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

FIT2014
Tutorial 7
Complexity: NP and Polynomial Time Reductions

SOLUTIONS

1.
(a)
Here is a polynomial-time verifier for EDGE-COLOURING.

Input: a graph G, a positive integer k.
Certificate: a function $f : E(G) \to \{1, 2, \ldots, k\}$ that assigns a colour to each edge.
For each vertex v of G:
{
  Make a list of all the colours that appear on the edges incident with v.
  If any colour appears more than once in this list, then REJECT.
  Otherwise, continue.
}
ACCEPT, since we only reach this point if every vertex has no colour repeated among the colours of its edges.

It is clear that $(G, k) \in$ EDGE-COLOURING if and only if there exists a certificate such that this algorithm accepts (since the certificate is just the edge-colouring). So this is indeed a verifier for EDGE-COLOURING.

The verifier runs in polynomial time: the outer loop is executed n times (where n is the number of vertices of G), and for each iteration, the inner loop involves a test that all list elements belong to the specified colour set and are distinct, which takes polynomial time using standard list methods (and using the fact that the number of edges incident with a vertex is $\leq n$).

So we have a polynomial-time verifier for EDGE-COLOURING.

(b) The problem can be solved in polynomial time when

- $k = 1$: in this case, G is not 1-edge-colourable unless it is a union of disjoint edges (i.e., edges that have no vertices in common);
- $k = 2$: in this case, G is not 2-edge-colourable unless it is a union of disjoint paths and even-length circuits. In the latter case, assigning one colour to any

edge of one of the paths or circuits forces the colours of all other edges in that component.

2.

(a) We prove that P = NP[SHORT] by proving set containment in each direction.

($\subseteq$)

Every language in P has a polynomial-time decider. But this is also a polynomial-time verifier with empty certificate. So it has a polynomial-time verifier with certificate length ≤ 100. So it belongs to NP[SHORT].

Therefore P $\subseteq$ NP[SHORT].

($\supseteq$)

Let $L \in$ NP[SHORT]. By definition of NP[SHORT], it has a polynomial-time verifier V with certificate length ≤ 100. We can use this in a polynomial-time decider for L, as follows.

Input: string x, being a possible input for L.
For every string y of ≤ 100 characters
{
    Run the L-verifier, V, on input x with certificate y.
    If V accepts (x, y), then ACCEPT.
    Otherwise, continue.
}
REJECT.

This decider runs in polynomial time, since it the number of outer loop iterations is the number of strings of ≤ 100 characters, which is $1+2+2^2+2^3+\cdots+2^{99}+2^{100}$, which is $< 2^{101}$, which is constant (although large)[1], and the time spent inside the loop is dominated by the time spent running V, which is bounded by a polynomial in the input size since V is a polynomial-time verifier.

We now show that the decider is indeed a decider for L, by showing that $x \in L$ if and only if the decider accepts x.

($\Longrightarrow$)

If $x \in L$, then there is some certificate y with $|y| \leq 100$ such that V accepts (x, y). So the decider will encounter y during its iteration over all possible certificates of length ≤ 100, and will accept in that iteration because V accepts then.

($\Longleftarrow$)

[1]It is routine to extend this argument to larger alphabets.

On the other hand, if the decider accepts x, then the only way that can happen is if some y causes V to accept (x, y) during a main loop iteration. That is only possible if $x \in L$, since V is a verifier for L.

So our decider is indeed a polynomial-time decider for L. So $L \in \mathrm{P}$.

Therefore $\mathrm{P} \supseteq \mathrm{NP[SHORT]}$.

We have now shown both the required containments. So $\mathrm{P} = \mathrm{NP[SHORT]}$.

(b) $f(n) = c \log n$ will do the job. We can replace 100 by $c \log_2 n$ throughout the above argument, and it will still work. The crucial observation that the number of certificates is bounded above by about $2^{c \log_2 n} = n^c$, so the number of iterations of the main loop in the decider is polynomially bounded.

Question: what happens if we use $f(n) = c(\log n)^2$? What about $f(n) = c \log n \log \log n$? What about other functions that grow more quickly than $c \log n$ (for any fixed c)?

(c) It's tempting to take some polynomial, such as $f(n) = n^2$. The class $\mathrm{NP}[n^2]$ contains a "large chunk" of NP; it includes, for example, SATISFIABILITY (for which the certificate is a list of binary values True/False for all the variables, so is in fact shorter than the input length), and 3-COLOURABILITY, and EDGE-COLOURABILITY, and HAMILTONIAN CIRCUIT, and much else. But, as far as we know, some problems in NP have verifiers whose certificates seem to need more than n^2 characters. The same issue arises if any polynomial function is used instead of n^2. It is not currently known whether there exists any constant c such that $\mathrm{NP} = \mathrm{NP}[n^c]$.

Instead, try $f(n) = 2^n$. This function grows so quickly that it eventually overtakes any polynomial, a fact we use below. We prove that $\mathrm{NP} = \mathrm{NP}[2^n]$ by proving the two required containments.

($\supseteq$)

Let $L \in \mathrm{NP}[2^n]$. A polynomial-time verifier with a certificate length bound is still a polynomial-time verifier, so L has a polynomial-time verifier, so $L \in \mathrm{NP}$.

($\subseteq$)

Let $L \in \mathrm{NP}$. Let V be a polynomial-time verifier for L. There exist constants a, k such that V only ever uses at most an^k characters in the certificate.[2] For sufficiently large n, we have $an^k < 2^n$. Let N be a constant such that $an^k < 2^n$ for all $n > N$.

We construct a new verifier for L as follows. All inputs of length $\leq N$ are treated as special cases and solved directly, by looking up a table if necessary. (This may be

[2]The polynomial time bound can be used as the bound on certificate length here, since at most one certificate character can be read in each time-step.

a large table, but it is nonetheless finite, and its size does not depend on n.) So, for these cases, no certificate is used. For inputs of length $> N$, the verifier V is used, and the length of certificate for these inputs is $< 2^n$. So, for *all* inputs, the certificate length for this new verifier is $< 2^n$. So we have shown that $L \in \mathrm{NP}[2^n]$.

3.

(a)

Here is a polynomial-time verifier for NEARLY 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` $\geq m - 1$ 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 NEARLY SAT if and only if there exists a truth assignment such that the number of unsatisfied clauses is ≤ 1. This is precisely the condition that there exists a certificate such that the above verifier accepts. Therefore this algorithm is indeed a verifier for NEARLY SAT.

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

(b)

Input: a Boolean expression φ in CNF.
Introduce a new variable, y, that does not appear in φ.
Create two new singleton clauses, one containing just y and the other containing just $\neg y$.
Let φ' be the expression $\varphi \wedge (y) \wedge (\neg y)$.
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 assigning a truth value to y. (It does not matter whether y is True or False.) This new truth assignment satisfies all the clauses of φ and exactly one of the two singleton clauses we added. So it satisfies all but one of the clauses of φ'. So $\varphi' \in$ NEARLY SAT.

If $\varphi' \in$ NEARLY SAT, then there exists a truth assignment to φ' that satisfies all, or all but one, of the clauses of φ'. So there is at most one clause that is not satisfied. Observe that it is impossible for any truth assignment to satisfy both y and $\neg y$. So one of these clauses may be unsatisfied. It follows that all of φ must be satisfied (since at most one clause of φ' can be unsatisfied). Therefore $\varphi \in$ SAT.

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

(c) It is NP-complete.

(d) 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 $\neg y_i$ ($i = 1, 2, \ldots, k$).

4.

(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.

5.

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.

6.

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.

7.

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.

8.

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 $\binom{|S|}{2}$ 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.

9.

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.

10.

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.

11.

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

Inductive hypothesis:

There is a polynomial-time reduction from kDM to $\ell\,$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 $\ell\,$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 $\ell\,$DM.

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

Conclusion:

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