Monash University
Faculty of Information Technology
2^{nd} Semester 2017

FIT2014
Tutorial 6
Decidability, Undecidability and Complexity

SOLUTIONS

Although you may not need to do all the exercises in this Tutorial Sheet, it is still important that you attempt the first ten exercises and a selection of the supplementary exercises, and that the exercises you attempt cover a variety of problem types.

Even for problems that you do not choose to attempt seriously, you should still give some thought to how to do them before reading the solutions.

1.

(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 $O(n)$.

(e) Recall that, if A and B are languages such that $A \leq_P B$ and $B \in \text{P}$ then $A \in \text{P}$.

Since our two reductions are polynomial-time computable (by part (d)), we have $L \leq_P L^{\text{even}}$ and $L^{\text{even}} \leq_P L$. It follows that $L \in \text{P}$ if and only if $L^{\text{even}} \in \text{P}$.

2.

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.

3.

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

4.

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

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.[1] 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.

5.

($\Rightarrow$)

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.

($\Leftarrow$)

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 $P(x, y)$. Let D be the Turing machine which correctly computes $P(x, y)$ for any x, y. What we'd like to do is simulate D on input (x, y), for every string y in parallel, and accept x if any one of the y leads to acceptance. But this sounds like doing infinitely many things in parallel. So, instead, we adopt a similar strategy to that used when we showed that every r.e. language has an enumerator.

Denote all strings by $y_1, y_2, \ldots, y_i, \ldots$.

Create a new TM M which works as follows:

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

1. Input: x

2. For each $k = 1, 2 \ldots$
   For each $i = 1, \ldots, k$:
   {
   Simulate the next step of the execution of D on (x, y_i).
   If $D(x, y_i)$ stops (in the simulation of it) and returns True,
   then Accept and Stop the whole computation;
   else if $D(x, y_i)$ stops (in the simulation of it) and returns False,
   then skip i in all further iterations.
   }

Here, M accepts x if and only if there is some y_i such that D stops and returns True, for input (x, y_i). But this in turn holds if and only if there exists y_i such that $P(x, y_i)$. By assumption, this holds if and only if $x \in L$.

So Accept$(M) = L$. Therefore L is r.e.

6.

Let $\tau_t(n)$ be the times taken by your program to solve this problem, t years from now, on an input of size n.

At present, we have $\tau_0(n) = an^5$.

According to Moore's Law, processor speed doubles every two years. So, assuming all programs are run on computers whose processor speed over time follows Moore's Law exactly (which is a big simplification, especially for small t),

$$\tau_t(n) = \tau_0(n)/2^{t/2}. \tag{1}$$

Also, observe that, if the input is increased by a factor of k, then (for fixed t) the time goes up by a factor of k^5:

$$\tau_t(kn) = k^5 \tau_t(n). \tag{2}$$

(a)

The condition given in this question can be written in an equation:

$$\tau_t(4n) = \tau_0(n).$$

We want to solve this for t.

Using (1), the equation becomes

$$4^5 \tau_t(n) = \tau_0(n).$$

Then, using (2) gives

$$\frac{4^5 \tau_0(n)}{2^{t/2}} = \tau_0(n),$$

so that

$$2^{t/2} = 4^5.$$

Solving this gives $t = 20$. So you have to wait for 20 years.

(b)

Now, $\tau_0(n) = 2^n$.

Equation (1) still holds.

Equation (2) no longer holds. Instead we have

$$\tau_t(kn) = \frac{\tau_0(kn)}{2^{t/2}} = \frac{\tau_0(n)^k}{2^{t/2}}.$$

The condition for (b) is once again

$$\tau_t(4n) = \tau_0(n).$$

Doing the appropriate substitutions gives

$$\frac{\tau_0(n)^4}{2^{t/2}} = \tau_0(n).$$

This simplifies to

$$\frac{\tau_0(n)^3}{2^{t/2}} = 1,$$

which in turn gives

$$\tau_0(n)^3 = 2^{t/2}.$$

Substituting $\tau_0(n) = 2^n$ gives $2^{3n} = 2^{t/2}$, so $t = 6n$. Using our lower bound, $n \geq 40$, we find $t \geq 240$. So your friend has to wait for at least 240 years.

This illustrates how even a large polynomial time complexity (and, in most practical contexts, running time n^5 would be considered slow) gives much more computational power, and handles increases of input size much better, than an exponential time complexity.

Thanks to Han Phan (FIT2014 tutor, 2013) for spotting a bug in an earlier version of this solution, and fixing it.

7.

(a)

If m is the length of the input string in bases, then we are given that the running time is $5m^3$. Each base needs two bits to specify it, so $n = 2m$ and $m = n/2$. Therefore the running time is $5m^3 = 5(n/2)^3 = \frac{5}{8}n^3$.

(b)

No change is needed. Big-O running time absorbs constant factors.

8.

An adjacency matrix is an $n \times n$ matrix of bits, so its length, as an input, is n^2 bits.

If algorithm A solves problem Π in polynomial time, then there exists k such that its time complexity is

$$\begin{aligned} &= O(\text{ (length of string representing } \textit{adjacency matrix})^k \text{ }) \\ &= O((n^2)^k) \\ &= O(n^{2k}). \end{aligned}$$

We will now derive an inequality that relates the lengths of adjacency matrices and edge lists.

We use a *lower* bound for the length of the edge list.

Suppose the graph has m edges. Then the edge list consists of m pairs (i, j). For each edge, the pair (i, j) specifies its two endpoints i and j.

The total length of the edge list is clearly $\Omega(m)$. Since $m = \Omega(n)$ (since the graph is connected), we find that the length of the edge list is $\Omega(n)$, which means that

$$n = O(\text{ (length of string representing edge list) }). \tag{3}$$

So the time complexity is:

$$\begin{aligned} &= O(n^{2k}) \\ &= O(\text{ (length of edge list)}^{2k} \text{ }) \\ &\quad \text{(using (3)).} \end{aligned}$$

Since the exponent $2k$ is fixed (i.e., does not depend on n), this shows that the algorithm still runs in polynomial time when the input is represented as an edge list.

A good exercise is to do the reverse problem: show that, if a problem on graphs can be solved in polynomial time when the input is given as an edge list, then it can also be solved in polynomial time when the input is instead given as an adjacency matrix.

The way input is represented does affect how quickly algorithms run in practice, so it is important. But most "reasonable" representation schemes should not affect the classification of problems as polynomial-time solvable or not.

9.

(a) Vertex v gets at least one colour: $w_v \vee b_v$

(b) Vertex v does not get more than one colour: $\neg w_v \vee \neg b_v$

(c) Vertices u and v do not get the same colour: $(\neg w_u \vee \neg w_v) \wedge (\neg b_u \vee \neg b_v)$

Given G, create φ as follows. Firstly, take the conjunction (over all v) of all clauses of type (a); secondly, take the conjunction of all clauses of type (b); thirdly, take the conjunction, over all edges uv of G, of all expressions of type (c). Then take

the conjunction of (a), (b), (c) together. Then you have a single expression φ, in CNF, with exactly two literals in each clause, and it is satisfiable if and only if G is 2-colourable.

10.

(a)

Let φ be a Boolean expression in CNF with exactly two literals in each clause such that each variable appears at most twice.

If a variable appears either only in normal form, or only in negated form, then there is nothing to be lost by making it True or False, respectively. It can be eliminated from any clause where it appears. (This covers the case where a variable appears just once in φ.)

So we may assume that each variable appears exactly twice, once normal and once negated. Furthermore, if these two literals are in the same clause, then that clause is satisfied, and has no interaction with any other clause (since the variable appears nowhere else), so the clause may be eliminated. So we now assume that the two occurrences of each variable are in different clauses.

Draw a graph on all the literals in φ, as follows. The literals are the vertices. Put a green edge between two members of the same clause, and a red edge between two literals which have the same variable. By the assumptions we've been able to make on the way, no two vertices are joined by both a red and a green edge, and every vertex is incident with one red edge and one green edge. Since each clause has exactly two literals, the graph has no loops.

This graph must be a disjoint union of circuits, with all the circuits of even length. For each such circuit, mark every second vertex. This marks exactly one vertex on each green edge (and exactly one on each red edge, too). These are the literals which we will make True. We do so by taking its variable, and making the variable True if this literal is just the variable, and False if the literal is the negation of the variable. The result is that, for each green edge — i.e., for each clause — one of the vertices is marked, meaning that one of the literals in the clause is True. So the whole expression is satisfied.

So every member of 2SAT2 is satisfiable.

(b)

When a literal is True, so its vertex is marked, the neighbour along a red edge — the other literal with the same variable — cannot be marked, since that literal must be False. But then *its* neighbour along a green edge must be True, else the clause corresponding to that green edge is not satisfied. This continues all the way round the circuit. Similarly, if a literal is False, its green neighbour must be True (else that clause is not satisfied), and again we can continue all around the circuit.

So there are only two possible satisfying truth assignments to the variables appearing in a single circuit.

Disjoint circuits are independent for our purposes. So the total number of satis-

fying truth assignments is 2^c, where c is the number of disjoint circuits in the graph we have constructed. (This is if we consider the simplified Boolean expression which only contains variables appearing in both normal and negated form, where the two occurrences of each variable are always in different clauses. Additional variables in the original, unsimplified expression may increase this total number further.)

Supplementary exercises

11.

Let L_1 and L_2 be the languages described by R_1 and R_2 respectively. Observe that

$$L_1 \cap L_2 \text{ are disjoint} \iff L_1 \cap L_2 = \emptyset \iff \overline{L_1 \cap L_2} = \Sigma^* \iff \overline{L_1} \cup \overline{L_2} = \Sigma^*.$$

We can find a regular expression for the language $\overline{L_1} \cup \overline{L_2}$ as follows.

1. Find a FA, A_1, for R_1, using the usual method to convert a regexp to a FA that recognises the same language.

2. Create a new FA, A_1', from A_1 by changing Final states to non-Final and vice versa, and leaving everything else unchanged. This new FA A_1' recognises the complement, $\overline{L_1}$, of the language L_1 recognised by A_1.

3. By the same process as used in the first two steps, create a new FA, A_2', which recognises $\overline{L_2}$.

4. Now we create a NFA to recognise $\overline{L_1} \cup \overline{L_2}$. To do this, we take our two FAs, A_1' and A_2', and identify (i.e., merge into one) their Start states. The rest of the two FAs — their states and transitions — are kept separate. So the new automaton has a single Start state, and at that state there is some nondeterminism as each alphabet letter now has two transitions going out of that state. This NFA recognises $\overline{L_1} \cup \overline{L_2}$.

5. Convert this NFA to a FA using the usual method for converting an NFA to FA.

6. We now have a FA (call it A) to recognise $\overline{L_1} \cup \overline{L_2}$. We need to see if the language A recognises is just Σ^*. In other words, we need to see if it accepts every possible string over our alphabet. In other words, we need to see if it recognises the same language as the trivial FA with a single state, serving both as a Start and a Final state, and with a loop at that state for each letter in the alphabet Σ (since this trivial FA accepts every string).

To do this, we use the algorithm for minimising states in an FA (Lecture 8, slides 12–16). This algorithm is guaranteed to find an FA which recognises the same language and does so with the minimum possible number of states. So, if the language recognised by A is indeed Σ^*, then the FA minimisation algorithm applied to A will produce the trivial FA that accepts Σ^*. But if the language recognised by A is something else, then the FA minimisation algorithm gives some other FA instead of our trivial one.

7. If we end up with the trivial FA by this process, we answer Yes, otherwise we answer No.

What we have given is the outline of an algorithm, and it solves our decision problem (stopping and giving the correct answer in all cases). So that problem is decidable.

It may appear that this fact follows from the closure of regular languages under complement and union. Certainly, these closure properties (along with Kleene's Theorem) tell us that the language $\overline{L_1} \cup \overline{L_2}$ does indeed have a FA that recognises it, and if we had such an FA, we could simplify it to see if it is trivial and therefore accepts every string. But our task in this question was to give a *method* to determine, from R_1 and R_2, whether their languages are disjoint. So we needed to actually *construct* the FA for $\overline{L_1} \cup \overline{L_2}$, not to just rely on its existence.

Having said that, the proofs of the closure properties should give us the methods we need. It's just that *the closure properties themselves* just assert *existence*. (E.g., closure under union asserts that, if L is regular then so is $\overline{L}$, which tells us that *there exists* a regular expression and a FA for $\overline{L}$.)

So it's the *algorithms that come with the closure properties* that we need, rather than just the raw closure properties themselves. (Of course, we use Kleene's Theorem too.)

12.

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

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.

13.

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

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

(c) Given a 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).

14.

(a)

Input: x

$z := 1$ // *z is smallest factor of x found so far.*

For $y := 2$ to $\lfloor\sqrt{x}\rfloor$ // *y is the number to be tested to see if it is a factor.*

// *Smallest factor must be $\leq \sqrt{x}$.*

{

If y is a factor of x, then put $z := y$

}

If $z > 1$ then output z, else report that x is prime.

(b)

In the worst case, the number of loop iterations is $\sqrt{x}$. Each such iteration requires one divisibility test, a comparison, an increment, and possibly an assignment. We'll call this a constant number of steps, which is sweeping a lot under the carpet but it will do for the purposes of this exercise.

So we'll take the complexity to be $O(\sqrt{x})$. If we write n for the input size then, since x is in unary, $n = x$. So the complexity is $O(\sqrt{n}) = O(n^{1/2})$.

(c)

This complexity is polynomial, since it is $O(n^k)$ for some constant k (in this case, $k = \frac{1}{2}$).

(d)

If x is in binary, then its length as a string of bits is $\lfloor \log_2 x \rfloor + 1$. This is close to $\log_2 x$, and we'll use this approximation for now.

Writing n for the input size again, we now have $n = \log_2 x$, so $x = 2^n$. Substituting this into our complexity gives $O(x^{1/2}) = O((2^n)^{1/2}) = O(2^{n/2})$.

(e)

The complexity is now exponential in the input size. The algorithm does not run in polynomial time.

(f)

If we use base $b > 2$, then input size is $n \approx \log_b x$ and $x = b^n$. The complexity is $O(b^{n/2})$. This also is exponential time, not polynomial time.

(g)

(i)

It would be very difficult. Even something as simple as an increment would take a long time.

(ii)

In this encoding, a number is prime if and only if its encoding consists of a single 1, preceded by a (possibly empty) sequence of 0s. Its representation has the form $(0, 0, \ldots, 0, 1)$. We just have to check, for every position in the input sequence except the last, whether or not it is 0, and then we check that the last is 1. This takes constant time per position, and the number of positions is $O(n)$, where n is the input length. So the time complexity is $O(n)$, which is linear time, and certainly polynomial time.

Incidentally, the input length n is the number of prime numbers $\leq x$. This number is approximately $x/\log x$, according to the Prime Number Theorem. But we don't need to know this for this question, since we are expressing complexity in terms of the input length n, and not in terms of the number x itself.

15.

(a)

Put $x =$ True, since there is nothing to be lost by doing so, and remove every clause in which it appears. The new, smaller Boolean expression is satisfiable if and only if the original one is.

(b)

Put $x =$ False, for similar reasons, and remove every clause in which the literal $\neg x$ appears, since the negated literal equates to True.

(c)

If a clause contains two identical literals, just replace it by a clause containing one copy of that literal, since $x \vee x = x$ and $\neg x \vee \neg x = \neg x$.

If a clause contains two opposite literals (i.e., $x \vee \neg x$, for some variable x), then eliminate that clause, since it must be satisfied, no matter what truth value is assigned to x.

(d)

Make that literal True. So, if it is x (for some variable x), then put $x =$ True, and if the literal is $\neg x$, then put $x =$ False. Then, eliminate any clause containing the literal (since it is True so every such clause is satisfied). Also, remove every occurrence of the opposite literal (but don't remove the clauses they are in, since they still need to be satisfied).

(e)

Algorithm: SIMPLIFY

Input: φ

If any clause contains two identical literals, replace it with a clause containing just one copy of that literal.

If any clause contains two opposite literals in the same variable — i.e., both x and $\neg x$ — then eliminate that clause, as it's always satisfied, whatever truth value is given to that variable.

Loop:

{

While there is some variable x that appears only positively
(i.e., only as x, never negated) or only negatively (i.e., only as $\neg x$) in φ:
Set x to True or False respectively,
Eliminate from φ all clauses in which it appears.
(Eliminated clauses are satisfied.)
If any clause is empty, then Reject. (An empty clause cannot be satisfied.)
If φ has a clause with just one literal:
Assign a truth value to the variable to make that literal True.
Eliminate every clause in which this literal appears.
Remove every occurrence of the opposite literal
(i.e., the one with the same variable but opposite truth value).

} **until** no more changes occur to φ.

Output φ, together with the truth assignments made so far.

If φ now has no clauses, then Accept it (as all its clauses have been eliminated, and so satisfied).

Once the loop finishes, if φ is not empty (i.e., it has at least one clause), then we know that each clause has exactly two literals (in two different variables), and each variable appears both positively and negatively (but in different clauses).

(f)

Let c be the number of clauses of φ.

Inductive basis: if $c = 1$, then φ has a single clause, with just one literal. The algorithm SIMPLIFY enters the main outer loop, then in its first iteration, enters the inner loop (While ...) as its condition is satisfied, and the body of that loop makes this literal True (by appropriate truth assignment to its variable, then eliminates this single clause, making φ empty. That completes execution of the While loop. The next statement (If ...) is not done, as there is no empty clause (in fact, no clause at all). The next statement (also If ...) is also skipped, as there is no clause. The final exit-loop condition (until ...) is not yet met, as φ changed. The main outer loop will be done one more time, but nothing will be done in it, and at the end of that iteration, φ will not have changed, so we exit the loop, then output the now-empty φ together with the single truth assignment we made, then Accept it as it has no clauses.

Inductive step:

Assume the assertion of (f) is true whenever the number of clauses is $< c$, and let φ be a Boolean expression of the required type with exactly c clauses.

Consider the first iteration of the outer loop of SIMPLIFY. Rejection, at the first If statement, can only happen if φ is unsatisfiable, in which case we are done. So suppose we don't reject at this stage. The While loop and the first If statement ensure that, by the time we get to the second If statement, we have an expression with no empty clauses, and which has every variable appearing both positively and negatively. If there is no one-literal clause, then we are done already. So suppose there is a one-literal clause. Then the condition of the second If statement is met. The variable in this literal (call it x) is then given the appropriate truth value, to satisfy this clause, and the clause is eliminated. Now, since every variable appears both positively and negatively, there is some clause containing the opposite form of x. This literal is False, so is removed from that clause. That leaves a new one-literal clause in the modified φ. The literal in that clause cannot involve x, else this clause would have had two literals with the same variable (in the previous φ, just before these latest modifications). So it involves another variable, to which we have not yet assigned a truth value.

Since φ has just changed, the condition on the outer loop (until ...) is not satisfied. So we go back to the start of the outer loop again. We are, in fact, in the same situation as applying SIMPLIFY to this modified φ. But the modified φ has fewer clauses than the φ we started with. So we can apply the inductive hypothesis, which tells us that the algorithm will complete with either a rejection or by finding a satisfying truth assignment for φ and outputing it. This, together with the truth assignments for the variables that were eliminated in the process of modifying φ, gives a satisfying truth assignment for the original φ.

This completes the proof of the inductive step. So, by the Principle of Mathematical Induction, the assertion of (f) holds.

(g)

Algorithm for 2SAT:

Input: φ

Apply SIMPLIFY to φ.

If we have not yet stopped, then (by (f)) we know that φ has exactly two literals, with different variables, in each clause.

Pick any variable x. It must appear both positively and negatively.

Put $x =$ True:

$\varphi' := \varphi$

Eliminate from φ' each clause in which the literal x appears. (This must happen at least once.)

Remove from φ' every literal $\neg x$. This must happen at least once, and will lead to at least one clause with just one literal, so (f) is applicable.

Run SIMPLIFY on φ'.

If it accepts, then we have a satisfying truth assignment for φ, by (f): we just take the satisfying truth assignment for φ', and augment it with $x =$ True. So we Accept.

If it rejects, then we know $x =$ True cannot lead to a satisfying truth assignment for φ. So, we try the other possibility.

Put $x =$ False:

$\varphi' := \varphi$

Eliminate from φ' each clause in which the literal $\neg x$ appears. (This must happen at least once.)

Remove from φ' every literal x. This must happen at least once, and will lead to at least one clause with just one literal, so (f) is applicable.

Run SIMPLIFY on φ'.

If it accepts, then we have a satisfying truth assignment for φ, by (f): we just take the satisfying truth assignment for φ', and augment it with $x =$ False. So we Accept.

If it rejects, then we know that neither $x =$ True nor $x =$ False can lead to a satisfying truth assignment for φ. So we Reject.

(h)

This approach will not work for 3SAT because, when a literal is removed from a clause of size 3, we have two remaining literals and we don't know which one has to be True in order for the whole expression to be satisfied. We can extend the algorithm by branching here, treating each of the two remaining literals in turn. This leads to an exponential-time algorithm.