it encounters.*

Then if we had an algorithm for Y we would obtain the following algorithm for X: Take an input u, transform it to u' and apply the algorithm for Y to u'.[1]

Let's see how this works in practice with our collection of problems from the last section:

3.1 Halting Problem, version 1

We reduce L_{TM} to this problem. Given a Turing machine $\mathcal{M}$, we modify it to produce a new Turing machine $\mathcal{M}'$ as follows: We replace the rejecting state by a new state q^* such that if $\mathcal{M}'$ enters q^*, it runs forever. The modification is shown in Figure 1.

The result is that $\mathcal{M}'$ halts on an input string w if and only if $\mathcal{M}$ accepts w, because if $\mathcal{M}$ rejects w, $\mathcal{M}'$ would run forever. So if we had an algorithm for the halting problem, we would get one for L_{TM}. Given an input $(\mathcal{M}, w)$ for L_{TM}, we transform it to $(\mathcal{M}', w)$, and test if $\mathcal{M}'$ halts on w. If it does, then $\mathcal{M}$ accepts w, and if $\mathcal{M}'$ does not halt on w, then $\mathcal{M}$ does not accept w.

3.2 Halting Problem, version 2

We reduce version 1 to this problem. Given $\mathcal{M}, w$, we will show how to construct a Turing machine $\mathcal{M}'$ such that $\mathcal{M}'$ halts on every input if and only if $\mathcal{M}$ halts on w. Here is how $\mathcal{M}'$ will work: initially, it *erases* its input, and then writes the

[1] Students often get which way the reduction works backwards, and for good reason. To show Y is undecidable, we reduce the known undecidable problem X to Y. But by this means, according to a more common use of 'reduce', we are reducing the problem of proving Y undecidable to the problem of proving X undecidable. So, yes, it's confusing. You just have to think it through.

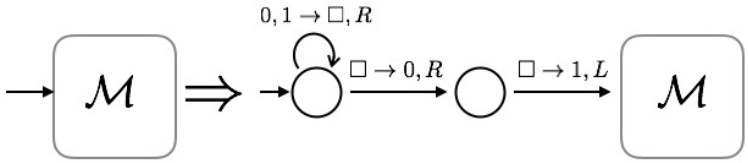

Figure 2: *Illustration of the reduction for the second version of the halting problem, as well as the finiteness problem. The new machine begins by erasing its input and then writing the string w (in this case 01) on the tape, and then running $\mathcal{M}$ on w.*

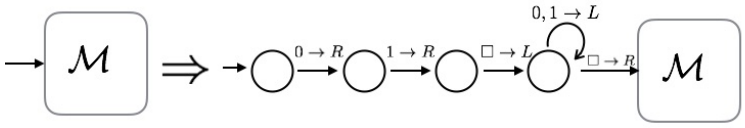

Figure 3: *Illustration of the reduction for the nonemptiness problem. The new machine first verifies that its input is identical to w (in this case 01), rejecting otherwise. It then runs $\mathcal{M}$ on w.*

input string w on its tape—this is hard-wired into the transitions of $\mathcal{M}'$. After this initial phase, $\mathcal{M}'$ returns to the start of the input and does what $\mathcal{M}$ does. This is illustrated in Figure 2 for the case $w = 01$. Observe that in contrast to the previous problem, the construction of $\mathcal{M}'$ depends on the string w. So we can test if $\mathcal{M}$ halts on w by testing whether $\mathcal{M}'$ halts on every input.

3.3 Nonemptiness

We reduce L_{TM} to this problem. Given $\mathcal{M}, w$ we construct a TM $\mathcal{M}'$ that accepts the string w if $\mathcal{M}$ accepts w, and accepts nothing at all if $\mathcal{M}$ does not accept w. So to test if $\mathcal{M}$ accepts w, we construct $\mathcal{M}'$ and test it for nonemptiness. $\mathcal{M}'$ works as follows: In an initial scan, it checks to see if its input is equal to w, and if it is not, $\mathcal{M}'$ *rejects* the input. Otherwise, $\mathcal{M}'$ returns to the left end of the tape, and simulates $\mathcal{M}$. Thus $\mathcal{M}'$ is a Yes instance of the nonemptiness problem if and only if $\mathcal{M}, w$ is a Yes instance of L_{TM}. The construction is illustrated in Figure 3 for the case $w = 01$.

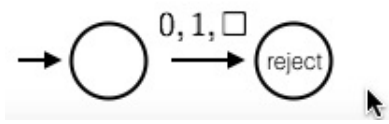

Figure 4: *The machine $\mathcal{M}_\emptyset$ used in the reduction of the equivalence problem to the emptiness problem. This machine accepts no string over the alphabet $\{0, 1\}$, thus it recognizes the empty language.*

3.4 Finiteness

Again, we reduce from L_{TM}. We use exactly the same construction as in version 2 of the halting problem: Given $\mathcal{M}$ and w, we construct a new TM $\mathcal{M}'$ that erases its input, writes w on the tape, and then simulates $\mathcal{M}$ on w. Thus $\mathcal{M}'$ accepts an infinite set of strings (in fact *every* string) if $\mathcal{M}$ accepts w, and a finite set of strings (the empty set) if $\mathcal{M}$ does not accept w.

3.5 Equivalence

Reduction from nonemptiness (or, rather, from its complement emptiness). If we could test for equivalence of two Turing machines, we could test whether a given Turing machine $\mathcal{M}$ is equivalent to the TM $\mathcal{M}_\emptyset$ shown in Figure 4, which recognizes the empty language. That is, we transform an instance $\mathcal{M}$ of the emptiness problem into the instance $(\mathcal{M}, \mathcal{M}_\emptyset)$ of the equivalence problem. Thus if we could decide whether two TMs are equivalent, we could decide whether a given TM recognizes the empty language.

4 Reductions formally

Remember that in a more formal setting, decision problems are languages over some input alphabet. So, formally, a reduction of a (known undecidable) problem X to another problem Y is a reduction of a an undecidable language $L \subseteq \Sigma_1^*$ to $L' \subseteq \Sigma_2^*$. The reduction itself is a function

$$f : \Sigma_1^* \to \Sigma_2^*$$

with the property that $w \in L$ if and only if $f(w) \in L'$. One additional, crucial requirement is that f be *computable*. There has to be an algorithm for computing

f; that is, there is some TM $\mathcal{N}$ such that $f = f_{\mathcal{N}}$. We adopt our usual practice of assuming the Church-Turing thesis, and simply describe the algorithm, rather than the Turing machine $\mathcal{N}$.

For instance, the reduction for version 2 of the halting problem is officially a function that takes a string $< \mathcal{M}, w > \in \{0,1\}^*$ and produces the string $< \mathcal{M}' > \in \{0,1\}^*$.

5 A different kind of reducibility

The reductions described above are called *mapping* reductions. There is a more general kind of reduction, which we can also use to prove undecidability. Let us describe another proof that the first version of the halting problem is undecidable, using the fact that L_{TM} is decidable. Assuming we have an algorithm $\mathcal{A}$ for the halting problem, here is our algorithm for L_{TM}: Given $\mathcal{M}$, w, use $\mathcal{A}$ to determine if $\mathcal{M}$ halts on w. If the answer is 'no', then reject: we know that $\mathcal{M}$ does not accept w. If the answer is 'yes', then we know that $\mathcal{M}$ either accepts or rejects w. So now we can just run $\mathcal{M}$ on w, assured that we will eventually get a result, and accept or reject accordingly. Thus we are able to decide in any case whether $\mathcal{M}$ accepts w. Of course we already know that we *can't* decide this, so the algorithm $\mathcal{A}$ does not exist.

6 Some decidable problems

Consider the following two problems:

- Input: a TM $\mathcal{M}$ and an input word w over the input alphabet of $\mathcal{M}$. Output: Yes, if $\mathcal{M}$ halts within the first million steps when run on w; no otherwise.
- Input: a TM $\mathcal{M}$ and a word w over the input alphabet of $\mathcal{M}$. Output: Yes, if $\mathcal{M}$ visits fewer than 20 tape cells when run on w; no otherwise.

The first of these problems is obviously decidable: The algorithm to decide it is to simply simulate $\mathcal{M}$ on w, counting steps, until either $\mathcal{M}$ halts, or one million steps have been performed and the machine is still going.

While this is less obvious, the second problem is also decidable. Consider the following algorithm, which will tell us if the reading head ever moves more than 20 tape cells from the starting point. The total space we will allow the head

to cover consists of 41 cells. If Γ is the tape alphabet, then there are $|\Gamma|^{41}$ ways we can write symbols into those 41 cells. There are $|Q|$ possible states, and 41 possible positions for the reading head. All in all, this makes

$$41 \cdot |Q| \cdot |\Gamma|^{41}$$

allowable configurations that $\mathcal{M}$ can be in and still remain within the allowed region. So our algorithm is this: Simulate $\mathcal{M}$ on w for $1+41\cdot|Q|\cdot|\Gamma|^{41}$ steps. If the reading head leaves the allowed region during this simulation, then then answer is no. If the reading head does not leave the region, then some configuration must have occurred twice: That is, from some configuration c, after running for some positive number of steps, the machine re-enters c. This means that it will continue to repeat the same sequence of configurations, over and over again, and will never leave the allowed region. (It also means we know the machine will never halt.) The original problem asks whether the machine ever visits 20 or more cells. So far, we have only determined whether or not it strays out of the allowed 41-cell region. Of course, if it does, then we know it visited more than 20 cells. If it doesn't, we can count the exact number of distinct cells visited during the simulation and get the answer.