Monash University
Faculty of Information Technology
2nd Semester 2023

FIT2014
Exercises 9
Undecidability and recursively enumerable languages
SOLUTIONS

1.

We give a mapping reduction from the Diagonal Halting Problem (DHP) to HALT ON EVEN STRINGS.

Let M be a Turing machine provided as input to DHP.

Construct a new TM M' that takes, as input, a string x, and then ignores it and simulates the running of M on input M.

The function that maps M to M' is clearly computable.

If M is in DHP, then it halts on input M. Therefore M' will eventually halt, whatever the input string x was (including for all strings of even length).

If M is not in DHP, then it loops forever on input M. Therefore M' also loops forever, whatever the input string x was (including for all strings of even length).

So the function that maps M to M' is a mapping reduction from DHP to HALT ON EVEN STRINGS.

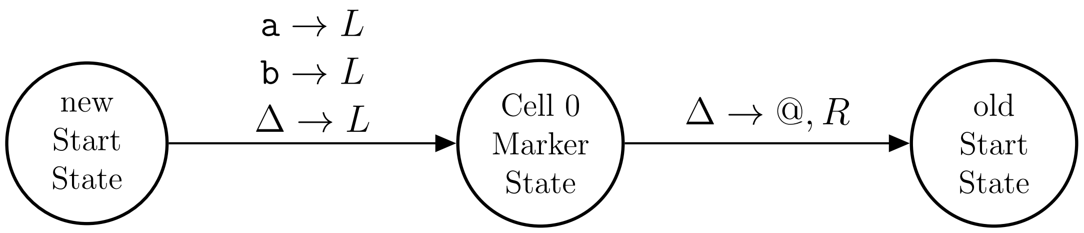
Together with the fact that DHP is undecidable, this implies that HALT ON EVEN STRINGS is undecidable.

2. TM ACCEPTANCE is undecidable, so there can be no mapping reduction from it to any decidable problem. The problem FINITE AUTOMATON ACCEPTANCE is clearly decidable: you can just simulate the FA on the string, to determine if it accepts or not. So there can be no mapping reduction from TM ACCEPTANCE to FINITE AUTOMATON ACCEPTANCE.

3. (a) Let M be a normal Turing machine. Create a new Turing machine, M_2, by modifying M as follows.

1. Add a new symbol to the tape alphabet that wasn't there already; suppose it's @, without loss of generality. (This will be used to mark the cell at position 0, immediately to the left of the leftmost letter of the input string.)
2. Give the Turing machine a Reject state, if it doesn't have one already. (This will be used to simulate the rejection that occurs when the tape head of a normal Turing machine tries to move to the left of the leftmost tape cell.)
3. For each state, add a new transition labelled $@ \rightarrow \Delta, R$ from that state to the Reject State. (This ensures that, if ever we read the @ we placed at tape cell 0, we reject. This is to simulate crashing off the left end of the tape.)

4. Add a new Start State, and another new state we'll call the Cell 0 Marker State. Demote the old Start State so that it is no longer the Start State. Add new transitions involving these new states:



This completes the construction of the TWIT Turing machine M_2.

When M_2 runs, it first moves one cell to the left of the first input letter and marks that cell (which is cell 0) with @, then it goes back to the first letter of the input. Now it is in the old Start State, and computation proceeds as for M. The only thing that might happen that's different to M is if the TM tries to move to the left of cell 1. In the normal TM M, that would be a crash, hence rejection. In M_2, it doesn't crash, but it then reads @ in cell 0. Whatever state it is in, there will be just one transition labelled @ (since @ wasn't in the tape alphabet of M). So, that transition is followed, and this leads to the Reject State, so M_2 rejects. So the final decision is the same as for M, i.e., rejection, even though the mechanism is different (crashing in M, Reject State in M_2). So, for every input string, the two machines have the same behaviour, in the sense that the eventual outcomes (Accept/Reject/Loop) are the same. Furthermore, if the machine halts, the final contents of the positive portions of the tapes of M and M_2 are the same (and the left side of the tape, i.e., all cells numbered 0 or negatively, is entirely blank). It follows that the functions computed by the two Turing machines are the same.

(b) Our mapping reduction is:

1. Input: $\langle M \rangle$, where M is a Turing machine.
2. Construct M_2 from M, as above.
3. Output: $\langle M_2 \rangle$, $\langle M \rangle$.

This is computable, because construction of M_2 from M is computable, by (a). Also,

$$\begin{aligned}
\langle M \rangle \in \text{Diagonal Halting Problem} \quad &\iff \quad M \text{ halts on input } M \\
\iff \quad & M_2 \text{ halts on input } M \quad \text{(because } M \text{ and } M_2 \text{ behave the same, on any input string)} \\
\iff \quad & (\langle M_2 \rangle, \langle M \rangle) \in \text{TWIT Halting Problem}.
\end{aligned}$$

Therefore the function we gave above is indeed the required mapping reduction.

(c) The TWIT Halting Problem is undecidable, because we have given a mapping reduction from a problem we already know to be undecidable (namely, the DHP) to the TWIT Halting Problem.
(d) The TWIT Halting Problem is r.e., because its language is the set of inputs accepted by a TM that takes a pair $(\langle M' \rangle, \langle M \rangle)$, where M' is a TWIT Turing machine and M is a normal Turing machine, and simulates M' running on input $\langle M \rangle$.

4.

Decision Problem	your answer (tick **one** box in each row)	
Input: a Java program P (as source code). Question: Does P contain an infinite loop?	☐ Decidable	✓ Undecidable
Input: a Java program P (as source code). Question: Does P contain the infinite loop `for(int i=0; i>=0; i++);`?	✓ Decidable	☐ Undecidable
Input: a Java program P (as source code). Question: Does P contain recursion?	✓ Decidable	☐ Undecidable
Input: a Java program P (as source code). Question: If P is run, will some method of P keep calling itself forever?	☐ Decidable	✓ Undecidable
Input: a Turing machine M. Question: is there some input string, of length **at most** 12, for which M halts in at most 144 steps?	✓ Decidable	☐ Undecidable
Input: a Turing machine M. Question: is there some input string, of length **at least** 12, for which M halts in at most 144 steps?	✓ Decidable	☐ Undecidable
Input: a Turing machine M. Question: Is the time taken for M to halt even?	☐ Decidable	✓ Undecidable
Input: a Turing machine M. Question: Is there a palindrome which, if given as input, causes M to eventually halt?	☐ Decidable	✓ Undecidable
Input: a Turing machine M. Question: Is M a palindrome?	✓ Decidable	☐ Undecidable

5.
(a)

$$
\begin{aligned}
&27 \to 82 \to 41 \to 124 \to 62 \to 31 \to 94 \to 47 \to 142 \to 71 \to 214 \to 107 \to 322 \to \\
&161 \to 484 \to 242 \to 121 \to 364 \to 182 \to 91 \to 274 \to 137 \to 412 \to 206 \to 103 \to \\
&310 \to 155 \to 466 \to 233 \to 700 \to 350 \to 175 \to 526 \to 263 \to 790 \to 395 \to 1186 \to \\
&593 \to 1780 \to 890 \to 445 \to 1336 \to 668 \to 334 \to 167 \to 502 \to 251 \to 754 \to 377 \to \\
&1132 \to 566 \to 283 \to 850 \to 425 \to 1276 \to 638 \to 319 \to 958 \to 479 \to 1438 \to 719 \to \\
&2158 \to 1079 \to 3238 \to 1619 \to 4858 \to 2429 \to 7288 \to 3644 \to 1822 \to 911 \to 2734 \to \\
&1367 \to 4102 \to 2051 \to 6154 \to 3077 \to 9232 \to 4616 \to 2308 \to 1154 \to 577 \to 1732 \to \\
&866 \to 433 \to 1300 \to 650 \to 325 \to 976 \to 488 \to 244 \to 122 \to 61 \to 184 \to 92 \to 46 \to \\
&23 \to 70 \to 35 \to 106 \to 53 \to 160 \to 80 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1
\end{aligned}
$$

This calculation was done in a Google Sheet, but it could equally well be done in another spreadsheet, or in a Python program, or by a Turing machine, etc.

(b) Consider the algorithm that takes a positive integer x as input and keeps applying f repeatedly, only stopping and accepting if it reaches 1.

This algorithm accepts x if and only if $x \in$ COLLATZ. So, if the algorithm is implemented as a Turing machine M, then Accept(M) = COLLATZ. Therefore COLLATZ is recursively enumerable.

(c) We do not yet know whether or not this algorithm (for (b)) gives a decider, since we do not yet know if it halts for all inputs, and we do not yet know if there is *any* positive integer that does *not* belong to COLLATZ.

If the Collatz conjecture is true, then the algorithm is indeed a decider for COLLATZ, since the truth of the conjecture would imply that the algorithm always halts and accepts. But, in that case, a much simpler decider would work! What is it?

If the Collatz conjecture is false, then it is still possible, in principle, as far as we know at present, that COLLATZ might still be decidable. Why?

(d) This problem has resisted the efforts of some of the best computer scientists and mathematicians in the world. But that should not discourage a student from having a go at it! It needs a fresh idea, a new direction. If you do think you have made significant progress, be sure to also read the two articles cited in the footnote for this question in the tutorial sheet.

6.
($\Rightarrow$)

Let L be a r.e. language. Then there is a TM M such that Accept$(M) = L$.

Let the predicate P take two arguments, a string x (to be considered as input to M) and a positive integer t. It is defined to be True if, when M is run on input x, it accepts in $\leq t$ steps, and False otherwise. This predicate is computable, since to determine whether or not $P(x, t)$ is True, we just run M on input x for t steps, and if it accepts within that time, we know $P(x, t)$ is True, and if it does not (so either it is still going, or has rejected), we know it is False.

Having defined this predicate, we see that $x \in L$ if and only if M accepts x, which holds if and only if there is some time t such that M accepts x in $\leq t$ steps. (Just take t to be any number at least as large as the number of steps that M takes on input x until it halts. We know it must halt for input x, since it accepts.) But this in turn is true if and only if $P(x, t)$ is True.

($\Leftarrow$)[1]

Now suppose that there is a decidable two-argument predicate P such that $x \in L$ if and only if there exists y such that $P(x, y)$ (i.e., such that $P(x, y)$ is True). Let D be a Turing machine which correctly computes $P(x, y)$ for any x, y. Note that D always halts.

We create a new TM M which does the following. Given input string x, we simulate D on input (x, y), for every string y in sequence, and accept x if any one of the y leads to acceptance.

In order to spell this algorithm out a bit more, denote all strings by $y_1, y_2, \ldots, y_i, \ldots$. (E.g., $y_1 = \varepsilon$, $y_2 =$ `a`, $y_2 =$ `b`, $y_2 =$ `aa`, etc.) Our TM M works as follows:

```
1. Input: $x$

2. For each $i = 1, 2, \ldots$
   {
         Run $D$ on $(x, y_i)$.
         If $D(x, y_i)$ stops and returns True, then Accept.
         //    The only other possibility is that $D(x, y_i)$ stops and returns False,
         //    in which case just continue to next iteration.
   }
```

Since D always halts, the computation of $D(x, y)$ for each y never gets stuck in an infinite loop on any y. The series of simulations of $D(x, y)$ always moves on from one y to the next y, eventually.

If $x \in L$, then $\exists y : P(x, y)$, so eventually the simulation reaches some y for which $D(x, y)$ returns True (when $P(x, y)$ is True). So our algorithm accepts x in this case.

If $x \notin L$, then $P(x, y)$ is False for every y. So, for each y considered in our simulation, $D(x, y)$ returns False. So the simulation will go on forever, since the only way it could ever stop is if some y makes $D(x, y)$ True. So our algorithm does not accept x in this case; instead, it loops forever.

Therefore, the set of accepted strings of our algorithm is just L.

We have demonstrated the existence of a Turing machine M such that $\text{Accept}(M) = L$.

This means that L is recursively enumerable.

[1]Thanks to FIT2014 tutor Alejandro Stuckey de la Banda for suggesting a significant improvement to this part of the proof.

7.

(a) $\overline{\text{HALT}}$, i.e., $\{M : M$ loops forever, for input $M\}$

(b) Given a CFG, does it generate *every* string (over its alphabet)?

(c) Given a (multivariate) polynomial, does it have no integer roots?

In each case, we have taken a decision problem/language known to be r.e. but undecidable, and taken its complement. Such a language is co-r.e. by definition, and its also undecidable (since the complement of a decidable language is decidable).

Supplementary exercises

8.

(a) Let f be the function that maps any string x to the string xx obtained by concatenating x with a copy of itself. In other words, f just doubles x. String copying and concatenation are computable operations, so f is computable.

Let x be any string (which may or may not be in L).

If $x \in L$, then $xx \in L^{\text{even}}$, since L is closed under doubling and xx has even length. Therefore $f(x) \in L^{\text{even}}$, since $f(x) = xx$.

If $f(x) \in L^{\text{even}}$, then $xx \in L^{\text{even}}$ (by definition of $f(x)$), and hence $xx \in L$ (since $L^{\text{even}} \subseteq L$). But xx has even length, and so can be halved, giving x, which must then be in L as L is closed under halving.

So f is a mapping reduction from L to L^{even}.

(b) Let y be some specific string not in L. Such a y exists, since L is not universal.

We use the function g defined by

$$g(x) := \begin{cases} x, & \text{if } |x| \text{ is even;} \\ y, & \text{if } |x| \text{ is odd.} \end{cases}$$

Note that y does not depend on x. So, all odd-length strings are mapped to the same string y. But each even-length string is mapped to itself.

This is computable, since determining if a string x has even length is computable, and outputing the constant string y (when needed) is also computable.

If $x \in L^{\text{even}}$, then $|x|$ is even, so $g(x) = x$ (by definition of g), so $g(x) \in L^{\text{even}}$ (since x is), so $g(x) \in L$ (since $L^{\text{even}} \subseteq L$).

If $x \notin L^{\text{even}}$ we have two cases.

If $|x|$ is even, then $g(x) = x$ (by definition of g), so $g(x) \notin L^{\text{even}}$ (since x isn't), so $g(x) \notin L$ (since L^{even} contains all even-length strings in L).

If $|x|$ is odd, then $g(x) = y$ (by definition of g), so $g(x) \notin L$ (since $y \notin L$).

So, regardless of the parity of $|x|$, we have $g(x) \notin L$.

We have shown that g is a mapping reduction from L^{even} to L.

(c) If L^{even} is decidable, then a decider for it, together with the mapping reduction from L to L^{even}, gives a decider for L. So L is decidable.

If L is decidable, then a decider for it, together with the mapping reduction from L^{even} to L, gives a decider for L^{even}. So L^{even} is decidable.

Therefore L is decidable if and only if L^{even} is decidable.

Therefore L is undecidable if and only if L^{even} is undecidable.

(d) Concatenating a string with itself can be done by a Turing machine in quadratic time. (The machine has to repeatedly copy a bit of x to a location n tape cells to its right, then go back to get the next bit. This must be done for each of the n bits of x.) So the time complexity of f is $O(n^2)$.

The function g just has to check the parity of $|x|$ (which can be done by one pass all along the input string) and maybe output the fixed string y, which does not depend on x (and takes constant time). So the time complexity of g is $O(n)$.

9.

Suppose we have a decider that can tell us whether or not a Turing machine is self-reproducing.

Let P be any program. Write $P(x)$ for the program P with hard-coded input x. (So, for example, a statement that reads input and puts the result into a variable `v`, is replaced by a statement that simply assigns the hard-coded string x to the variable `v`.)

We are going to construct, from P, another program which will be self-reproducing if and only if P halts when given itself as input.

To do this, we will make use of the specific self-reproducing program S described in the question. The exact details of how S works need not concern us, but such programs are known to exist. (Look up ‘quines’. Challenge: write a self-reproducing Turing machine!)

The program we construct from P is just $S_{P(P)}$, the self-reproducing program S with the code for $P(P)$ (i.e., P with hard-coded input being P itself) inserted at its special point.[2] Provided P halts on input P, the program $P(P)$ will halt, so that $S_{P(P)}$ will simply reproduce itself. But if P does not halt on input P, then $P(P)$ never halts, so $S_{P(P)}$ never halts. So $S_{P(P)}$ is self reproducing if and only if P eventually halts for input P. So, if we have a decider for self-reproducibility, we can use it to make a decider for the Diagonal Halting Problem. This is a contradiction, since the Diagonal Halting Problem is known to be undecidable. So there cannot exist a decider for self-reproducibility. So self-reproducibility is undecidable.

10.

For any Turing machine P, construct a Turing machine U_P which works as follows.

[2]If P shares any variables etc. with S, just rename them to avoid this.

1. Input: Turing machine M, input string x for M.
2. Simulate P on input P. After P stops (in the simulation), continue.

   (This can easily be done in such a way as to still leave (M, x) sitting on the tape afterwards.)
3. Run U on input (M, x). (It simulates the execution of M with input x.)

If P halts for input P, then U_P is the same as U except for a delay at the start while P is run on input P. So U_P is universal.

If P does not halt on input P, then U_P never halts, no matter what input it is given. So it cannot be universal any more.

So: P halts on input P if and only if U_P is universal.

So, if we had a decider for universality, then we could use it to make a decider for the Diagonal Halting Problem.

Hence there is no decider for universality. So, determining whether or not a Turing machine is outwardly universal is undecidable.