Monash University
Faculty of Information Technology
2^{nd} Semester 2018

FIT2014
Tutorial 6
Decidability, Undecidability and Complexity

Although you may not need to do all the exercises in this Tutorial Sheet, it is still important that you attempt all the main questions and a selection of the Supplementary Exercises.

ASSESSED PREPARATION: Question 1.
You must present a serious attempt at this entire question to your tutor at the start of your tutorial.

---

1. Consider the following four problems.

FINITE AUTOMATON ACCEPTANCE
INPUT: Finite Automaton M, string x.
QUESTION: Does M accept x?

NFA ACCEPTANCE
INPUT: Nondeterministic Finite Automaton M, string x.
QUESTION: Does M accept x?

PDA ACCEPTANCE
INPUT: Pushdown Automaton M, string x.
QUESTION: Does M accept x?

TM ACCEPTANCE
INPUT: Turing machine M, string x.
QUESTION: Does M accept x?

(a) Outline a mapping reduction from NFA ACCEPTANCE to FINITE AUTOMATON ACCEPTANCE.

(b) Does there exist a mapping reduction from PDA ACCEPTANCE to FINITE AUTOMATON ACCEPTANCE? Why or why not?

(c) Does there exist a mapping reduction from TM ACCEPTANCE to FINITE AUTOMATON ACCEPTANCE? Why or why not?

2. If x is any string, we can *double* it by concatenating it with a copy of itself, forming the string xx. If x is any string of even length, then we can *halve* it by dividing it in half exactly, giving two strings x^L and x^R, being the left and right halves, respectively, of x. (So $x = x^L x^R$.)

A language is *closed under doubling* if every string formed by doubling a string in the language is also in the language. A language is *closed under halving* if halving a string in the language always gives two other strings that are also in the language.

Let L be any nonempty non-universal[1] language that is closed under doubling and halving. Let L^{even} be the set of all strings of even length in L.

(a) Prove that there exists a mapping reduction from L to L^{even}.

(b) Prove that there exists a mapping reduction from L^{even} to L.

(c) Prove that L is undecidable if and only if L^{even} is undecidable.

(d) Determine the time complexity, in big-O notation, of the mapping reductions you found in (a) & (b).

(e) Prove that $L \in \text{P}$ if and only if $L^{\text{even}} \in \text{P}$.

3. Prove that the following problem is undecidable, using a mapping reduction from the Diagonal Halting Problem.

HALT ON EVEN STRINGS
INPUT: Turing machine M.
QUESTION: Does M always halt when the input string has even length?

[1]The *universal* language is Σ^*, i.e., the set of all finite strings over the alphabet being used.

4.

For each of the following decision problems, indicate whether or not it is decidable.

You may assume that, when Turing machines are encoded as strings, this is done using the Code-Word Language (CWL).

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; );` ?	☐ 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: Does M halt on an even number of steps?	☐ 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 program is *self-reproducing* if it outputs a copy of itself and then stops. Self-reproducibility is a property of programs such as computer viruses and worms.

Prove that the language of self-reproducing programs is undecidable.

Assumptions you may make:

- All programs under discussion are written in some fixed programming language such as Java.
- The Diagonal Halting Problem, for programs in this language, is undecidable.
- There exists a specific self-reproducing program S with the property that you can insert a code chunk C (of a certain type, to be explained shortly) into S, at some special point in S, so as to get another self-reproducing program S_C which executes C before reproducing itself. The inserted code chunk C is required to be "independent" of all the code in S, in that it has uses no variables, functions, methods, etc. that are used in S.

What does this mean for the possibility of an infallible virus-testing program?

6.

Prove the following theorem (stated in Lecture 18, slide 17):
A language L is r.e. if and only if there is a decidable two-argument predicate P such that:

$$x \in L \text{ if and only if there exists } y \text{ such that } P(x, y). \quad [2]$$

7.

You are running a program with running time an^5, where a is a constant.

(a) Assuming Moore's Law, how long do you have to wait before your program can solve inputs that are four times as large as those you can solve now, in the same time?

(b) Your friend has a program that runs in time 2^n. Give a lower bound on how long she has to wait, under the same scenario. (Assume for this part that your input size is at least 40.)

8.

Suppose you work in bioinformatics, and you have written a program that takes, as input, a DNA string, regarded as a sequence of *bases*, where each base is one of C,G,A,T. Your program runs in time $5n^3$, where n is the length of the input string.

[2] A predicate is said to be *decidable* if its corresponding function — from the set of all its inputs to {True, False}, always correctly indicating whether or not the predicate is True for its input — is computable.

Now, your boss insists on measuring string length in bits, where each of the four bases is represented by two bits.

(a) How would you state your program's running time, if n now represents the number of *bits* in the input string?

(b) You rush off to change the description of the program's running time in the documentation. You discover that, in your team, documentation gives all running times in big-O notation. What changes do you need to make to the big-O statement of the running time?

9.

Suppose algorithm A takes an adjacency matrix[3] of a graph as input and solves a particular graph-theoretic problem Π in polynomial time.

1. Prove that, if the graph is instead given as an edge-list representation[4], then the problem Π can still be solved in polynomial time.

2. Does this mean that the way the input is represented does not matter? (See also Q15.)

10.

Consider 2SAT, discussed in Lecture 19, and 2COL, the set of 2-colourable graphs.

Given a graph G, associate with each vertex v two variables, w_v and b_v, with the following meanings:

w_v : Vertex v is coloured White.

b_v : Vertex v is coloured Black.

Show how to write Boolean expressions, in Conjunctive Normal Form, with each of the following meanings:

- Vertex v gets at least one colour.
- Vertex v does not get more than one colour.
- Vertices v and u do not get the same colour.

Give an algorithm for constructing, from G, a Boolean expression φ in CNF with the meaning: the colouring of G is a proper one (every vertex is coloured White or

[3] An *adjacency matrix* for a connected graph with v vertices is a $v \times v$ matrix whose rows and columns are indexed by the vertices, where the entry in row i and column j is 1 if vertices i and j are adjacent in the graph, and 0 otherwise. So the matrix consists entirely of 0s and 1s. How many bits do you need to represent a graph in this way?

[4] An *edge list* for a graph with v vertices and e edges consists of a list of e pairs, one for each edge. A pair (i, j) represents an edge between vertices i and j. If vertices i and j are not adjacent, then the pair (i, j) does not appear in the list. How many bits do you need to represent a graph in this way?

Black, but not both, and adjacent vertices get different colours). In other words, we require that G is in 2COL if and only if φ is in 2SAT.

11.

Let 2SAT2 be the set of Boolean expressions in CNF, with exactly two literals in each clause (as in 2SAT), such that each variable appears at most twice (which could be as identical literals — both negated, or both unnegated — or could be one negated and one normal).

(a) Prove that every member of 2SAT2 is satisfiable.

(b) How would you work out the number of satisfying truth assignments for such an expression?

Supplementary exercises

12.

Prove that the following problem is decidable.

Input: two regular expressions R_1 and R_2, using an alphabet Σ.
Question: are the languages described by these expressions disjoint?

13.

For any Turing machine U whose input is a pair (e, x) of strings, we say that U is *outwardly universal* if there is a computable function f which takes any Turing machine and produces a string (intended to be an encoding of the TM, which might be in the Code-Word Language, or might use some other scheme) such that, for all Turing machines M and all strings x,

- U halts for input $(f(M), x)$ if and only if M halts for input x, and
- when they halt, they produce the same output.

Observe that this definition is broader than the usual definition of UTMs! This is not only because we allow any computable encoding scheme, and not just the one used in lectures. It is mainly because we do not require step-by-step simulation, but only that the results of the computations be the same.

Prove that determining whether or not a Turing machine is outwardly universal is undecidable.

14.

Give an example of a co-r.e., non-r.e. problem about each of the following:

(a) Turing machines

(b) Context-free grammars

(c) Polynomials and their roots

15.

(a)

Write a simple, naive algorithm which takes a positive integer x as input and outputs the smallest factor of x which is greater than 1 and less than x (if such exists).

For example, if $x = 6$, your algorithm should output 2. If $x = 7$, your algorithm should report that no such factor exists (since the only factors of 7 are 1 and 7).

For this question, you may assume (not quite realistically) that testing integer divisibility takes constant time.

(b) If the integer is given in *unary* (i.e., as a string of x 1s), what is the complexity of the algorithm, in big-O notation, in terms of the input size n?

(c) Does this algorithm run in polynomial time?

(d) If the integer is given in *binary*, what is the complexity of the algorithm, in big-O notation, in terms of the input size n?

(e) Does the algorithm run in polynomial time now?

(f) What would happen if you used some other base $b > 2$?

(g) [Advanced] Now suppose that each positive integer x is encoded by a sequence giving the successive exponents in its unique prime factorisation. So, if $x = 126$, then its prime factorisation is

$$126 = 2 \cdot 3 \cdot 3 \cdot 7 = 2^1 \cdot 3^2 \cdot 5^0 \cdot 7^1$$

so this x is represented by the sequence $(1, 2, 0, 1)$.

(i) How feasible would it be to use your algorithm with this way of encoding integers?

(ii) Write a faster algorithm, based on this representation, and determine its big-O complexity.

16.

Suppose you have a Boolean expression φ in Conjunctive Normal Form (CNF), with exactly two literals in each clause. You want to determine whether or not it is in 2SAT.

(a) Suppose some variable appears only in unnegated form (i.e., if the variable is x, then it only appears as x, and never as $\neg x$). How can you eliminate this variable from consideration, and simplify your expression as a result?

(b) Suppose some variable appears only in negated form (i.e., always as $\neg x$, never as x). How can you eliminate this variable from consideration, and simplify your expression as a result?

(c) How can you simplify the expression when some variable appears twice in a single clause?

(d) If there is a clause with just one literal, what can you do about its truth value?

(e) Write an algorithm, SIMPLIFY, which takes, as input, a Boolean expression φ in Conjunctive Normal Form (CNF), with **at most** two literals in each clause, and repeatedly applies the actions identified in (a)–(d) for as long as possible. The end result should be *either* rejection, if it is discovered that φ is not satisfiable, *or* it outputs

- a Boolean expression φ' which is satisfiable if and only if φ is satisfiable, and such that, in φ', every clause has exactly two literals, and every variable appears both in its normal and negated form, and that these never appear in the same clause.

- a truth assignment to all variables that appear in φ but not in φ', and which satisfies all clauses of φ that were eliminated in constructing φ'.

(f) Prove, by induction on the number of clauses of φ, that if φ has a clause with just one literal, then SIMPLIFY either rejects φ if it is not satisfiable, or outputs a satisfying truth assignment for it and accepts it.

(g) Put it all together to make a polynomial time algorithm for 2SAT.

(h) Would this approach work for 3SAT? Why, or why not?