CS3230 Chapter 9 - NP-Completeness

Based on lectures by Chang Ee-Chien
Notes taken by Andrew Tan
AY18/19 Semester 1

1 Decision problems

There are several types of algorithmic problems:

1. **Decision problems** - The output is a binary bit $\{0, 1\}$ representing the decision "YES" or "NO".
2. **Optimization problems** - There are many feasible solutions, and each solution is associated with some values. We wish to find the solution with the optimal value, and can either output
   (a) the optimal solution, or
   (b) the value of the optional solution.
3. **Counting problems** - The output is a non-negative integer which counts the number of feasible solutions.

Take the minimum spanning tree problem as an example. We can formulate the problem into any of the categories above:

1. Our input is a graph and a value b. Does the graph contain a spanning tree with weight at most b?
2. (a) What is the minimum spanning tree?
   (b) What is the weight of the minimum spanning tree?
3. Our input is a graph and a value b. What is the number of spanning trees with weight less than b?

We can also consider our decision problem algorithm as a test of membership. Consider a decision problem, and let L be the set of input instances such that the output is "YES". Thus, an algorithm that solves the decision problem will correctly determine whether the input x is in the set L.

We often represent a decision problem as a set. If given a decision algorithm, that on input x, outputs "YES" if, and only if, $x \in L$, we say that the algorithm decides L. For example, for a decision algorithm that determines if an integer is prime, we can represent the decision problem as the set of all primes.

2 $\mathcal{P}$, Polynomial time algorithms

An algorithm is polynomial time if its worst-case running time is in $O(n^k)$ where n is the size of the input, and k is a constant independent of n.

We consider polynomial time algorithms as "efficient".

2.1 Polynomial time problems

$\mathcal{P}$ is a collection of decision problems. A decision problem L is in $\mathcal{P}$ if, and only if, there is a *deterministic* polynomial time algorithm that decides L.

We can view $\mathcal{P}$ as the collection of problems that can be efficiently solved. For example, let K be the set of weighted graphs whose minimum spanning tree have a weight of less than some constant a. As finding the minimum spanning tree of a graph takes polynomial time, we say that $K \in \mathcal{P}$.

While an algorithm with $\Theta(n^{100})$ may seem slow, it turns out that there are many important problems where the best known algorithms have non-polynomial time. Furthermore, if we find a polynomial time algorithm for a problem, it is usually possible to reduce the size of the polynomial through the discovery of more efficient algorithms. Finally, polynomials are closed under addition, multiplication, and composition, so if we compose multiple polynomial algorithms together, we will still have a polynomial time algorithm.

3 $\mathcal{NP}$, Non-deterministic Polynomial time algorithms

Oftentimes, it is tough to find the solution or decide whether an instancec has a solutions. However, if the solution is found, one can easily verify that the solution is correct.

For example, it is difficult to factorize an integer, but if given the solution, it is easy to verify it. We can view problems whose solutions are easy to verify as non-deterministic polynomial.

3.1 Non-deterministic polynomial time problems

$\mathcal{NP}$ is a collection of decision problems. A decision problem K is in $\mathcal{NP}$ if, and only if, there exists a $Q \in \mathcal{P}$ and polynomial $p()$ such that:
$x \in K$ if, and only if, there exists a y where $|y|$ where $|y| < p(|x|)$ and $\langle x, y\rangle \in Q$.

In the above definition, $|y|$ refers to the number of bits required to represent x. Furthermore, the y in the definition is known as the "**proof**" or the "**witness**". It is the witness that the element x is in K.

We can also view y as the solution of a problem, while the decision problem is on whether the instance has a solution. The solution y is a proof that the instance x indeed has a solution. Furthermore, $|y| < p(|x|)$ refers to the proof being short, and $\langle x, y\rangle \in Q$ refers to the proof being able to be efficiently verified by computing Q.

In essence, $x \in K$ if, and only if, there exists a short proof that can be verified in polynomial time.

3.1.1 An example

Let K be the set of non-primes. We claim that $K \in \mathcal{NP}$.

Proof:

1. let Q be the set of $\langle x, y\rangle$, where x is divisible by y. $Q = \{\langle 4, 2\rangle, \langle 6, 2\rangle, \langle 9, 3\rangle, \dots\}$.
2. By definition of non-primes, a number $x \in K$ if, and only if, there exists a $y > 1$ such that x is divisible by y.
3. Let's take the above y as the witness. Since $y < x$, then $|y| < |x|$. Also, $x \in K$ if, and only if, there exists a y such that $\langle x, y\rangle \in Q$.
4. Since $Q \in P$, all conditions in the definition are made. Thus $K \in \mathcal{NP}$.

For example, 135 is not a prime because there exists a witness $y = 5$, and $\langle 135, 5\rangle \in Q$. Furthermore, the witness is short and easily verified.

4 $\mathcal{P} = \mathcal{NP}$

We have the following theorem: $\mathcal{P} \subseteq \mathcal{NP}$.

This theorem states that for any decision problem that can be decided in polynomial time, its witness can also be verified in polynomial time. Essentially, if the solution can be found in polynomial time, then the solution can be verified in polynomial time.

Now, one of the most important questions in computer science is if $\mathcal{NP} \subseteq \mathcal{P}$? If this is true (which implies $\mathcal{NP} = \mathcal{P}$), then *any* problem in $\mathcal{NP}$ can be solved efficiently.

5 Decision reducible

Suppose we have an algorithm that solves a $\mathcal{NP}$ decision problem, that is, given an input, it outputs "YES" or "NO" correctly. Using this algorithm, can we find the witness, if it exists?

Let's call the algorithm an oracle. We say that a decision problem is **decision-reducible** if, given an "oracle" that solves the decision problem in $O(1)$ time, we can find the witness if polynomial time.

6 Polynomial time reduction

Suppose we have an algorithm $\mathcal{B}$ that "efficiently" solves a particular problem B. Can we use it to efficiently solve another problem A?

We can view $\mathcal{B}$ as an oracle, and our job is now to use the oracle to solve A. If there is such an efficient algorithm $\mathcal{A}$ that uses $\mathcal{B}$ to solve A, we may make the following statements:

- Problem A is no harder than B,
- A is easier or equivalent to B,
- $A \leq B$,
- Algorithm $\mathcal{A}$ reduces problem A to B.

The above is a rough description. There are many different precise formulations of "reduction", and decision reducability is one such type. The focus throughout the rest of the chapter will be on *many-to-one reduction* ($\leq_m$) that is formulated for $\mathcal{NP}$.

6.1 Polynomial many-to-one reduction

Suppose an algorithm $\mathcal{B}$ can solve a given decision problem B in polynomial time. We want to design an algorithm $\mathcal{A}$ that uses $\mathcal{B}$ to solve decision problem A.

We can approach it in this way: given an input x of problem A, we transform x into an instance for B, and feed it as an input to $\mathcal{B}$. The output by $\mathcal{B}$ will be our output.

We can fomalize this idea as such:

> Let A and B be two decision problems. A is many-to-one reducible to B ($A \leq_m B$), if there is a polynomial time algorithm that computes the transofrmation T such that: $x \in A$ if, and only if, $T(x) \in B$.

We also obtain the following properties:

- If $B \in \mathcal{NP}$ and $A \leq_m B$, then $A \in \mathcal{NP}$. (This implies that many-to-one is a meaningful reduction for $\mathcal{NP}$ problems. If a harder problem A is in $\mathcal{NP}$, then so is the easier problem B).
- (transitivity) If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.

- (reflexivity) $A \leq_m A$.
- If $A \leq_m B$ and $B \leq_m A$, then we write $A =_m B$.

Recall that when $A \leq_m B$, the efficient algorithm $\mathcal{A}$ uses $\mathcal{B}$ to solve A in this way:

1. First, transform x into an instance for B.
2. Next, feed the transformed y as an input to $\mathcal{B}$.
3. We output whatever the output of $\mathcal{B}$ is.

Note that the algorithm $\mathcal{B}$ can be invoked at most once, and the output of $\mathcal{B}$ cannot be modified. If we remove the above restrictions, we obtain Turing reduction, written as $\leq_T$ (which is outside the scope of this chapter).

6.2 Proving $A \leq_m B$

Suppose we wish to prove that $A \leq_m B$. The proof typically has the following structure:

1. Describe a transformation T
2. Show that $x \in A \Rightarrow T(x) \in B$
3. Either show that
   (a) $x \notin A \Rightarrow T(x) \notin B$, or
   (b) $T(x) \in B \Rightarrow x \in A$.

2 and 3 together implies that $x \in A \Leftrightarrow T(x) \in B$.

7 $\mathcal{NP}$-Completeness

We extend a few definitions:

$\mathcal{NP}$-hard - A decision problem K is $\mathcal{NP}$-hard if $K \geq_m Y$ for every $Y \in \mathcal{NP}$.

$\mathcal{NP}$-complete / **$\mathcal{NP}$-C** - A decision problem K is $\mathcal{NP}$-complete if

- $K \in \mathcal{NP}$, and
- K is $\mathcal{NP}$-hard.

An $\mathcal{NP}$-hard problem is not easier than any problem in $\mathcal{NP}$.

If we can derive a polynomial time algorithm to solve a $\mathcal{NP}$-hard problem, then we can efficiently solve any problem in $\mathcal{NP}$. That is, $\mathcal{P} = \mathcal{NP}$. Inversely, if we can show that there is an $\mathcal{NP}$ problem that doesn't have a polynomial time algorithm, then we can show that $\mathcal{NP} \neq \mathcal{P}$.

7.1 Existence of $\mathcal{NP}$-C problems

The Cook-Levin theorem states that the boolean satisfiability problem is $\mathcal{NP}$-complete. Cook also showed that, given any problem A in $\mathcal{NP}$, it is possible to convert its "verifier" into an instance in 3SAT. That is, reduce A to 3SAT. In other words,s for any A in $\mathcal{NP}$, $A \leq_m$ 3SAT.

We extend the following corollary: If $K \in \mathcal{NP}$, and $Y \leq_m K$, where Y is $\mathcal{NP}$-complete, then K is $\mathcal{NP}$-complete.

It turns out that many problems in $\mathcal{NP}$ are not easier than 3SAT. Thus, many problems are equivalent to 3SAT and are thus also $\mathcal{NP}$-C.

8 Approximation algorithms

If a problem is $\mathcal{NP}$-hard, we may still tackle the problem in several ways:

- Use a fast algorithm that finds the solution for small input.
- Use an algorithm that finds an approximate solution.
- Use an algorithm that finds solutions for special types of instances.

In this section, we will focus on approximation algorithms.

Let OPT be the cost of the optimal solution (minimizing cost). If we can find a solution with cost APR that is bounded above by APR $< \epsilon$ OPT, where ϵ is a constant greater than 1, then we say that the solution is a ϵ-approximation solution, and the algorithm that finds the approximation solution is called the ϵ-approximation algorithm.

Unfortunately, it can be shown that there are problems that do not have approximation algorithms (unless $\mathcal{P} = \mathcal{NP}$)

A Boolean satisfiability problem and 3SAT

The Boolean satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.

3SAT is a boolean satisfiability problem where each clause in the formula is limited to at most three literals. For example, $(x_1 \vee x_2 \vee \sim x_3) \wedge (\sim x_2 \vee x_3 \vee \sim x_4) \wedge \ldots$