Monash University
Faculty of Information Technology
2nd Semester 2023

FIT2014
Exercises 8
Decidability and Mapping Reductions

SOLUTIONS

1.
(a) There are many ways to do this. Here's one. We use the following pairs of letters to specify, for a node, what kinds of children it has.

letter pair	interpretation
`ab`	node only has a left child
`ba`	node only has a right child
`aa`	node has two children
`bb`	node has no children, i.e., it's a leaf.

To encode a tree, we just visit the nodes in breadth-first order and represent each node by its letter pair.

For example, consider the binary tree on six nodes where the root has two children, the root's left child has a right child only, and the root's right child has two children. No other nodes have children.

Using the above scheme, this tree is encoded by the string `aabaaabbbbbb`.

(b) Each node requires exactly two letters, so the string has length $2n$.

(c) The Turing machine needs to check each non-overlapping pair of letters in turn (so it checks first and second letter, then checks third and fourth letter, etc; it does *not* check second and third letter). For each pair, it tests if the pair is `ab` or `ba`. If that ever happens, then we have a node with one child and we reject. If we never get one of those two letter pairs, then every node either has two children or is a leaf, so the tree is full and we accept.

(d) The Turing machine can just visit every letter in turn, always moving right, using appropriate states to keep track of what it needs to know at any time (since the only information it needs to know is whether it is in an even or odd position on the tape and, if the former, what the previous letter was). It may stop early, to reject. If it doesn't reject, it will go all the way to the end of the input string. It only knows that it has visited the entire encoded tree when it reaches the blank just beyond the input string; then it takes one more step to enter the Accept state. So the number

of steps taken is $\leq 2n + 1$ (since the encoded string has two letters for each of the n nodes of the tree).
(e) Since $\ell = 2n$ (see (b)), the time taken is $\leq \ell + 1$, by (d).

2. [1]

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 put them together as follows. First, create a new Start State separate from both FAs. Then create two new empty transitions from the new Start State, one to the Start State of A_1' and the other to the Start State of A_2'. Then we demote the Start States of A_1' and A_2' so that they are no longer Start States but otherwise function as before. The rest of the two FAs — their states and transitions — are unchanged and separate from each other. So the new automaton has a single Start state, and at that state there is some nondeterminism due to the two empty transitions going out of it. 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).

[1] Thanks to FIT2014 tutor Rebecca Young for improvements to this solution.

To do this, we use the algorithm for minimising states in an FA (Lecture 10, slides 5–10). This algorithm is guaranteed to find an FA which recognises the same language and does so with the minimum possible number of states.[2] 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, and we can easily recognise that it is different.[3]

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.

Alternative to Steps 6–7:

6. We now have a FA (call it A) to recognise $\overline{L_1} \cup \overline{L_2}$. Change it to a FA (call it $\overline{A}$) to recognise the complementary language $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$. (This can be done by changing Final States to non-Final States and vice versa.) We need to see if it recognises the empty language. In other words, we need to determine if it belongs to FA-Empty. We can do this using the method described in Lecture 20, slides 9–10.

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.)

[2] In general, to ensure that the simplified FA has the minimum possible number of states, you also need to remove any unreachable states. But the FA we have constructed here will not have any unreachable states.

[3] In general, recognising if two FAs with the same number of states are really the same FA (i.e., they define the same language, and differ only in the labelling of states) is a decidable problem. It's closely related to the famous GRAPH ISOMORPHISM problem which we meet later in the semester. In general, it's not known to be solvable in polynomial time (a concept we meet in Lecture 24). But, in the case of interest here, one of the FAs being compared has just one state, so this test is easy.

3. [4]

(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. Any clause containing it can therefore be satisfied, without affecting anything else. So those clauses can be eliminated, and we can focus on the smaller expression formed from the remaining clauses. (This step 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.[5] 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.

[4] Thanks to FIT2014 student Amy Liu for a correction and improvement to this solution.

[5] A **circuit** in a graph is a closed path in which no vertex or edge is repeated anywhere on the path. A **disjoint union of circuits** is a collection of circuits that have no vertices or edges in common.

Disjoint circuits are independent for our purposes. So the total number of satisfying 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.)

4.

If we don't use colour Black, then all clauses with literals of the form $\neg \mathtt{v}i\mathtt{Black}$ are no longer needed. (There is no longer any need to say that a vertex cannot be both Black and Red, or Black and White, or to say that an edge cannot have both its endpoints coloured Black.)

The only other clause of the original expression φ_G that referred to Black are the ones that say that each vertex gets at least one colour. Those clauses had the form

$$\mathtt{v}i\mathtt{Red} \vee \mathtt{v}i\mathtt{White} \vee \mathtt{v}i\mathtt{Black}.$$

These were in fact the only clauses of size 3 in φ_G. Restricting to colours Red and White means we only need clauses of size 2 for this now:

$$\mathtt{v}i\mathtt{Red} \vee \mathtt{v}i\mathtt{White}.$$

This means the new φ_G has all clauses of size 2. So the mapping reduction now reduces 2-COLOURABILITY to 2-SAT.

5.

(a) Given M, x as input to NFA ACCEPTANCE, convert M to an equivalent FA; let's call it M'. (See Kleene's Theorem.) Then M accepts x if and only if M' accepts x. In other words, (M, x) is a Yes-input to NFA ACCEPTANCE if and only if (M', x) is a Yes-input to FINITE AUTOMATON ACCEPTANCE. Furthermore, the construction $(M, x) \mapsto (M', x)$ is computable. So, it's a mapping reduction from NFA ACCEPTANCE to FINITE AUTOMATON ACCEPTANCE.

(b) Given M, x as input to FINITE AUTOMATON ACCEPTANCE, convert M to an equivalent TM (see Lecture 18, slide 14); let's call it M'. Then M accepts x if and only if M' accepts x. In other words, (M, x) is a Yes-input to FINITE AUTOMATON ACCEPTANCE if and only if (M', x) is a Yes-input to TM ACCEPTANCE. Furthermore, the construction $(M, x) \mapsto (M', x)$ is computable. So, it's a mapping reduction from FINITE AUTOMATON ACCEPTANCE to TM ACCEPTANCE.

(c) The problem PDA ACCEPTANCE is decidable. Any decidable problem has

a mapping reduction to (almost[6]) anything else.[7] So, in particular, PDA ACCEPTANCE has a mapping reduction to FINITE AUTOMATON ACCEPTANCE. We'll explain how that happens in this particular case.

Let D be any decider for PDA ACCEPTANCE. (For example, D could just simulate the execution of the PDA M on input x, and see if it accepts or not.) Pick any FA A and any string y such that A accepts y. Also pick any FA B and any string z such that B *rejects* z. (Note that A, y, B, z are all *fixed* in advance, before we even look at any input to PDA ACCEPTANCE.)

The mapping reduction from PDA ACCEPTANCE to FINITE AUTOMATON ACCEPTANCE works as follows.

1. Input: a PDA M, and a string x.
2. Run D on the pair (M, x) to determine if M accepts x.
   (a) If D reports that M accepts x: OUTPUT (A, y).
   (b) If D reports that M rejects x: OUTPUT (B, z).

If (M, x) is a Yes-input to PDA ACCEPTANCE, then D reports that M accepts x, and therefore the algorithm outputs the fixed pair (A, y), which is a Yes-input to FINITE AUTOMATON ACCEPTANCE.

On the other hand, If (M, x) is a No-input to PDA ACCEPTANCE, then D reports that M rejects x, and therefore the algorithm outputs the fixed pair (B, z), which is a No-input to FINITE AUTOMATON ACCEPTANCE.

Finally, the algorithm is clearly computable.

Keep in mind, though, that a mapping reduction *from* a decidable language to another language is not useful in practice. If we already know how to decide the language, there's nothing much to be gained by reducing it to something else. See Lecture 21, slides 16–17.

(d) It tells us that NFA-ACCEPTANCE is decidable. We can mapping-reduce NFA-ACCEPTANCE to a decidable language, so NFA-ACCEPTANCE is decidable too.

(e) Nothing! A decidable language can be mapping-reduced to (almost) anything, including any undecidable language. Such a mapping-reduction tells us nothing at all about the language it is being reduced to.

6.

(a) Yes, because given any CFG we can construct a grammar in CNF that generates the same language, except that the empty string needs to be treated as a special case

[6] What are the exceptions?

[7] This suggests another approach to (b).

(since a CNF grammar cannot generate the empty string).

Before we give the construction, we need to fix a couple of specific grammars in advance.

Pick any (A, y) where A is a Chomsky Normal Form CFG that does not generate the empty language, and y is a string generated by A. Also pick any (B, z) where B is a Chomsky Normal Form CFG that does not generate the universal language, and z is a string that is NOT generated by B.[8] (Note that A, y, B, z are all *fixed* in advance, before we even look at any input to CFG GENERATION.)[9]

Now we can describe the mapping reduction.

1. Input: a CFG G, and a string x.
2. If x is nonempty:
   (a) convert G to a CNF CFG H.
   (b) OUTPUT (H, x)

   else: // $x = \varepsilon$
   (a) Apply the Nullability algorithm (Lecture 16, slide 14) to determine if G generates ε.
      i. If it does: OUTPUT (A, y).
      ii. If not: OUTPUT (B, z).

(b) Yes, because we can just use the mapping reduction that does nothing: for every input (G, x), the output is also (G, x). This works because a CNF CFG is still a CFG.

[8] Thanks to FIT2014 student Stefan Sudar for spotting and reporting an error.

[9] An alternative approach would be to pick a CNF CFG A which generates a language which is neither empty nor universal. Then you would pick y to be any fixed string generated by A and z to be any fixed string not generated by A. Then you wouldn't need B, and would use A instead of it later on.

Supplementary exercises

7.

(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.