NP-Completeness

Hsiang-Hsuan (Alison) Liu

Since exponential-time algorithms can be very time-consuming, we always want to have some polynomial-time algorithms for the problems we want to solve. However, there are several problems that people have tried for decades but have not found a polynomial time algorithm: Maximum Clique, Minimum Vertex Cover, Partition, Subset Sum... There has been a long-last discussion: for these problems, which have no polynomial time algorithms (yet), should we try harder to find a polynomial time algorithm, or is it impossible to have one?

1 Polynomial-time reduction and NP-hardness

Besides solving the problems directly, we can also show that some problems are solvable by reducing them to other problems. A *reduction* is a way of converting one problem to another such that a solution to the second problem can be used to solve the first problem. That is, if problem A reduces to problem B, we can use a solution to B to solve A.

Definition 1. *Language A is* polynomial-time reducible *to language B, written $A \leq_P B$, if a polynomial-time computable function f exists, where for every w,*

$$w \in A \Leftrightarrow f(w) \in B.$$

The function f is called the polynomial-time reduction *of A to B.*

Reduction and hardness. Intuitively, if problem A reduces to problem B, B is not easier than A. That is, problem B is as hard as A, or even harder than A.

From the time complexity point of view, if problem A is polynomial-time reducible to problem B and problem B is polynomial-time solvable, then A can be solved in polynomial-time, too. The polynomial-time algorithm of A can be constructed as follows:

$\text{ALG}_A =$ " On input w :

1. Compute $f(w)$

2. Run the polynomial-time algorithm ALG_B of B on input $f(w)$.

3. If ALG_B accepts $f(w)$, *accept*. Otherwise, *reject*.

Correctness of reduction. A valid reduction should guarantee that after transforming the input of problem A, w, to an input of problem B, $f(w)$, w is a yes-instance of A if and only if $f(w)$ is a yes-instance of B. Not every transformation of the input works. Hence, to show that a reduction is valid, we have to show that the transformation covers questions about membership testing in A to membership testing in B correctly.

That is, to show that a polynomial-time reduction from A to B is valid, for any input w to A, we should prove the following three things:

- The function f can be calculated in polynomial time in the length of w.
- If $f(w)$ is a yes-instance of problem B, w is a yes-instance of problem A.
- If $f(w)$ is a no-instance of problem B, w is a no-instance of problem A.

Note that by the contrapositive statement, the last item is equivalent to "If w is a yes-instance of problem A, $f(w)$ is a yes-instance of problem B."

NP-hardness. Recall that if problem A reduces to problem B, B is at least as hard as A. The following definition uses this property to define a class of problems that are not easier than any problem in NP.

Definition 2. *A problem is* NP-hard *if all problems in NP are polynomial-time reducible to it.*

2 NP-Completeness

We always want to design a polynomial-time algorithm for the problem we want to solve. However, there are some problems people have tried for ages, but no polynomial-time algorithm was found. For these problems, we do not know if we should try harder, or the problem is impossible to be solved in polynomial-time.

In 1971, Stephen Cook published a paper and proposed that there is a problem SAT such that if the SAT problem can be solved (by a deterministic Turing machine) in polynomial time, then all problems in NP can be solve in polynomial time. In 1972, Richard Karp published another paper and proved that 21 other problems also have the property that if they can be solved in polynomial time, then P = NP.

These problems in NP whose individual complexity is related to that of the entire NP class are called *NP-complete*. More precisely, a problem is NP-complete if it is in NP and NP-hard.

Intuitively, NP-complete problems are the most difficult problems in NP, as every problem in NP can be reduced to them. If an NP-complete problem is proved to be polynomial-time solvable, all NP problems are solvable. On the contrary, if there is any problem in NP that needs more than polynomial time to solve, it must be one of the NP-complete problems.

To theoretical computer scientists, NP-complete problems are interesting as they are the breaking point for proving P = NP or not. On the practical side, the phenomenon of NP-completeness can prevent wasting time searching for polynomial-time algorithms.

3 NP-Hardness proofs

3.1 3SAT

The first NP-complete problem is the *satisfiability problem* (SAT). Variables that can take on the values TRUE (1) and FALSE (0) are called *Boolean variables*. The *Boolean operations* are `AND` ($\wedge$), `OR` ($\vee$), and `NOT` ($\neg$). A *Boolean formula* is an expression involving Boolean variables and Boolean operations. For example, $\phi = (\bar{x} \wedge y) \vee (x \vee \neg z)$ is a Boolean formula.

A Boolean formula is *satisfiable* if there exists an assignment of TRUEs (1s) and FALSEs (Os) to the variables that make the formula evaluate to TRUE (1). Let

$$\text{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula.}\}$$

The following is **Cook-Levin theorem**, which proves that SAT is a (first) NP-complete problem (the proof is very complicated, and we skip it for this course.):

Theorem 1. *SAT* $\in$ $\boldsymbol{P}$ *if and only if* $\boldsymbol{P} = \boldsymbol{NP}$.

A Boolean formula is in *conjunctive normal form* (CNF) if it is in the following form:

$$\phi = (x \vee y \vee \bar{y}) \wedge (\bar{x} \vee y) \vee (z)$$

That is, it consists of *clauses* that are connected by `AND`s, and the *literals* in each clauses are connected by `OR`s. In the example, there are three clauses:

- $(x \vee y \vee \bar{y})$
- $(\bar{x} \vee y)$
- (z)

In the first clause, there are three literals:

- x
- y
- $\bar{y}$

In this Boolean formula, there are three variables: x, y, and z. It is satisfiable as there is an assignment, $x = 0$, $y = 0$, and $z = 1$ such that the whole formula evaluates to 1.

Similar to the SAT problem, we can define the CNF-SAT problem:

$$\text{CNF-SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable conjunctive normal form Boolean formula.}\}$$

The set of conjunctive normal form Boolean formulas is a subset of the set of (arbitrary) Boolean formulas. However, the CNF-SAT problem can be proven to be NP-complete by polynomial-time reduction from SAT.

Next, we further define a more constrained form of Boolean formulas: 3CNF-formula. A Boolean formula is 3CNF if it is in conjunctive normal form and all the clauses have exactly three literals. For example,

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee x_5) \wedge (x_2 \vee x_2 \vee \overline{x_4}).$$

Let

$$\text{3SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3CNF Boolean formula.}\}.$$

Now we show that 3SAT is NP-complete. For the NP-hardness part, we prove it by polynomial-time reduction from CNF-SAT.

Proof Ideas of NP-Hardness. In the input instance of CNF-SAT, a clause may have an arbitrary number of literals. However, in a legal instance of the 3SAT problem, every clause has exactly three literals. The key here is to transform any instance of CNF-SAT to one of 3SAT while maintaining satisfiability. That is, if the original input to CNF-SAT is a satisfiable Boolean formula, the transformed input to 3SAT should also be a satisfiable Boolean formula. Similarly, if the original input to CNF-SAT is not satisfiable, the transformed input to 3SAT should also be not satisfiable.

For the clauses with less than 3 literals, the transformation is easier: we only need to duplicate one of the literals in that clause. The satisfiability of the clause is not changed since the three literals are connected by `AND`s.

For clauses with more than 3 literals, the transformation is trickier. We have to "split" a big clause into a set of multiple 3-literal clauses. The problem is, in the original big clause, it can be TRUE if there is (at least) one literal is TRUE, while the set of multiple small clauses is altogether TRUE only if there is at least one TRUE literal in *each* of the small clauses. We can resolve this issue by adding dummy literals in each small clause, letting the dummy literals be TRUE, and leaving the original literal(s) FALSE. Then there is another issue: how can we prevent the set of small clauses from being TRUE just because of the dummy literals while the original big clause is FALSE?

Theorem 2. *3SAT is NP-complete.*

Proof. First, we prove that 3SAT is in NP. (⋯ The proof is in the exercise.)

Next, we prove that 3SAT is NP-hard by reduction from CNF-SAT.

To reduce CNF-SAT to 3SAT, we convert any CNF-formula F into a 3CNF-formula F', with F is satisfiable if and only if F' is satisfiable: First, let $C_1, C_2, \cdots, C_m$ be the clauses in F. If F is a 3CNF-formula, we just set $F' = F$. Otherwise, the only reasons why F is not a 3CNF-formula are:

- Some clauses C_i has less than 3 literals, or

- Some clauses C_i has more than 3 literals.

For each clause that has less than 3 literals, we duplicate one of the literals until the total number is three.

For each clause that has more than 3 literals, we split it into several clauses and add additional dummy variables d_js to preserve the satisfiability or non-satisfiability of the original clause: each of the clauses $(x_1 \vee x_2 \vee x_3 \vee x_4 \vee \cdots \vee x_{k-1} \vee x_k$ can be replaced with the $k-2$ clauses:

$$(x_1 \vee x_2 \vee d_1) \wedge (\overline{d_1} \vee x_3 \vee d_2) \wedge (\overline{d_2} \vee x_4 \vee d_3) \wedge \cdots \wedge (\overline{d_{k-3}} \vee x_{k-1} \vee x_k).$$

The conversion can be done in $O(n \cdot m \cdot k)$ time, where n is the number of variables, m is the number of clauses, and k is the number of literals in the largest clause.

Now, we prove that the satisfiability or non-satisfiability of the 3SAT problem is preserved. That is, F is satisfiable if and only if F' is satisfiable.

If F is satisfiable, there exists a corresponding truth assignment in F' such that $F' = 1$ (TRUE). For each of the clauses with less than or equal to 3 literals in F which is true, the corresponding clause in F' is also true since we only duplicate the literals from the same clause. For each clause with more than 3 literals in F, since F is satisfiable, there must be at least one literal x_t (in the clause) which has value 1. There exists a corresponding true assignment in F': $d_j = 1$ for all $j \leq t-2$, and $d_j = 0$ for all $j \geq t-1$.

If F' is satisfiable, the corresponding truth assignment for variables $x_1, x_2, \cdots, x_n$ makes $F = 1$ (TRUE). For each of the clauses with less than or equal to 3 literals in F, all these clauses are TRUE since duplicating the literals from the same clause does not change the TRUE/FALSE of a clause. For each clause with more than 3 literals in F, since F' is satisfiable, the corresponding clauses in F' must be all TRUE as no truth value of a dummy literal can solely make more than one clause TRUE.

□

3.2 Subset Sum

In the following, we show an example of NP-completeness proof of SubsetSum. Recall that the Subset-Sum problem is defined formally as follows:

$$\text{Subset-Sum} = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_n\} \text{ and for some } \{y_1, \cdots, y_k\} \subseteq \{x_1, \cdots, x_n\},$$

$$\text{we have } \sum_i y_i = t.\}$$

That is, given a set of numbers $S = \{x_1, \cdots x_n\}$ and a target number t, a correct algorithm should return *yes* (or *accept*) if there is a subset of S such that the sum of elements in the subset is equal to the target number t.

Theorem 3. *Subset-Sum is NP-hard.*

Proof. We show that SubsetSum is NP-hard by polynomial-time reduction from 3SAT.

For any instance of 3SAT, ϕ with ℓ variables $x_1, x_2, \cdots, x_\ell$ and k clauses $c_1, c_2, \cdots, c_k$, we construct an instance of SubsetSum, set $S = y_1, y_2, \cdots y_\ell, z_1, z_2, \cdots, z_\ell, g_1, g_2, \cdots, g_k, h_1, h_2, \cdots, h_k$ and target t as follows:

- For every variable x_i in 3SAT, the two numbers y_i and z_i in S, which both have $\ell + k$ decimals.
  - The i^{th} decimal of y_i is 1, and all other decimals in the first ℓ decimals of y_i are 0. The $(\ell + j)^{\text{th}}$ decimal of y_i is 1 if and only if the clause c_j contains literal x_i.
  - The i^{th} decimal of z_i is 1, and all other decimals in the first ℓ decimals of z_i are 0. The $(\ell + j)^{\text{th}}$ decimal of z_i is 1 if and only if the clause c_j contains literal $\overline{x_i}$.
- For every clause c_j in 3SAT, create two numbers g_j and h_j in S. These two numbers are identical and consist of a single 1 at the $(\ell + j)^{\text{th}}$ decimal, and all other decimals are 0's.
- t is set as ℓ 1's followed by k 3's.

The construction can be done in polynomial time in the input size of S as we only have $O((\ell+k)^2)$ decimals, each needs at most one scanning of the input, while the input size is $\Omega(\ell+k)$.

Next, we show that for any yes-instance of the SUBSETSUM problem, the corresponding instance of the 3SAT problem is a yes-instance. Suppose that a subset of S sums to t. We construct a satisfying assignment to ϕ. First, we observe that no carry into the next decimal is needed since all the decimals in members of S are either 0 or 1 and each decimal together contains at most five 1's. Hence, to get a 1 in each of the first ℓ decimals, the subset must have either y_i or z_i for each $i = 1, 2, \cdots, \ell$ (but not both).

Now, we make the satisfying assignment. If the subset contains y_i, we assign x_i TRUE; otherwise, we assign it FALSE. Since in each of the final k decimals, the sum is always 3, and there are at most two 1's coming from g_i or h_i, there is at least one 1 coming from some y_i or z_i. Hence, this assignment satisfies ϕ

Next, we show that for any yes-instance of the PARTITION problem, the corresponding instance of the SUBSETSUM problem is a yes-instance. Suppose ϕ is satisfiable. We construct a subset of S as follows. We select y_i if x_i is assigned TRUE in the satisfying assignment and z_i if x_i is assigned FALSE. For each of the first ℓ decimals, the sum is exactly 1 since the assignment is legal (that is, exactly one of y_i and z_i is true for every i). Furthermore, each of the last k decimals is between 1 to 3 because each of the 3-literal clauses has at least one TRUE literal. By selecting enough of the g and h numbers to bring each of the last k decimals up to 3, the large target is hit. □

3.3 Clique

A *clique* is a set of vertices where each pair of vertices are adjacent to each other. The *maximum clique problem* is: given a graph G, what is the size of the maximum clique in G? The decision version of the maximum clique problem is:

$$\text{CLIQUE} = \{\langle G, k\rangle \mid \text{ There is a clique in } G \text{ of size at least } k.\}$$

We prove that the CLIQUE problem is NP-complete. The NP-hardness is proven by polynomial-time reduction from 3SAT.

Proof Ideas of NP-Hardness. To reduce 3SAT to CLIQUE, we have to transform any 3CNF Boolean formula ϕ to a graph G and make sure that ϕ is satisfiable if and only if there is a k-clique in G.

Theorem 4. CLIQUE *is NP-complete.*

Proof. First, we show that CLIQUE is in NP. Let string c that encodes a clique with size k in G as a certificate.

$A =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G.
2. Test whether G contains all edges connecting vertices in c
3. If both 1 and 2 pass, *accept*; otherwise, *reject*."

Step 1 takes at most $|c| = k$ times of scanning through the input. Step 2 takes at most $|c|^2$ times of scanning through the input. Hence, A runs in polynomial time in the input length.

Now we show that CLIQUE is NP-hard by polynomial-time reduction from 3SAT. For any instance of 3SAT, $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, we generate an instance of CLIQUE, $G = (V, E)$ and k, as follows: For each clause C_i containing three literals, ℓ_{i_1}, ℓ_{i_2}, and ℓ_{i_3}, there are three vertices in V that are corresponding to the three literals.

For any pair of vertices ℓ_x and ℓ_y in V, there is an edge (ℓ_x, ℓ_y) in E if and only if

- The two vertices ℓ_x and ℓ_y come from different clauses, and
- The corresponding literals of ℓ_x and ℓ_y are not the negation to each other.

Finally, we let k equal to m, the number of clauses in ϕ. The construction can be done in polynomial time since $|V| = 3m$ and there are $O(m^2)$ edges, where each of the edges needs constant time to check.

Now, we show that the reduction works by showing that there is a satisfying assignment to ϕ if and only there is a k-clique in G. Suppose that ϕ has a satisfying assignment, we construct a k-clique by selecting one of the vertices which are corresponding to a literal with "TRUE" value from each of the clauses. Since ϕ is satisfiable, there must be one of such a literal in every clause. As the satisfying assignment is feasible, every variable is assigned to either TRUE or FALSE but not both. Hence, there must be an edge between two vertices picked from different clauses. Therefore, the picked vertices form a k-clique.

Suppose that G has a clique V' of size k. No edges in G connect vertices in the same clause, so V' contains exactly one vertex from each of the k clauses. We assign the value TRUE to the corresponding literal. It is a feasible assignment since there is no edges between literals corresponding to x and $\overline{x}$ for each variable x. Hence, each clause has one literal, which is assigned TRUE, and the formula ϕ is satisfied.

□

3.4 Bin Packing

Recall the (offline) Bin Packing problem, where we aim to pack n items with size within $(0, 1]$ into the minimum number of capacity-1 bins. The decision version is defined as follows:

BinPacking $= \{\langle U, k\rangle \mid$ The items in U can be partitioned into at most k disjoint subsets such that the total size of the items in each subset is no more than $1\}$.

The Bin Packing problem is NP-complete. The NP-hardness can be shown by polynomial-time reduction from Partition.

Theorem 5. *BinPacking is NP-complete.*

Proof. To prove that BinPacking is in NP, we use a k-partition of items in U as the certificate. In detail, the k-partition can be encoded in an $|U|$-array, where the i-th entry is the index of the part the i-th item belongs to. The certificate has total size $|U| \log k$, which is polynomial in the input $\langle U, k\rangle$. The verifier should check if this partition is a proper partition of U and if each subset has a sum of size no more than 1. The checking time is in a polynomial of the number of elements in U.

To prove the NP-hardness, we show that Partition $\leq_P$ BinPacking. For any instance of Partition, S, we construct an instance of BinPacking, (U, k) as follows. For each element $a_i \in S$, there is a corresponding item $u_i \in U$ and $size(u_i) = \frac{2 \cdot a_i}{X}$, where X is half of the sum of all elements in S. We set $k = 2$. The construction can be done in polynomial time.

Now, we prove that the reduction works. Suppose there is a partition of S, S_1 and S_2. For all elements $a_i \in S_1$, the sum is X. The sum of corresponding u_i's is 1, so the corresponding items can be placed in one bin. It also holds for S_2. Hence, the items can be packed into 2 bins.

For the other direction, suppose that the items in U can be packed in two bins. Each of the bin has total size 1 since the total size of all items in U is $\Sigma_i\ size(u_i) = \frac{\Sigma_i\ a_i}{X} = 2$. The corresponding two subsets of S has equal size and form a partition. □

General case, special case, and hardness. Note that in the NP-hardness proof, we reduce the Partition problem to a special case of BinPacking problem, where the items can be packed into two bins. Therefore, more precisely, by this reduction, we show that the special case of BinPacking is NP-hard. Then, how about the general case of BinPacking, where the packed-in-two-bins restriction is relaxed? Intuitively, the general case is not easier than the special case (recall the "if one can solve the general case, they can use the algorithm for the general case to solve the special case" argument). Therefore, the general case is at least NP-hard if the special case is NP-hard. This can be formally proved by reducing the special case to the general case in polynomial time (if you want).