Computational Complexity — tutorial 4'

NP-completeness, PSpace-completeness

Reductions

A (Karp) reduction from a decision problem X to a decision problem Y is a function f mapping inputs of X to the inputs of Y which preserves the answer (YES/NO) to the problem.

Simple examples:

- if X is a decision problem „is the number n given to you divisible by 10?", and Y is „does the remainder of the division of n by 10 equal 3?", then $f(n) = n + 3$ is a reduction from X to Y—n is divisible by 10 if and only if the remainder of the division of $f(n)$ by 10 equals 3.
- if X is a decision problem „is the number n prime?", and Y is a decision problem „given (n, k), is the number n pairwise coprime with all the numbers in range $[1, k]$?", then there exists a reduction from X to Y: $f(n) = (n, \lceil\sqrt{n}\rceil)$.

Time/space bounds. A reduction f is said to be polynomial time (or logarithmic space) if f can be computed in polynomial time (or logarithmic space).

Intuition. If there is a reduction f from X to Y, then—in some sense—X is easier/not harder to solve than Y. Why? If we had a function `SolveY` which solves Y, then using it we can also solve X: `SolveX`(input) := `SolveY`(f(input)).

So, if a reduction f is reasonably fast (polynomial time/logarithmic space), then e.g. a polynomial time solution to Y immediately implies a polynomial time solution to X.

NP-completeness

A problem X is said to be NP-complete if X is in NP and X is NP-hard (i.e., **every** problem in NP can be reduced in polynomial time (logarithmic space) to X). In some sense, NP-complete problems are *the hardest* problems in NP. Note that if we solve **any** NP-complete problem in polynomial time, then we immediately get a polynomial-time solution to **every** problem in NP (see above).

The following two problems can be directly shown to be NP-complete:

NP Halting Problem
Input: non-deterministing Turing Machine $\mathcal{M}$, its input w, integer k in unary
Output: is there an accepting run of $\mathcal{M}(w)$ terminating in at most k steps?

SAT
Input: a binary formula φ with variables $x_1, x_2, \ldots, x_n$ for some n
Output: is there an assignment $(x_1, \ldots, x_n) \in \{\text{false}, \text{true}\}^n$ which satisfies the formula?

The latter fact is known as Cook's theorem, and its NP-completeness was shown on the 3rd lecture.

Proving NP-completeness directly is painful, but there's an easier way.

Observation. Let X, Y be two decision problems in NP. X is NP-complete. If there is a polynomial time reduction from X to Y, then Y is also NP-complete.

Proof. We know that X is NP-hard, i.e. each decision problem Z which is in NP can be reduced to X in polynomial time. We also assumed that Y is in NP, so we only need to show that Y is NP-hard. In order to do that, let's pick an arbitrary decision problem $Z \in \mathsf{NP}$, and let's show that Z can be reduced to Y in polynomial time.

Let f_{XY} be the polynomial time reduction from X to Y (which exists per our assumptions). Let also f_{ZX} be the polynomial time reduction from Z to X (which exists since X is NP-hard). We define f_{ZY}, a polynomial time reduction from Z to Y, in a natural way: $f_{ZY}(\texttt{input}) := f_{XY}(f_{ZX}(\texttt{input}))$. We note that:

- f_{ZY} can be computed in polynomial time (since obviously the composition of any two polynomial time functions is itself polynomial time).
- For every instance `input` of Z, the answer to `input` must be the same as the answer to $f_{ZX}(\texttt{input})$ in X (since f_{ZX} is a reduction), which in turn must be the same as the answer to $f_{XY}(f_{ZX}(\texttt{input})$ in Y (since f_{XY} is a reduction).

Hence, f_{ZY} is a polynomial time reduction from Z to Y as well. As Z was an arbitrary problem in NP, this concludes the proof.

Note that we can do a similar proof for logarithmic space reductions, but there's one part of the proof that doesn't translate easily to this case—namely:

Lemma. if f, g are any two functions computable in logarithmic space, then the composition of f and g (i.e., $x \to g(f(x))$) is computable in logarithmic space as well.
In other words, the class LogSpace is closed under compositions.

We won't prove this lemma yet (we will later, though), but this is a really nice exercise for you!

So, how to prove NP-completeness?

The observation above provides us with a powerful method of proving that some decision problem is NP-complete:

1. We want to prove that some decision problem X is NP-complete.
2. We first show that X is in NP. It's usually the most straightforward step—in most problems in NP, you can easily understand what would be a solution to your problem (e.g. 3-coloring of a given graph, a satisfying assignment of binary variables). You can „guess" this solution non-deterministically and then verify its correctness.
3. In order to show that X is NP-hard, pick **any** problem that you already know that it's NP-complete (e.g. NP Halting Problem, SAT, or maybe any of the problems we'll see in the following section). You just need one such problem! Let's call this problem A.
4. Find a polynomial time reduction from A to X (an efficient way of transforming any correct input to A to an equivalent input of X). The **Observation** above implies that X is NP-hard.
5. Since X is in NP and it is NP-hard, we find that X is NP-complete.

More NP-complete problems

Using the method above, we can prove that the following, more manageable, variants of SAT are also NP-complete.

CIRCUIT-SAT
INPUT: a binary circuit (with binary OR/AND gates, unary NOT gates), with n inputs, named $x_1, x_2, \ldots, x_n$, and one output, named y. The circuit cannot have any cycles.
OUTPUT: is there an assignment $(x_1, \ldots, x_n) \in \{\text{false}, \text{true}\}^n$, for which $y = \texttt{true}$?

Proof. CIRCUIT-SAT is obviously in NP: guess an assignment of logic variables on the inputs, and verify that the output is `true`.

Now, we want to pick any logical formula φ and convert it to an equivalent binary circuit. Assume that φ has n variables: $x_1, x_2, \ldots, x_n$. We turn each of these variables into inputs $x_1, x_2, \ldots, x_n$ of our circuit. Now, we'll build our circuit recursively:

- If $\varphi = x_i$ for some $i \in [1, n]$, the output of the circuit is simply the input x_i. We can easily see that the output of the circuit is `true` if and only if $x_i = \texttt{true}$.
- If $\varphi = \neg\psi$ for some formula ψ, we create the circuit for ψ recursively, and append a NOT gate to the output of that circuit. We can see that the output of the circuit is `true` iff ψ evaluates to `false`, or equivalently, φ evaluates to `true`.
- If $\varphi = \psi_1 \vee \psi_2$ for some formulas ψ_1 and ψ_2, we create two circuits for ψ_1 and ψ_2 recursively; for simiplicity, assume that these circuits share the inputs $x_1, \ldots, x_n$. Let the output of the first circuit be y_1, and the output of the second circuit—y_2. We now create an OR gate whose inputs are y_1 and y_2, and whose output is y. We let y be the output of the circuit for φ. It's obvious that $y = \texttt{true}$ if and only if φ evaluates to `true`.
- If $\varphi = \psi_1 \wedge \psi_2$, we do similarly to above, only that we use an AND gate instead.

Obviously, the circuit can be created in linear (polynomial) time. Moreover, it's easy to see that the circuit evaluates to `true` for some inputs $x_1, x_2, \ldots, x_n$ if and only if the same assignment of variables satisfies the initial formula φ. Hence, the circuit is satisfiable (=the answer to an instance of CIRCUIT-SAT is positive) if and only if the original formula was satisfiable (=the answer to an instance of SAT is positive). Therefore, we provided a polynomial-time reduction from SAT to CIRCUIT-SAT. We get that CIRCUIT-SAT is NP-hard. Since it was also in NP, we conclude that CIRCUIT-SAT is NP-complete.

In order to introduce another variant of SAT, we need a definition. We say that a formula φ is in k-CNF form if it is a conjunction (AND) of k-clauses, each k-clause is a disjunction (OR) of exactly k literals, and each literal is of the form x_i or $\neg x_i$. E.g. $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3 \vee \neg x_5)$ is in 3-CNF.

3-CNF-SAT
INPUT: a binary formula φ with variables $x_1, x_2, \ldots, x_n$ for some n, in 3-CNF form
OUTPUT: is there an assignment $(x_1, \ldots, x_n) \in \{\text{false}, \text{true}\}^n$ which satisfies the formula?

Proof. Obviously, 3-CNF-SAT is in NP (we guess the assignment of variables and check in polynomial time if the formula evaluates to `true`).

We now prove that 3-CNF-SAT is NP-hard through a reduction from Circuit-Sat (we can do that now since we now know that Circuit-Sat is NP-complete!).

Assume that we're given a binary circuit C with inputs $x_1, x_2, \ldots, x_n$ and output y, and we want to convert it to an equivalent formula in 3-CNF form, which is satisfiable if and only if there exists an assignment of variables $x_1, \ldots, x_n$ which makes y true. To that goal, we'll create a 3-CNF formula with the following variables:

- $x_1, x_2, \ldots, x_n$, denoting the inputs of the circuit C,
- $y_1, y_2, \ldots, y_m$ with m equal to the number of logic gates in C, corresponding to the outputs of these logic gates. We can assume for simplicity that $y = y_m$ (i.e., y_m is the output of C).

Each y_i is the result of a logic operation (OR/AND/NOT) on one or two inputs (variables). For instance, from some circuit C we can have that $y_1 = x_1 \wedge x_3$, $y_2 = y_1 \vee x_2$, $y_3 = \neg y_1$, $y_4 = x_2 \wedge y_3$. Hence, the following formula is satisfiable if and only if the circuit C is satisfiable:

$$(y_1 = x_1 \wedge x_3) \ \wedge \ (y_2 = y_1 \vee x_2) \ \wedge \ (y_3 = \neg y_1) \ \wedge \ (y_4 = x_2 \wedge y_3) \ \wedge \ y_4.$$

This is not in 3-CNF form yet, but it somehow resembles 3-CNF: we have a conjunction (AND) of a bunch of formulas, each regarding at most three different variables. Thanks to this, we can transform it into 3-CNF in a straightforward way:

- The term y_m is not a 3-clause. But we can write $(y_m \vee y_m \vee y_m)$ instead.
- Let's transform a formula of the form $(p = \neg q)$ into 3-clauses. We have $q \to \neg p$ (or equivalently, $\neg p \vee \neg q$), and similarly $\neg q \to p$ (or equivalently, $p \vee q$). Hence,

$$(p = \neg q) \quad \Leftrightarrow \quad (p \vee q) \wedge (\neg p \vee \neg q) \quad \Leftrightarrow \quad (p \vee p \vee q) \wedge (\neg p \vee \neg p \vee \neg q).$$

- Let's transform a formula of the form $(p = q \vee r)$ into 3-clauses. Note that we cannot have simultaneously $q = \texttt{false}$, $r = \texttt{false}$, and $p = \texttt{true}$. Hence, $\neg(\neg q \wedge \neg r \wedge p)$, or equivalently, $(q \vee r \vee \neg p)$. We can write analogous conditions for any combinations of q and r. Hence,

$$(p = q \vee r) \quad \Leftrightarrow \quad (q \vee r \vee \neg p) \ \wedge \ (q \vee \neg r \vee p) \ \wedge \ (\neg q \vee r \vee p) \ \wedge \ (\neg q \vee \neg r \vee p).$$

- We can do similarly for $(p = q \wedge r)$.

This way, we produced a 3-CNF formula which is satisfiable if and only if the circuit C was satisfiable. Since all the steps are easy to do in polynomial time complexity, we conclude that the problem 3-CNF-SAT is NP-hard. Since it's also in NP, we get that 3-CNF-SAT is NP-complete.

Now, try to solve the following problem:

> Input: three boolean formulas $\varphi_1, \varphi_2, \varphi_3$.
> Output: is there an assignment $(x_1, \ldots, x_n) \in \{\text{false}, \text{true}\}^n$ which satisfies at least two formulas at the same time?

Hint 1: reduce from SAT.

Hint 2: $f(\varphi) = (\varphi, \varphi, \varphi)$.

Hint 3: prove that φ is satisfiable if and only if in the triple of formulas $(\varphi, \varphi, \varphi)$, at least two formulas are satisfiable at the same time.

Another problem with a (mostly) full solution:

CLIQUE
INPUT: an undirected graph G, an integer k
OUTPUT: is there a subset $A \subseteq V(G)$, $|A| = k$, which is a clique in G?

Hint 1: reduce from 3-CNF-SAT. Hence, we are writing a function transforming inputs to 3-CNF-SAT into inputs to CLIQUE. The function should preserve the answer to the problem (YES/NO)—if the formula φ is satisfiable, we must produce a graph with a sufficiently large clique, otherwise a graph with no large cliques.

Hint 2: assume there are m clauses in the formula φ. Create $3m$ variables: one for each literal.

Hint 3: $k = m$. We want to ensure that the clique picks exactly one literal from each clause and that no contradiction arises.

Solution (assuming you read all the hints): it's trivial to see that CLIQUE is in NP. We remind that we write a reduction from 3-CNF-SAT to CLIQUE. Hence, we need to write a function taking an arbitrary formula φ and transforming it into an instance of CLIQUE.

We assume that φ has n variables and m clauses (things of the form $(x_i \vee \neg x_j \vee \neg x_k)$). In the i-th clause ($1 \leqslant i \leqslant m$), for each of three literals (thing of the form x_i or $\neg x_i$), we create a vertex v_{ij} ($j \in \{1, 2, 3\}$); v_{ij} corresponds to the j-th literal in the i-th clause.

We connect two vertices $v_{i_1 j_1}, v_{i_2 j_2}$ with an edge if:

- $i_1 \neq i_2$, and
- $v_{i_1 j_1}, v_{i_2 j_2}$ do not represent contradictory clauses; e.g. they can represent x_3 and $\neg x_5$, as well as x_2 and x_2; but they cannot represent x_6 and $\neg x_6$.

We also set $k := m$. We have finished the construction of the instance of CLIQUE: we just provided a graph and the value of k. It remains to show that the answer to the input to 3-CNF-SAT is the same as the answer to the constructed instance of CLIQUE.

If the formula φ has a satisfying assignment $(x_1, x_2, \ldots, x_n) \in \{\text{false}, \text{true}\}^n$, then we can pick one satisfied literal from each clause: j_1-st literal from the first clause, j_2-nd literal from the second clause, ..., j_m-th literal from the m-th clause. We can now easily show that $\{v_{1j_1}, v_{2j_2}, \ldots, v_{mj_m}\}$ is a clique of size m (an easy exercise).

Conversely, assume that the graph contains a clique of size m. Since no pair of vertices $v_{i_1 j_1}, v_{i_2 j_2}$ is connected by an edge for $i_1 = i_2$, we infer that each vertex v_{ij} in the clique must have a different i. Hence, the clique consists of vertices $v_{1j_1}, v_{2j_2}, \ldots, v_{mj_m}$ for some $j_1, j_2, \ldots, j_m \in \{1, 2, 3\}$. These vertices correspond to m mutually non-contradictory literals (that is, no pair of vertices can correspond to literals x_i and $\neg x_i$, correspondingly). Hence, we can recover a satisfying assignment from these literals—e.g. if some variable x_i occurs only positively (without negation) in the selected literals, we set $x_i = \text{true}$. Otherwise, we set $x_i = \text{false}$.

Therefore, the reduction preserves the answer—the positive instances of 3-CNF-SAT are transformed into positives of CLIQUE, and negative instances—into negative instances.

PSPACE-completeness

We define PSpace-completeness analogously to NP-completeness. A problem X is PSpace-complete if:

- X is in PSpace,
- X is PSpace-hard (i.e., for all problems Y in PSpace, there exists a polynomial time reduction from Y to X).

Or equivalently (since the **Observation** from NP-completeness works here as well):

- X is in PSpace,
- X is PSpace-hard (i.e., **for some** PSpace-complete problem Y, there is a polynomial time reduction from Y to X).

Some well known PSpace-complete problems:

PSpace Halting Problem
Input: deterministing Turing Machine $\mathcal{M}$, its input w, integer k in unary
Output: does $\mathcal{M}(w)$ accept so that the memory usage of $\mathcal{M}$ does not exceed k?

QBF (Quantified Boolean Formula)
Input: a binary formula φ with quantifiers and no free variables, e.g. $\exists x_1 \forall x_2 \exists x_3 (x_1 \wedge (\neg x_2 \vee x_3))$
Output: is φ true?

Note that in QBF, we can assume that all quantifiers occur at the beginning of the formula: e.g. a formula $\exists x_1 \forall x_2 [(x_1 \vee x_2) \wedge \exists x_3 (x_3 \wedge \neg x_1)]$ can be transformed into $\exists x_1 \forall x_2 \exists x_3 (x_1 \vee x_2) \wedge (x_3 \wedge \neg x_1)$.

We'll see that the following problem is PSpace-hard:

Formula Game
Input: a binary formula φ with variables $x_1, x_2, \ldots, x_n$.
Output: Consider a two-player game where players alternately choose the values of variables: the first player chooses the value of $x_1 \in \{\texttt{false}, \texttt{true}\}$, then the second player chooses the value of x_2, then the first player chooses x_3 etc. The first player wins if φ evaluates to true when the values of all variables have been chosen. Does the first player win?

Proof: we can easily prove that Formula Game is in PSpace, by an exhaustive search of the game tree; we just need to take care not to memoize any of the states we have previously visited:

```
# Returns whether the first player can win on formula [phi] with remaining [variables],
#  with [player_to_move] indicating the player which chooses next variable.
def FormulaGame(phi, player_to_move, variables):
    if no variables remain:
        return phi.evaluate() == True
    elif player_to_move == First:
        first_var = variables[0]
        # We win if we can pick any move forcing the second player to lose.
        return (FormulaGame(phi.set(first_var, False), Second, variables[1:]) or
                FormulaGame(phi.set(first_var, True), Second, variables[1:]))
```

```
    else:
        first_var = variables[0]
        # The first player wins if the second player cannot force the first player to lose.
        return (FormulaGame(phi.set(first_var, False), First, variables[1:]) and
                FormulaGame(phi.set(first_var, True), First, variables[1:]))
```

(Note: we don't have to be such meticulous about proving that a problem is in PSpace; the first paragraph is OK, I just wanted to show the full algorithm for more clarity.)

In order to prove that Formula Game is PSpace-hard, we'll reduce from Quantified Boolean Formula. Given a quantified binary formula φ, we want to create (in polynomial time) a binary formula ψ such that the first player wins Formula Game on ψ if and only if φ is true.

As above, we can assume that in φ (the instance of QBF), all quantifiers precede the remaining part of the formula. Also, we can transform φ to a form in which the first quantifier is $\exists$ (existential), and the quantifier kinds alternate; that is, the sequence of quantifiers looks like this: $\exists\forall\exists\forall\exists\forall\ldots\exists\forall$. If the original formula is not of this form, we can introduce dummy variables and quantify over them in an arbitrary part of the formula, e.g.

$$\exists_{t_1}\forall_{t_2}\forall_{t_3}\forall_{t_4}\ \alpha(t_1,t_2,t_3,t_4) \qquad \Leftrightarrow \qquad \exists_{t_1}\forall_{t_2}\exists_{z_1}\forall_{t_3}\exists_{z_2}\forall_{t_4}\ \alpha(t_1,t_2,t_3,t_4) \qquad \text{for every formula } \alpha.$$

Now, assume that the formula φ looks like this: $\exists_{x_1}\forall_{x_2}\exists_{x_3}\forall_{x_4}\ldots\exists_{x_{n-1}}\forall_{x_n}\psi(x_1,x_2,\ldots,x_n)$ (the last quantifier will be $\exists$ instead if n is odd).

Lemma. φ is true if and only if the first player wins the Formula Game on ψ.

The proof of the lemma is inductive. For $n=0$, the proof is obvious. Now, assume that the lemma holds for all formulas with n quantifiers, and let's prove that it works for all formulas with $n+1$ quantifiers. Let's pick a formula with $n+1$ quantifiers:

$$\exists_{x_1}\forall_{x_2}\exists_{x_3}\forall_{x_4}\ldots\forall_{x_n}\exists_{x_{n+1}}\psi(x_1,x_2,\ldots,x_n,x_{n+1}).$$

(Note: this sequence of quantifiers assumes that n is even; for n odd the final quantifier would be $\forall$, but the overall proof wouldn't change.)

Let's rewrite the formula above equivalently:

$$\exists_{x_1}\ \neg[\exists_{x_2}\forall_{x_3}\exists_{x_4}\ldots\exists_{x_n}\forall_{x_{n+1}}\ \neg\psi(x_1,x_2,\ldots,x_n,x_{n+1})]. \tag{0.1}$$

(We added two negations, and all quantifiers after the first one changed to the opposite.)

We now notice two things:

- After the first player picks the value of x_1, say, `true`, the second player, in fact, starts an n-round Formula Game on $\neg(\psi[x_1 := \texttt{true}])$ (so the game is played on $\neg\psi$ instead of ψ, and the first and second players swap). From the inductive assumption, the second player (starting this sub-game) wins this game if and only if

$$\exists_{x_2}\forall_{x_3}\exists_{x_4}\ldots\exists_{x_n}\forall_{x_{n+1}}\ \neg\psi(\texttt{true},x_2,\ldots,x_n,x_{n+1}).$$

- The first player has a winning strategy if and only if they have a move which requires the second player to lose (if both players then play optimally). We can now easily see that this is exactly what Equation (0.1) means. Hence, the first player wins if and only if Equation (0.1) holds.

This finishes the proof of the lemma.

The lemma, in fact, proves that the reduction is correct—we generated a Formula Game instance ψ in which the first player wins if and only if the original QBF formula φ was true. We can easily see that the reduction can be done in polynomial time. Hence, Formula Game is PSpace-hard, and since it's in PSpace, it's PSpace-complete as well.

TODO: Generalized Geography.