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

FIT2014
Tutorial 7
Complexity: NP and Polynomial Time Reductions

SOLUTIONS

1.
(a)
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 $\overline{y}$.
Let φ' be the expression $\varphi \wedge (y) \wedge (\overline{y})$.
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 (s.t.a.) 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 $\overline{y}$. 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.

(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, $(y_1), (\overline{y_1}), (y_2), (\overline{y_2}), \ldots, (y_k), (\overline{y_k})$
Let φ' be the expression $\varphi \wedge (y_1) \wedge (\overline{y_1}) \wedge (y_2) \wedge (\overline{y_2}) \wedge \cdots \wedge (y_k) \wedge (\overline{y_k})$.
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 s.t.a. 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 $(\overline{y_i})$. 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 $(y_1), (\overline{y_1})$, $(y_2), (\overline{y_2}), \ldots, (y_k), (\overline{y_k})$. 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)

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.

2.

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.

3.

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.

4.

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.

5.

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.

6.

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.	$\overline{x_u} \vee \overline{x_v}$	
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

$$\overline{x_u} \vee \overline{x_v}.$$

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 $\begin{pmatrix} N \\ 2 \end{pmatrix}$ 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 $\leq \begin{pmatrix} N \\ 2 \end{pmatrix} + 3n$. 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.

7.

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.

8.

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.