Help with reductions

Reduction proofs are short and look simple. And your solutions to homework problems will be very similar to the examples in the book and lectures. However, reductions involve an unfamiliar type of abstraction, so you may find these problems unexpectedly hard. This handout tries to provide some additional help and examples to consider, to supplement those in the textbook.

Is my language decidable or not?

Some problems ask you to determine whether some language $L$ is decidable or not (and perhaps prove that your answer is correct).

Rice’s theorem (Sipser solved exercise 5.28) states that any non-trivial property of Turing machines that depends only on the language that the Turing machine accepts is undecidable. For example, it’s undecidable whether a Turing machine has a language that’s non-empty, contains 17 elements, contains the string “Vassar”, is infinite, and so forth.

Facts about the Turing machine’s structure are decidable. For example, you can tell if the Turing machine has more than 15 states, has no transitions into its accept state, or the like. Such properties just require parsing the Turing machine’s code and doing some straightforward check (e.g., counting the number of states listed).

Facts about a Turing machine’s behavior may or may not be decidable, and it’s not always easy to tell which. For example, it is decidable whether a Turing machine ever moves left, but it is not decidable whether a Turing ever moves left three times in a row.

Basic template

To prove that a language $L$ is undecidable, you will normally want to use a reduction from a language already known to be undecidable (e.g., $A_{TM}$). A reduction proof will look something like:

Suppose that $L$ is decidable. Let $R$ be a Turing machine deciding $L$. We will now construct a Turing machine $S$ that decides $A_{TM}$.

$S$ is constructed as follows:

- Input is $\langle M, w \rangle$, where $M$ is the code for a Turing machine, and $w$ is a string.
- [Pseudocode explaining how $S$ decides whether $M$ accepts $w$, using $R$ as a subroutine.]

But we know that $A_{TM}$ is undecidable, so $S$ can’t exist. Therefore we have a contradiction, and $L$ must have been undecidable.

This is called reducing $A_{TM}$ to $L$. Sometimes you can write a shorter/simpler proof by replacing $A_{TM}$ with some other problem known to be undecidable. If so, remember that you
may need to change the inputs to $S$, e.g., elements of the language $EQ_{TM}$ are pairs of Turing machines. So a reduction from $EQ_{TM}$ requires constructing a Turing machine $S$ whose inputs are pairs $(M_1, M_2)$ of two Turing machines.

If you aren’t sure what problem to reduce to $L$, $A_{TM}$ is normally a good choice.

For simple examples of the basic idea, see Sipser’s proof that $HALT_{TM}$ is undecidable (p. 217) and his proof that $EQ_{TM}$ is undecidable (p. 220).

**Turing machine language problems**

In the simple examples above, $S$ fed its inputs $M$ and $w$ more or less directly into the subroutine $R$. Many other problems require $S$ to rewrite the code for $M$ before feeding it to $R$. It is often easier to imagine your Turing machines as written in Python or C. That is, $S$ takes the source code for $M$ – imagine a file of Python code – and writes a new file of source code for a new machine $M'$. Often, you can imagine $S$ doing the rewrite by simple copy-and-paste, e.g., copying the input code for $M$ and then adding a new main function and perhaps a few additional declarations.

The new Turing machine typically has $w$ “hardcoded” into it. If you were writing the new machine’s code in Python or C, you would write a local variable declaration and copy the input value of $w$ into that declaration. E.g., if the input value was “Bob the Builder”, the code for the new machine would contain a line like: $w = 'Bob the Builder'$. When hardcoding input $w$, we will typically use $M_w$ as the name for the new Turing machine.

The Turing machine $M_w$ is typically designed so it accepts one set of strings (call the set $X$) if $M$ accepts $w$, and a totally different set of strings (call the set $Y$) if $M$ doesn’t accept $w$. One of these sets should have the property that $R$ tests for and the other should not. For example, in the reductions for $L_{VASSAR}$ and $HALT_{EMPTY_{TM}}$ (in the additional examples section below), $M_w$ accepts all input strings $x$ if $M$ accepts $w$, and it accepts no input string $x$ if $M$ doesn’t accept $w$.

The code for $M_w$ typically involves simulating $M$ on $w$, plus some other tests on $x$ that can clearly be computed in a predictable amount of time.

For some reductions, we have $M_w$ test $x$ first, e.g.,

- input is a string $x$.
- If $x$ has some easily tested property $P$, accept.
- Otherwise, simulate $M$ on $w$ and accept if $M$ accepts $w$.

For either of these two steps, you could also take the opposite action, e.g., reject all strings $x$ that have property $P$. For examples of this pattern, see Sipser’s reductions for $E_{TM}$ (p. 217) and $REGULAR_{TM}$ (p. 219).

For some reductions, we have $M_w$ simulate $M$ on $w$ first, before we test $x$, e.g.,

- Simulate $M$ on $w$.
- If $M$ rejects, reject.
- Otherwise, accept exactly when $x$ has some easily tested property $P$.

Examples of this pattern include $L_3$ (in the additional examples section), and the proof of Rice’s theorem (see Sipser’s solved exercise 5.28 on p. 213).
Common problems writing these reductions

A common problem is to get confused about which values are input to which Turing machines. In the above proofs, the string \( w \) is an input to the machine \( S \), i.e., the machine that decides \( A_{TM} \). The string \( x \) is an input to the Turing machine \( M_w \). Look back through the proofs and check carefully where \( x \) occurs and where \( w \) occurs.

The input to the Turing machine \( R \) is usually the code for the machine \( M_w \). You can imagine \( R \) as a Python or C program, which reads an ASCII file of source code for some other program. Note that \( w \) and \( x \) are not inputs to \( R \). The value of \( w \) is hardcoded into the code for \( M_w \). The value of \( x \) is not specified. You can imagine that \( M_w \) will ask the user to input a value for \( x \).

Since the value of \( x \) is not specified, \( R \) has to consider all possible values for the input \( x \) and figure out what \( M_w \) would do on each of these values. If \( R \) is testing a non-trivial property of \( M_w \), you can imagine that it might be hard to figure out what it will do on all possible input strings \( x \). Remember that \( R \) is only a hypothetical Turing machine whose existence will be contradicted at the end of the proof. That is, \( R \) can't really exist, because it's trying to do a task that's too hard.

Proving that a behavior is undecidable

Undecidable behavior properties normally restrict the Turing machine's behavior in cosmetic ways but still allow it to complete the standard range of Turing machine tasks. Such a problem involves a language \( L \) which is a set of all Turing machines with a specific behavior \( B \). E.g.,

\[
L = \{ \langle M \rangle \mid M \text{ prints three 1s in a row on blank input} \}
\]

or

\[
L = \{ \langle M \rangle \mid M \text{ has a state which is never entered on any input string} \}.
\]

For this class of problems, reductions typically require modifying the input Turing machine code \( \langle M \rangle \) to create code for a new Turing machine \( \langle M' \rangle \) which does pretty much the same thing as \( M \) except that

- When \( M \) would have halted, \( M' \) first does behavior \( B \) before halting, and
- \( M' \) never accidently does behavior \( B \) while it's computing.

Typical examples of this type of reduction are the proofs for \( L_{\text{111}} \) and \( L_x \) in the additional examples section. In the case of \( L_{\text{111}} \), we ensure that \( M' \) can never accidently print three ones in a row by replacing every instance of 1 in its code by a new character \( 1' \).

Proving that a behavior is decidable

Decidable behavior properties normally restrict the Turing machine's behavior in a way that significantly cripples it. For an example of a decidable Turing machine behavior, consider the language

\[
L = \{ \langle M \rangle \mid M \text{ never moves left on input "VASSAR"} \}.
\]
L is decidable, because Turing machines that never move left are so crippled that they either halt very quickly, or not at all.

Specifically, if a Turing machine $M$ never moves left, it reads through the whole input, then starts looking at blank tape cells. Once it is on the blank part of the tape, it can cycle through its set of states, but after $|Q|$ moves, it has run out of distinct states and must be in a loop. So, if you watch $M$ for six moves (the length of the string “VASSAR”) plus $|Q| + 1$ moves, it has either halted or it is in an infinite loop.

Therefore, to decide $L$, you run the input Turing machine for $|Q| + 7$ moves. After that many moves, it has either

- moved left (in which case you reject), or
- has halted or gone into an infinite loop without ever moving left (in which case you accept).

This algorithm is a decider (not just a recognizer) for $L$, because it definitely halts on any input Turing machine $M$.

**More Reduction Examples**

**The language $L_{\text{VASSAR}}$**

Let $L_{\text{VASSAR}} = \{ \langle M \rangle \mid L(M) \text{ contains the string “VASSAR”} \}$. $L_{\text{VASSAR}}$ is undecidable.

Proof by reduction from $A_{\text{TM}}$. Suppose that $L_{\text{VASSAR}}$ were decidable and let $R$ be a Turing machine deciding it. We use $R$ to construct a Turing machine deciding $A_{\text{TM}}$. $S$ is constructed as follows:

- Input is $\langle M, w \rangle$, where $M$ is the code for a Turing machine and $w$ is a string.
- Construct code for a new Turing machine $M_w$ as follows:
  - Input is a string $x$.
  - Simulate $M$ on $w$. If $M$ accepts, accept; if $M$ rejects, reject.
- Feed $\langle M_w \rangle$ to $R$. If $R$ accepts, accept. If $R$ rejects, reject.

If $M$ accepts $w$, the language of $M_w$ contains all strings and, thus, the string “VASSAR” if $M$ doesn’t accept $w$, the language of $M_w$ is the empty set and, thus, doesn’t contain the string “VASSAR”. So $R(\langle M_w \rangle)$ accepts exactly when $M$ accepts $w$. Thus, $S$ decides $A_{\text{TM}}$.

But we know that $A_{\text{TM}}$ is undecidable, so $S$ can’t exist. Therefore we have a contradiction, and $L_{\text{VASSAR}}$ must have been undecidable.

**The language $HALEMPTY_{\text{TM}}$**

Let $HALEMPTY_{\text{TM}} = \{ \langle M \rangle \mid M \text{ halts on blank input} \}$. Show $HALEMPTY_{\text{TM}}$ is undecidable.

Proof by reduction from $A_{\text{TM}}$. Suppose that $HALEMPTY_{\text{TM}}$ were decidable and let $R$ be a Turing machine deciding it. We use $R$ to construct a Turing machine deciding $A_{\text{TM}}$. $S$ is constructed as follows:
• Input is \((M, w)\), where \(M\) is the code for a Turing machine and \(w\) is a string.

• Construct code for a new Turing machine \(M_w\) as follows:
  – Input tape is blank.
  – Write \(w\) on the tape.
  – Simulate \(M\) on \(w\). If \(M\) accepts, accept; if \(M\) rejects, reject.

• Feed \((M, w)\) to \(R\). If \(R\) accepts, accept. If \(R\) rejects, reject.

If \(M\) accepts \(w\), then \(M_w\) accepts when started with a blank tape. If \(M\) doesn't accept \(w\), \(M_w\) does not accept when started on a blank tape. So \(R((M, w))\) accepts exactly when \(M\) accepts \(w\). Thus, \(S\) decides \(A_{TM}\).

But we know that \(A_{TM}\) is undecidable, so \(S\) can't exist. Therefore we have a contradiction, and \(HALTEMPTY_{TM}\) must have been undecidable.

The language \(L_{III}\)

Let \(L_{III} = \{ (M) \mid M \text{ prints three } 1\text{s in a row on blank input} \}\). This language is undecidable.

Suppose that \(L_{III}\) were decidable. Let \(R\) be a Turing machine deciding \(L_{III}\). We will now construct a Turing machine \(S\) that decides \(A_{TM}\).

\(S\) is constructed as follows:
• Input is \((M, w)\), where \(M\) is the code for a Turing machine and \(w\) is a string.
• Construct the code for a new Turing machine \(M'\), which is just like \(M\) except that
  – every use of the character 1 is replaced by a new character \(1'\) which \(M\) does not use.
  – when \(M\) would accept, \(M'\) first prints 111 and then accepts
• Similarly, create a string \(w'\) in which every character 1 has been replaced by \(1'\).
• Create a second new Turing machine \(M'_w\) which simulates \(M'\) on the hardcoded string \(w'\).
• Run \(R\) on \((M'_w)\). If \(R\) accepts, accept. If \(R\) rejects, reject.

If \(M\) accepts \(w\), then \(M'_w\) will print 111 on any input (and thus on a blank input). If \(M\) does not accept \(w\), \(M'_w\) is guaranteed never to print 111 accidently. So \(R((M'_w))\) exactly when \(M\) accepts \(w\). Therefore, \(S\) decides \(A_{TM}\).

But we know that \(A_{TM}\) is undecidable, so \(S\) can't exist. Therefore we have a contradiction, and \(L_{III}\) must have been undecidable.

The language \(L_x\)

Let \(L_x = \{ (M) \mid M \text{ writes an } x \text{ at some point, when started on blank input} \}\). This language is undecidable.

Suppose that \(L_x\) were decidable. Let \(R\) be a Turing machine deciding \(L_x\). We will now construct a Turing machine \(S\) that decides \(A_{TM}\).

\(S\) is constructed as follows:
• Input is \( \langle M, w \rangle \), where \( M \) is the code for a Turing machine and \( w \) is a string.

• Construct the code for a new Turing machine \( M_w \) as follows
  - On input \( y \) (which will be ignored).
  - Simulate \( M \) on \( w \), but substituting \( X \) for \( x \), everywhere that \( x \) occurs in \( w \) or the code for \( M \).
  - If \( M \) rejects \( w \), reject.
  - If \( M \) accepts \( w \), print \( x \) on the tape and then accept.

• Run \( R \) on \( \langle M_w \rangle \). If \( R \) accepts, accept. If \( R \) rejects, reject.

If \( M \) accepts \( w \), then \( M_w \) will print \( x \) on any input (and thus on a blank input). If \( M \) rejects \( w \) or loops on \( w \), then \( M_w \) is guaranteed never to print \( x \) accidentally. So \( R \) will accept \( \langle M_w \rangle \) exactly when \( M \) accepts \( w \). Therefore, \( S \) decides \( A_{TM} \).

But we know that \( A_{TM} \) is undecidable, so \( S \) can't exist. Therefore we have a contradiction, and \( L_x \) must have been undecidable.

The language \( L_3 \)

Let \( L_3 = \{ \langle M \rangle \mid |L(M)| = 3 \} \). That is, \( L_3 \) contains all Turing machines whose languages contain exactly three strings. \( L_3 \) is undecidable.

Proof by reduction from \( A_{TM} \). Suppose that \( L_3 \) were decidable and let \( R \) be a Turing machine deciding it. We use \( R \) to construct a Turing machine deciding \( A_{TM} \). \( S \) is constructed as follows:

• Input is \( \langle M, w \rangle \), where \( M \) is the code for a Turing machine and \( w \) is a string.

• Construct code for a new Turing machine \( M_w \) as follows:
  - Input is a string \( x \).
  - Simulate \( M \) on \( w \).
  - If \( M \) rejects \( w \), reject.
  - Otherwise, accept \( x \) exactly when \( x \) is one of the three strings “Vassar”, “College”, or “Poughkeepsie”.

• Feed \( \langle M_w \rangle \) to \( R \). If \( R \) accepts, accept. If \( R \) rejects, reject.

If \( M \) accepts \( w \), the language of \( M_w \) contains exactly three strings. If \( M \) doesn't accept \( w \), the language of \( M_w \) is the empty set. So \( R(\langle M_w \rangle) \) accepts exactly when \( M \) accepts \( w \). Thus, \( S \) decides \( A_{TM} \).

But we know that \( A_{TM} \) is undecidable, so \( S \) can't exist. Therefore we have a contradiction, and \( L_3 \) must have been undecidable.