Monash University
Faculty of Information Technology
2nd Semester 2023

FIT2014
Exercises 11
Complexity: NP-completeness

SOLUTIONS

1.

(a)

Input: φ

Create a new variable x that does not appear in φ.

Form φ' by adding a new singleton clause, consisting just of the literal x, to φ:

$$\varphi' := \varphi \wedge (x).$$

Output: φ'

(b)

Assume φ is satisfiable. Then it has a satisfying truth assignment f. We can extend f to a new truth assignment, f', by assigning True to x. So,

$$f'(y) = f(y) \quad \text{for all variables } y \text{ in } \varphi;$$
$$f'(x) = \text{True}.$$

This new truth assignment f' satisfies all clauses in φ because, on those clauses, it agrees with f which is a satisfying truth assignment for φ. Also, f' satisfies the new singleton clause because it sets x to True. So f' satisfies φ'. So φ' is satisfiable.

Conversely, suppose φ' is satisfiable. Then it has a satisfying truth assignment f'. This truth assignment must satisfy all clauses of φ, since they are also clauses in φ'. So φ is satisfiable.

(c)

You need to prove that it runs in polynomial time.

The question doesn't require you to give such a proof. But, if it did, you could reason as follows.

Given φ, we need to pick a variable not in φ. We can scan along φ to compile a list of variables used, and then pick a string representing a variable that is not in this list, which could be one whose name or number is greater than (or longer than) any variable name appearing in φ. The time taken to do this is dominated by the time taken to search all the way along a list, which is polynomial time.

Then we need to add a new singleton clause. This requires appending a new clause to the current list of clauses of φ. This also takes polynomial time.

The exact details of how long these things take will depend on how we represent variables and clauses and the specific data structures we use to represent φ. But we do not need to worry about these details if we are just showing that something takes polynomial time.

(d)

You need to prove that it belongs in NP.

The question doesn't require you to give such a proof. But, if it did, you need to propose a certificate (which would be a truth assignment) and a verifier, which just checks if (i) the truth assignment is satisfying, which would work the same as a verifier for SAT, and (ii) that there is a singleton clause; then you show that it is indeed a verifier for the language, and that it runs in polynomial time.

2.

(a) It is NP-complete.

(b) Yes, it's still in NP. The polynomial-time verifier is like the one in (a), except that in the second-last line we test if `numberOfSatisfiedClauses` $\geq m - k$, rather than just $\ldots \geq m - 1$.

The polynomial-time reduction from SAT introduces k new variables, $y_1, \ldots, y_k$, and $2k$ new clauses, with each clause containing just one literal, y_i or $\neg y_i$ ($i = 1, 2, \ldots, k$). This gives a polynomial-time reduction from SAT to NEARLY SAT.

This problem is also NP-complete.

3.

(a)

Given a graph G, let the certificate be a Hamiltonian circuit of G. This can be verified in polynomial time, by checking that the circuit is indeed a circuit and that it visits each vertex exactly once.

(b)

Polynomial-time reduction from HAMILTONIAN PATH to HAMILTONIAN CIRCUIT:

Given a graph G (which may or may not have a Hamiltonian path), construct a new graph H by adding a new vertex v and joining it, by n new edges, to every vertex of G. (Here, n denotes the number of vertices of G.)

We show that G has a Hamiltonian path if and only if H has a Hamiltonian circuit.

Suppose G has a Hamiltonian path. Call its end vertices u and w. Then a Hamiltonian circuit of H can be obtained by adding the new vertex v, and the edges uv and wv, to the Hamiltonian path. So H has a Hamiltonian circuit.

Conversely, suppose H has a Hamiltonian circuit C. This circuit must include v, and two edges incident with v. Let u and w be the two vertices of G that are incident with v in C. (So C includes the edges uv and wv as well as the vertex v.) The rest of C must constitute a Hamiltonian path between u and w in G. So G has a Hamiltonian path.

This completes the proof that G has a Hamiltonian path if and only if H has a Hamiltonian circuit.

It remains to observe that the construction of H from G can be done in polynomial time. Therefore the construction of H from G is a polynomial-time reduction from HAMILTONIAN PATH to HAMILTONIAN CIRCUIT.

(c)

Since HAMILTONIAN PATH is NP-complete, and it is polynomial-time reducible to HAMILTONIAN CIRCUIT, and HAMILTONIAN CIRCUIT is in NP, we can conclude that HAMILTONIAN CIRCUIT is NP-complete.

4.

(a)

A graph is bipartite if and only if it is 2-colourable: just associate a different colour to each of the two parts of the partition. So BIPARTITE GRAPH COLOURING is in P, since 2-COLOURABILITY is in P.

(b)

It is routine to show that it is in NP: given (G, k, b) (the input) and an assignment of $\leq k$-colours to the vertices (the certificate), we can check all edges to see whether or not they are bad under this assignment of colours, and keep track of the number of bad edges as we go. At the end, we can easily check that the total number of bad edges is $\leq b$. Then note that all this takes polynomial time.

BAD GRAPH COLOURING can be shown to be NP-complete by reduction from GRAPH COLOURING, since GRAPH COLOURING is just a special case of BAD GRAPH COLOURING when $b = 0$.

The reduction just maps $(G, k) \mapsto (G, k, 0)$. We have

$$
\begin{aligned}
(G,k) \in \text{GRAPH COLOURING} \quad &\iff \quad G \text{ is } k\text{-colourable} \\
&\iff \quad G \text{ has an assignment of } \leq k \text{ colours to vertices} \\
&\qquad\quad \text{so that the number of bad edges is } 0 \\
&\iff \quad (G,k,0) \in \text{BAD GRAPH COLOURING}.
\end{aligned}
$$

So it's a mapping reductiion from GRAPH COLOURING to BAD GRAPH COLOURING. It's also polynomial-time computable, so in fact it's a polynomial-time mapping reduction. It follows that BAD GRAPH COLOURING is NP-complete, since we already know that GRAPH COLOURING is NP-complete.

(c)

It is routine to show that it is in NP, since a k-colouring can be efficiently verified in the same way as is done for GRAPH COLOURING, and it is easy to do the additional check that k is odd.

To see that it is NP-complete, we can reduce from 3-COLOURABILITY, which is known to be NP-complete. Given a graph G, the reduction just constructs $(G, 3)$. Clearly,

$$\begin{aligned} G \in \text{3-COLOURABILITY} &\iff G \text{ is 3-colourable} \\ &\iff (G,3) \in \text{GRAPH ODD-COLOURING.} \\ &\quad \text{(since 3 is odd).} \end{aligned}$$

So the function $G \mapsto (G, 3)$ is a mapping reduction from 3-COLOURABILITY to GRAPH ODD-COLOURING. The construction is clearly polynomial-time. So it's a polynomial-time mapping reduction. So GRAPH ODD-COLOURING is NP-complete.

It is also possible to reduce from GRAPH COLOURING to GRAPH ODD-COLOURING. If we want to do it that way, then one way to do the reduction is to map $(G, k) \mapsto (H, l)$, where:

- if k is odd:

  $H := G$,

  $l := k$.

- if k is even:

  $H :=$ graph obtained from G by adding a new vertex and joining it to every vertex of G,

  $l := k + 1$.

Observe that, if k is even then G is k-colourable if and only if H is $(k+1)$-colourable.

(d)

It is routine to show that it is in NP, much as in (c), except that the additional check is now that $|V(G)|$ is odd, but this is easy.

To see that it is NP-complete, we can reduce from GRAPH COLOURING, which is known to be NP-complete. Given (G, k), the reduction creates a new graph H from G as follows.

$$H := \begin{cases} G, & \text{if } |V(G)| \text{ is odd;} \\ G \text{ plus an extra isolated vertex,} & \text{if } |V(G)| \text{ is even.} \end{cases}$$

Note that this construction ensures that $|V(H)|$ is odd. Furthermore, the isolated vertex makes no difference to the k-colourability or otherwise of G. So we have

$$\begin{aligned} (G, k) \in \text{GRAPH COLOURING} &\iff G \text{ is } k\text{-colourable} \\ &\iff H \text{ is } l\text{-colourable} \\ &\iff (H, l) \in \text{GRAPH ODD-COLOURING.} \\ &\quad (\text{since } |V(H)| \text{ is odd}) \end{aligned}$$

So the function $(G, k) \mapsto (H, l)$ is a mapping reduction from GRAPH COLOURING to ODD GRAPH COLOURING. The construction is clearly polynomial-time. So it's a polynomial-time mapping reduction. So ODD GRAPH COLOURING is NP-complete.

In fact, we could have done the reduction from 3-COLOURABILITY to ODD GRAPH COLOURING instead. The reduction would then map $G \mapsto (H, l)$, where H is constructed as above and l is 3 if $|V(G)|$ is odd and 4 if $|V(G)|$ is even.

Supplementary exercises

5. (a) Here is a polynomial-time verifier for MOSTLY SAT.

Input: a Boolean expression φ in CNF.
Certificate: a truth assignment t for φ.
Initialisation: `numberOfSatisfiedClauses` $:= 0$.
$m :=$ the total number of clauses of φ.
For each clause C of φ:
{
    If some literal in C is True under t:
        increment `numberOfSatisfiedClauses`
        /* This clause is satisfied. */
}
If `numberOfSatisfiedClauses` $\geq 3m/4$ then ACCEPT.
Otherwise, REJECT.

It clearly always halts, and runs in polynomial time: the algorithm essentially looks at each literal in the expression at most once, checking the truth assignment for the corresponding variable to see if the literal is True, and seeing what effect this has on satisfaction of the clause. The size of the input is at least as large as the total number of literals, and the work done per literal is just consultation of a list of truth values and a small amount of checking.

The input expression φ belongs to MOSTLY SAT if and only if there exists a truth assignment such that the number of satisfied clauses is at least 3/4 of the total number of clauses. This is precisely the condition that there exists a certificate such that the above verifier accepts. Therefore this algorithm is indeed a verifier for MOSTLY SAT.

So it is, in fact, a polynomial-time verifier for MOSTLY SAT.

(b)
Input: a Boolean expression φ in CNF, with m clauses.
Introduce $k := \lfloor m/2 \rfloor$ new variables, $y_1, y_2, \ldots, y_k$, that do not appear in φ.
Create $2k$ (which is m if m is even, and $m-1$ otherwise) new singleton clauses, $(y_1), (\neg y_1), (y_2), (\neg y_2), \ldots, (y_k), (\neg y_k)$
Let φ' be the expression $\varphi \wedge (y_1) \wedge (\neg y_1) \wedge (y_2) \wedge (\neg y_2) \wedge \cdots \wedge (y_k) \wedge (\neg y_k)$.
Output: φ'.

The output expression φ' is clearly in CNF.

The function $\varphi \mapsto \varphi'$ is clearly polynomial-time computable.

If $\varphi \in$ SAT then there is a satisfying truth assignment for φ. Augment this truth assignment by giving each new variable y_i a truth value, for $i = 1, \ldots, k$. (It does not matter which truth values are used.) This new truth assignment satisfies all the m clauses of φ and, for each i, it satisfies exactly one of the two singleton clauses (y_i) and $(\neg y_i)$. So it satisfies $m + k$ clauses altogether. Since φ' has $m + 2k$ clauses altogether, the fraction of clauses that are satisfied is

$$\frac{m+k}{m+2k}.$$

If m is even, this is

$$\frac{m + (m/2)}{m+m} = \frac{3}{4}.$$

If m is odd, the fraction is

$$\frac{m + ((m-1)/2)}{m+m-1} = \frac{3m-1}{4m-2} \geq \frac{3}{4}.$$

So, either way, the fraction is at least 3/4. So, 3/4 of the clauses of φ' are satisfied. So $\varphi' \in$ MOSTLY SAT.

If $\varphi' \in$ MOSTLY SAT, then there exists a truth assignment to φ' that satisfies $\geq 3/4$ of the clauses of φ'. Since φ' has $m+2k$ clauses, this means the truth assignment

satisfies $\geq \frac{3}{4} \cdot (m + 2k)$ clauses. If m is even (so that $2k = m$), this lower bound is $3m/2$, which is an integer. If m is odd (so that $2k = m - 1$), then the lower bound is $3(m-1)/2 + 3/4$. Since, in this odd-m case, $m - 1$ is even, we see that $3(m-1)/2$ is an integer, and since the number of clauses must also be an integer, we can improve the lower bound in the odd case to the next integer after $3(m-1)/2 + 3/4$, namely $3(m-1)/2 + 1$.

Observe that any truth assignment must satisfy exactly k of the $2k$ clauses $(y_1), (\neg y_1), (y_2), (\neg y_2), \ldots, (y_k), (\neg y_k)$. So we can subtract k from the total number of clauses of φ' that are satisfied (see previous paragraph) in order to determine the number of clauses of the original expression φ that are satisfied. For m even, this calculation gives

$$\frac{3m}{2} - k = \frac{3m}{2} - \frac{m}{2} = m.$$

For m odd, the calculation gives

$$\frac{3(m-1)}{2} + 1 - k = \frac{3(m-1)}{2} + 1 - \frac{m-1}{2} = (m-1) + 1 = m.$$

Either way, we find that m of the clauses of φ must be satisfied. In other words, the truth assignment must satisfy *all* clauses of φ. Therefore $\varphi \in$ SAT.

Therefore the function $\varphi \mapsto \varphi'$ is a polynomial-time reduction from SAT to MOSTLY SAT.

(c) It is NP-complete.

6.

(a)

MONOTONE SAT is in P.

In fact, every monotone Boolean expression in CNF is satisfiable: just make every variable True.

(b)

ODD SAT is NP-complete.

It is routine to show that it is in NP, since a truth assignment can be efficiently verified in the same way as is done for SAT.

To see that it is NP-complete, we reduce from 3SAT, which is known to be NP-complete, as follows.

Given a Boolean expression φ in CNF with exactly three literals in each clause, we just output φ unaltered! If $\varphi \in$ 3SAT, then $\varphi \in$ ODD SAT, since the number of literals in each clause of φ, namely 3, is odd. If $\varphi \notin$ 3SAT, then $\varphi \notin$ ODD SAT, since it is not satisfiable. If φ does not have exactly three literals in each clause (in which case it cannot belong to 3SAT, whether or not it is satisfiable), then we output $x \wedge \neg x$, which is unsatisfiable and therefore does not belong to ODD SAT. (Any fixed

output that isn't in ODD SAT would do equally well.) It is clear that this reduction is computable in polynomial time.

(c)

EVEN SAT is NP-complete.

It is routine to show that it is NP, pretty much just as we did for ODD SAT.

To show that it is NP-complete, we use a modified form of the reduction from 2SAT to 3SAT (Lecture 26, slide 8). Let φ be any Boolean expression in CNF. For each clause C of φ, we replace C by two clauses that each have one more literal than C, such that the original clause is satisfiable if and only if the conjunction of the two new clauses is satisfiable. If $C = x_1 \vee x_2 \vee \cdots \vee x_m$, where each x_i is a literal, then the replacement is as follows.

$$x_1 \vee x_2 \vee \cdots \vee x_m \quad \mapsto \quad (x_1 \vee x_2 \vee \cdots \vee x_m \vee z_C) \wedge (x_1 \vee x_2 \vee \cdots \vee x_m \vee \neg z_C),$$

where z_C is a new variable that appears in no other clause. Let φ' be the new Boolean expression produced by this construction. Firstly, φ is satisfiable if and only if φ' is satisfiable. Secondly, φ has an odd number of literals in each of its clauses if and only if φ' has an even number of literals in each of its clauses, since each clause in φ' has exactly one more literal than the clause of φ that it was derived from. The construction can be done in polynomial time. Therefore ODD SAT $\leq_P$ EVEN SAT. So EVEN SAT is NP-complete.

(d)

DNF-SAT is in P.

Consider any Boolean expression φ in DNF. Suppose $\varphi = D_1 \vee D_2 \vee \cdots \vee D_m$, where each D_i is a conjunction of literals.

If any of the D_i has no variable appearing both in unnegated and negated form, then that D_i can be a satisfied by a truth assignment that makes all its literals True. (Any variable that appears unnegated in D_i is True, and every variable that appears negated in D_i is False.) Such a truth assignment also satisfies φ, since φ is satisfied by any truth assignment that makes at least one of the D_i True.

Now suppose D_i contains both x and $\neg x$ for some variable x, then that D_i is not satisfiable, because every truth assignment will make either x or $\neg x$ False, and if *one* of the literals in D_i is False, then D_i itself is False (regardless of the values of its other literals).

It follows that φ is satisfiable if and only if none of the D_i contains a variable in both unnegated and negated form. It is routine to check, for each i, whether any variable appears both unnegated and negated in D_i. So the satisfiability, or not, of D_i can be decided in polynomial time.

7. The verifier works as follows:

1. Input: T
2. Certificate: S
3. Check that there are n triples in S.
4. Check that each triple in S belongs to T.
5. For each pair of triples in S, check that they do not overlap.
6. If all these checks are satisfied, then Accept, otherwise Reject.

Let N be the number of triples in T. (The length of T, as a string, is approximately linear in N.)

Line 3: takes time $O(n)$.
Line 4: For each of the n triples in S, scan along T until it is found: time $O(nN)$.
Line 5: There are $\begin{pmatrix} |S| \\ 2 \end{pmatrix}$ pairs of triples in S. For each, a constant amount of time is needed to check for overlap. So we need time $O(n^2)$.

The total amount of time is $O(N^2)$. Here, we use $n \leq N$, which must be so if there is to be a 3D matching in T. (Strictly speaking, we should probably add an early step in the verifier that checks that $n \leq N$, and if this does not hold, rejects T. This doesn't take much time.)

So the verifier takes polynomial time.

The verifier accepts if and only if S is a 3D matching for T.

So the verifier is indeed a polynomial-time verifier for 3DM.

Hence 3DM $\in$ NP.

8.

Preamble

For each triple $t \in T$, we introduce a new Boolean variable x_t. This is intended to be True if $t \in S$, and False otherwise. The truth assignment is intended to describe a 3DM for T.

For each $a \in A$, and each $i \in \{1, 2, 3\}$, let $T_{a,i}$ be the set of all triples which have a as their i-th member. For example, suppose that $A = \{1, 2\}$ and $T = \{(1, 1, 1), (1, 1, 2), (1, 2, 1), (2, 2, 1)\}$, as earlier. Then $T_{1,2} = \{(1, 1, 1), (1, 1, 2)\}$.

The rules we want to capture, using clauses in CNF, are shown in the following table, together with the clauses that capture them.

Rule	Expression	
For every two triples u and v that overlap, they cannot both be in S.	$\neg x_u \vee \neg x_v$	
Every $a \in A$ must appear as the *first* member of some triple in S.	$x_{t_1} \vee x_{t_2} \vee \cdots \vee x_{t_k}$	where $T_{a,1} = \{t_1, \ldots, t_k\}$.
Every $a \in A$ must appear as the *second* member of some triple in S.	$x_{t_1} \vee x_{t_2} \vee \cdots \vee x_{t_k}$	where $T_{a,2} = \{t_1, \ldots, t_k\}$.
Every $a \in A$ must appear as the *third* member of some triple in S.	$x_{t_1} \vee x_{t_2} \vee \cdots \vee x_{t_k}$	where $T_{a,3} = \{t_1, \ldots, t_k\}$.

Polynomial-time reduction

Input: T
1. For every two triples u and v that overlap, create the clause

$$\neg x_u \vee \neg x_v.$$

2. For each $a \in A$ and each $i \in \{1, 2, 3\}$, determine the set of all triples in $T_{a,i}$. Let them be $\{t_1, \ldots, t_k\}$. Create the new clause

$$x_{t_1} \vee x_{t_2} \vee \cdots \vee x_{t_k}.$$

3. Combine all clauses created so far, using conjunction.
4. Output the Boolean expression we have constructed.

Polynomial time

Step 1:

If T has N triples, then there are at most $\binom{N}{2}$ pairs of triples, and for each, a constant amount of time is required to test if they overlap and create this new clause if needed. So the time required is $O(N^2)$.

Step 2:

There are two nested loops here. The outer loop has n iterations, the inner loop has 3 iterations. So, $3n$ iterations altogether. For each loop iteration, we could simple-mindedly go through each of the N triples in T and check if it has a in the i-th position of the triple, and if so, include the appropriate variable in the clause. This requires looping over T and doing a constant amount of work for each triple in T. So, altogether for this step, the time is $O(nN)$.

Step 3:

The number of clauses we have created so far is $\leq \binom{N}{2} + 3n$. Using $n \leq N$, this is $O(N^2)$. So the amount of work involved in combining them all into a conjunction is $O(N^2)$ too.

In total:

The time taken is $O(N^2)$ which is polynomial time.

9.

Input: T, a set of triples.

For each triple $(w, x, y) \in T$, create n 4-tuples, by creating a 4-tuple (w, x, y, a) for each $a \in A$.

Let T' be the set of all the 4-tuples that we have created.

Output: T'.

The number of 4-tuples created is Nn, where $N = |T|$ and $n = |A|$. Creation of each takes constant time. So the algorithm runs in polynomial time.

It remains to show that $T \in$ 3DM if and only if $T' \in$ 4DM.

We prove $\Rightarrow$, then $\Leftarrow$.

($\Rightarrow$)

Suppose $T \in$ 3DM, and let S be a 3D matching for T. Let the triples in S be $s_1, \ldots, s_n$. As usual, there are n of them. Suppose $A = \{a_1, \ldots, a_n\}$. For each i, append a_i to triple s_i, giving a 4-tuple. So, if s_i is (w_i, x_i, y_i), then the 4-tuple formed from it is (w_i, x_i, y_i, a_i). This gives us a set of n 4-tuples, which we call S'. The fact that all the triples in S are disjoint implies that all the 4-tuples in S' are disjoint. So S' is a 4D matching in T', and $T' \in$ 4DM.

($\Leftarrow$)

Conversely, suppose $T' \in$ 4DM, and let S' be a 4D matching in T'. Because S' is a 4D matching, we know that it has n 4-tuples. Now, construct S from S' by deleting the last member of each 4-tuple, turning the 4-tuple into a triple. Since the 4-tuples in S' are all disjoint, so are the triples so formed. So S is a 3D matching in T, and $T \in$ 3DM.

10.

We prove it by induction on $\ell - k$.

Inductive hypothesis:

There is a polynomial-time reduction from kDM to ℓ DM.

Base case:

If $\ell - k = 0$, then $\ell = k$, and the polynomial-time reduction exists because the identity map (which just maps any set of k-tuples to itself) does the job.

Inductive step:

Suppose the inductive hypothesis holds when $\ell - k = d - 1$, where $d \geq 1$. Now consider what happens for kDM and ℓ DM, where $\ell - k = d$.

We can assume (from the information given in the question) that there is a polynomial-time reduction from kDM to $(k + 1)$DM.

Since $\ell - (k+1) = \ell - k - 1 = d - 1$, the inductive hypothesis tells us that there is a polynomial-time reduction from $(k+1)$DM to ℓ DM.

These two polynomial-time reductions combine (i.e., compose) to give a polynomial-time reduction from kDM to ℓ DM.

Conclusion:

The result therefore follows, by the Principle of Mathematical Induction.