AI2615 算法设计与分析 2020-2021 春季学期

Lecture 14 – NP-Hardness and NP-Completeness

2021 年 5 月 28 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

In previous lectures, we have said multiple times a problem is *NP-hard*. In this lecture, we will define it in an informal way. First of all, let us define what is a *problem*.

Definition 1. A *decision problem* is a function $f : \Sigma^* \to \{0,1\}$.

In the definition above, Σ^* is the set of all strings (with an arbitrarily length) using alphabets from the set Σ. To be specific, $\Sigma^* = \bigcup_{n\geq 0} \Sigma^n$, where Σ^n is the Cartesian product of n Σ's (for example, $\Sigma^2 = \Sigma \times \Sigma = \{xy \mid x, y, \in \Sigma\}$). A string in Σ^* encodes an instance of a problem, and 0, 1 represent "no" and "yes". For example, in SAT, a string encodes a CNF formula. f maps it to 1 if this formula is satisfiable, and maps it to 0 if it is not a valid encoding of a CNF formula or it is not satisfiable.

1 P versus NP

Given a problem f, we say that $\mathcal{A}$ is an *algorithm* for f if $\mathcal{A}(x) = f(x)$ always holds. We say that f is computable in polynomial time if there exists an algorithm $\mathcal{A}$ such that, in addition to $\mathcal{A}(x) = f(x)$, $\mathcal{A}(x)$ always terminates in $|x|^{O(1)}$ time.

Definition 2. P is the set of decision problems that are computable in polynomial time.

Notice that P only includes those decision problems. Although many problems we have seen so far are not decision problems, but most of them can be formulated as decision problems in a natural way. For example, for the minimum spanning tree problem, a natural formulation to a decision problem can be: given an undirected weighted graph $G = (V, E, w)$ and a number $k \in \mathbb{Z}^+$, does G have a spanning tree with total weight at most k? In fact, if the decision problem is solvable in polynomial time, we can simply guess the weight of the minimum spanning tree by a binary search.

Definition 3. NP (short for "non-deterministic polynomial-time") is the collection of decision problems $f : \Sigma^* \to \{0,1\}$ for which there exists an algorithm $\mathcal{V} : \Sigma^* \times \Sigma^* \to \{0,1\}$ such that

1. for any $x \in \Sigma^*$ with $f(x) = 1$, there exists $y \in \Sigma^*$ such that $\mathcal{V}(x, y) = 1$,
2. for any $x \in \Sigma^*$ with $f(x) = 0$, then for any $y \in \Sigma^*$, we have $\mathcal{V}(x, y) = 0$, and
3. $\mathcal{V}(x, y)$ always terminates in $|x|^{O(1)}$ time.

If a decision problem f is in NP, the above definition says the followings. If x is a yes instance ($f(x) = 1$), then there exists a "proof" y such that the algorithm $\mathcal{V}$ can verify that $f(x) = 1$ with the help of y. This algorithm $\mathcal{V}$ serves as a *verifier*, which takes the instance x and a proof y as inputs, and outputs the decision whether $f(x) = 1$. If x is a no instance ($f(x) = 0$), then any proof y will not be able to fool the verifier $\mathcal{V}$ by making it output 1. In addition, we require that the verifier $\mathcal{V}$ should run in a polynomial time in terms of $|x|$. In particular, this requires the length of the proof, $|y|$, should be bounded by a polynomial of $|x|$ as well, for otherwise it takes more than a polynomial time for $\mathcal{V}$ to simply read the input (x, y). In the case $f(x) = 1$, the string y making $\mathcal{V}(x, y) = 1$ is called a *certificate*.

Theorem 4. *SAT ∈ NP.*

Proof. Let $\mathcal{V}(\cdot,\cdot)$ be defined as follows: $\mathcal{V}(x, y)$ outputs 1 if 1) x is a valid encoding of a CNF formula, 2) y is a valid encoding of an assignment, and 3) putting y into x evaluates to **true**. Otherwise, $\mathcal{V}(x, y)$ outputs 0. It is easy to see that $\mathcal{V}$ satisfies the three conditions in Definition 3. □

In a mathematical view, P can be viewed as those simple mathematical statements whose correctness can be checked easily, while NP is the set of those statements that can be verified if proofs are given.

Theorem 5. *P ⊆ NP.*

Proof. For any problem $f \in \mathrm{P}$, there exists a polynomial time algorithm $\mathcal{A}$ such that $\mathcal{A}(x) = f(x)$. Let $\mathcal{V}(x, y)$ be defined such that $\mathcal{V}(x, y) = \mathcal{A}(x)$. In particular, $\mathcal{V}$ is oblivious to the second argument y. It is easy to verify that $\mathcal{V}$ satisfies all the three conditions in Definition 3. □

The most prominent and important open problem, which you may have been heard about all the time, is whether $\mathrm{P} = \mathrm{NP}$. There are two possibilities: $\mathrm{P} = \mathrm{NP}$ and $\mathrm{P} \subsetneq \mathrm{NP}$. If the former is true, then every problem that can be verified in a polynomial time with the help of a certificate can also be solved in a polynomial time without knowing the certificate. If the latter is true, then there exists a problem in NP that cannot be solved in a polynomial time.

2 Examples for Problems in P and NP

In this section, we will see some examples of problems in P and NP. Table 1 lists a few problems that are in NP but not known to be in P (the left column), and a few problems that are known to be in P (the right column). In each line, the pair of two problems share many similarities. However, some subtle differences make the tractability of the two problems fundamentally different. We will define and remark about the problems in each row one-by-one.

1. We have seen that 2-SAT can be solved in a polynomial time by considering the strongly connected components in a directed graph. SAT, however, is not known to be solvable in a polynomial time.

	Problems in NP that are not known to be in P	Problems in P
1	SAT	2-SAT
2	Travel Salesman Problem (TSP)	Minimum Spanning Tree Problem (MST)
3	Hamiltonian Circuit	Eulerian Circuit
4	Balanced Cut	Min-Cut
5	Integer Programming and Zero-One Equations (ZOE)	Linear Programming
6	Perfect 3D-Matching	Perfect Matching
7	Longest Path	Shortest Path
8	Independent Set	Independent Set on Trees
9	Vertex Cover	Vertex Cover on Trees
10	Clique	Clique on Trees
11	Knapsack	
12	Subset-Sum	
13	Max-Cut	

Table 1: Examples of problems in P and NP

2. The travel salesman problem can be formulated by a decision problem as well: given a weighted graph $G = (V, E, w)$ and $k \in \mathbb{Z}^+$, does there exist a tour that visit each vertex exactly once with the total length at most k? Both TSP and MST are the problems about a subgraph. The difference is that TSP looks for a *path* while MST looks for a *tree*. We have seen that MST can be solved in a polynomial time by the greedy algorithm (Kruskal or Prim), while we do not know if TSP is in P.
3. The Hamiltonian circuit problem asks if a graph contains a cycle that contains each *vertex* exactly once, and the Eulerian circuit problem asks if a graph contains a cycle that contains each *edge* exactly once. Although the two problems are very similar, we know that the Eulerian circuit problem is in P (the graph is a yes instance if and only if all the vertices have even degrees), while the Hamiltonian Circuit problem is a well-known hard problem.
4. Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{Z}^+$, the balance cut problem asks if we can partition V into S and T with $|S|, |T| \geq \frac{n}{3}$ such that the number of edges between S and T is at most k, and the min-cut problem asks if we can partition V into S and T such that the number of edges between S and T is at most k. The only difference is that the balance cut additionally requires that S and T are somehow "balanced": $|S|, |T| \geq \frac{n}{3}$. While the balance cut problem is not known to be in P, the min-cut problem can be solved in a polynomial time. In fact, we can even solve the edge-weighted version of min-cut in a polynomial time. We have learned how to do this when the graph has a source s and a sink t (which can be solved by computing a max-flow), can you extend the method to the case without s and t?

5. We have seen that solving a linear program is easy and solving an integer program is hard. Indeed, a special case of the integer programming problem, the *Zero-One Equations* (ZOE) problem, is already hard. In ZOE, we are given an $n \times n$ zero-one matrix $A \in \{0,1\}^{n\times n}$, and we are asked if there exists a zero-one vector $\mathbf{x} \in \{0,1\}^n$ such that $A\mathbf{x} = \mathbf{1}$.
6. We have seen that the problem of deciding if a bipartite graph contains a perfect matching can be solved in a polynomial time (by converting it to a max-flow problem). In fact, this problem is even polynomial time solvable for general graphs. The perfect 3D-Matching problem is defined as follows. Given a tripartite hyper-graph $G = (A, B, C, E)$ where vertices are partitioned into A, B, C and each hyper-edge $e \in E$ contains exactly one vertex in each of A, B, C (i.e., $e = (a, b, c)$ for some $a \in A$, $b \in B$ and $c \in C$), decide if G contains a perfect matching (a collection of edges that contains each vertex exactly once). This problem can be imagined in this way: suppose A contains a set of boys, B contains a set of girls, and C is a set of pets; each hyper-edge is an admissible boy-girl-pet triple (that can form a family), and the goal is to match the boys, the girls with the pets such that each of them is matched in exactly one family. The perfect 3D-Matching problem is not known to be in P.
7. Given an undirected graph $G = (V, E)$, two vertices s, t and an integer $k \in \mathbb{Z}^+$, the shortest path problem (formulated as a decision problem) asks if there exists a path from s to t with length at most k, and the longest path problem (formulated as a decision problem) asks if there exists a path from s to t with length at least k that do not visit a vertex more than once. The shortest path problem can be solved in polynomial time (breadth-first search), even if the graph is edge-weighted (Dijkstra), and even if the graph can contain negatively-weighted edges (Bellman-Ford). The longest path problem, however, is not known to be in P.
8. We have seen that the independent set problem, the vertex cover problem, and the clique problem are hard in general. However, these problems defined on trees are all polynomial time solvable by dynamic programming, as we have learned in previous lectures.
9. Some additional NP problems that are not known in P are Knapsack, subset-sum and max-cut. We have seen Knapsack and max-cut in previous lectures. In the subset-sum problem, we are given a collection of positive integers S and an integer $k \in \mathbb{Z}^+$, and we are asked if S contains a subset whose sum is exactly k.

For all the problem in the right column in Table 1, we have seen in previous lectures that they can be solved in a polynomial time. We leave it as an exercise to show that all the problems in the left column are in NP, or their corresponding decision problems are in NP.

The problems in the left column are all hard problems. One natural question is, among those problems, are there any problems that are "harder" than the others? To answer this question, we will need a notion called *reduction*, which will be discussed in the next section. It turns out that all the problems in the left column are "equally hard", and they are "the hardest problems" in NP.

3 Reduction

Definition 6. Given two decision problems f and g, a *(Karp) reduction* from f to g is a polynomial time algorithm $\mathcal{R} : \Sigma^* \to \Sigma^*$ such that, for any $x \in \Sigma^*$, we have $f(x) = 1$ if and only if $g(\mathcal{R}(x)) = 1$. If there exists a reduction from f to g, we say that f is reducible to g, or g can be reduced from f. It is denoted by $f \leq_k g$.

The reduction defined above is called a *Karp reduction*, or a *many-to-one reduction.* It always maps a yes instance of f to a yes instance of g, and a no instance of f to a no instance of g. The mapping do not need to be one-to-one or onto. In particular, it is allowed that, for both x_1, x_2 with $f(x_1) = f(x_2) = 1$, we have $\mathcal{R}(x_1) = \mathcal{R}(x_2)$, as long as $g(\mathcal{R}(x_1)) = g(\mathcal{R}(x_2)) = 1$. This is why a Karp reduction is called a many-to-one reduction.
$f \leq_k g$ indicates that g is at least as hard as f. In fact, if we have a polynomial time algorithm to solve g, we can have a polynomial time algorithm to solve f. For a given input x, we can first compute $\mathcal{R}(x)$ (which can be done in polynomial time, as $\mathcal{R}$ is required to be polynomial time computable by definition), and then use the algorithm solving g to see if $g(\mathcal{R}(x)) = 1$. By using reductions, we can formally prove that one problem is no easier than the other.

Definition 7. A decision problem f is *NP-complete* if 1) $f \in \mathrm{NP}$ and 2) for any $g \in \mathrm{NP}$ we have $g \leq_k f$.

The NP-complete problems are those hardest problems in NP. The very first question may be, does a NP-complete problem exist? In fact, this is the case.

Theorem 8 (Cook-Levin Theorem). *SAT is NP-complete.*

We will not formally prove Cook-Levin Theorem, and we will give an intuitive argument instead. For any problem in NP, there exists a verifier $\mathcal{V}$ that satisfies the three conditions in Definition 3. Suppose we construct a *computation tableau* for $\mathcal{V}(x, y)$ that records every step in the execution of the algorithm $\mathcal{V}$. We can use a CNF formula to describe the validity of the tableau. The variables for the CNF formula are all the entries in the tableau. We can build a CNF formula to to test if all the steps in the execution of $\mathcal{V}$ are correct, and if the output of $\mathcal{V}(x, y)$ is 1. This shows that every NP problem can somehow be formulated by a CNF formula. Therefore, if we can solve SAT in a polynomial time, we can solve every NP problems. The relationship between P, NP and NP-complete problems are shown in Fig. 1.
A stronger version of reduction is called a *Cook reduction*, or a *polynomial time Turing reduction.* This reduction better describes the idea of using one problem to solve another.

Definition 9. Given two problems f and g, we say that f is *Cook reducible*, or *polynomial time Turing reducible*, to g, if there exists a polynomial time algorithm solving f that makes use of an oracle for solving g. This is denoted by $f \leq_c g$.

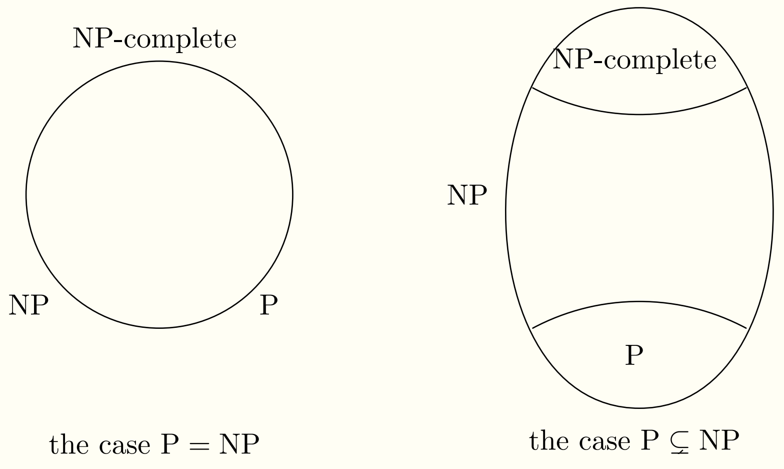
NP-complete
NP
P
the case $\mathrm{P} = \mathrm{NP}$
NP-complete
NP
P
the case $\mathrm{P} \subsetneq \mathrm{NP}$

Figure 1: The relationship between P, NP and NP-complete problems: in the case $\mathrm{P} = \mathrm{NP}$, the three sets are identical (prove that every problem in P is NP-complete if $\mathrm{P} = \mathrm{NP}$); in the case $\mathrm{P} \subsetneq \mathrm{NP}$, the set of problems in P is disjoint to the set of NP-complete problems (prove that if a NP-complete problem is in P, then $\mathrm{P} = \mathrm{NP}$).

In a Cook reduction, we do not need a mapping that maps yes (no) instances of f to yes (no) instances of g. All we need is that f is polynomial time solvable if we have a "black box" that can solve g in polynomial time. In algorithm design aspect, both a Karp reduction and a Cook reduction indicate that a problem is no easier than another. However, in complexity theory, these two reductions are quite different. Clearly, a Cook reduction is more powerful than a Karp reduction, it is an open problem in complexity theory if a Cook reduction is strictly more powerful.

Definition 10. A problem $f : \Sigma^* \to \Sigma^*$ is NP-hard if $g \leq_c f$ for any $g \in \mathrm{NP}$.

There are two differences between NP-hardness and NP-completeness. Firstly, an NP-hard problem do not have to be a decision problem. Even if an NP-hard problem is a decision problem, it does not have to be in NP. Secondly, we define NP-hardness using Cook reductions, while it is crucial that NP-completeness can only be defined using Karp reductions.

We have identified the first NP-complete problem, SAT. To identify more NP-complete problems, we do not need to go through the arguments in the proof of Cook-Levin theorem again. To prove a decision problem f is NP-complete, all we need to do is to prove that 1) $f \in \mathrm{NP}$ and 2) $\mathrm{SAT} \leq_k f$. In general, if we know a decision problem f is NP-complete, to show that g is NP-complete, we only need to prove 1) $g \in \mathrm{NP}$ and 2) $f \leq_k g$.

Theorem 11. *If $f, g \in NP$, f is NP-complete, and $f \leq_k g$, then g is NP-complete.*

Exercise 12. Prove Theorem 11. Hint: prove that $h \leq_k f$ and $f \leq_k g$ imply $h \leq_k g$. Notice that this implies the theorem: for any $h \in \mathrm{NP}$, we have $h \leq_k f$ since f is NP-complete, and we have $f \leq_k g$; this implies $h \leq_k g$, so g is NP-complete by definition.

To conclude this section, we will see an example. Given an undirected graph $G = (V, E)$ and two vertices s and t, the *Hamiltonian path* problem asks if there exists a path from s to t that visits each vertex exactly once. Suppose we know that the Hamiltonian path problem is NP-complete (which is true in fact), and we want to show that the Hamiltonian circuit problem is NP-complete. Firstly, it is easy to see that the Hamiltonian circuit problem is in NP, as the sequence of the edges in the circuit can serve as a certificate. Secondly, there is a simple reduction from the Hamiltonian path problem to the Hamiltonian circuit problem.

Proposition 13. HamPath$\leq_k$HamCircuit

Proof. Given a Hamiltonian path instance $G = (V, E)$, we construct a Hamiltonian circuit instance as follows. Create a vertex x and connect it to both s and t. Let G' be the resultant graph. If G has a Hamiltonian path from s to t, then appending the path $s \to \cdots \to t \to x \to s$ gives a Hamiltonian circuit in G'. Thus, a yes instance of the Hamiltonian path problem is mapped to a yes instance of the Hamiltonian circuit problem. We also need to prove that a no instance of the Hamiltonian path problem is always mapped to a no instance of the Hamiltonian circuit problem. We prove the contra-positive of this. Suppose G' contains a Hamiltonian circuit. It is easy to see that s-x-t must be a part of the circuit (this is the only way to reach x and to exit x). Then, by removing x from the circuit, we have a Hamiltonian path from s to t in G.
Finally, this mapping can certainly be computed in a polynomial time. □

Therefore, we have proved that 1) the Hamiltonian circuit problem is in NP and 2) there is a Karp reduction from the Hamiltonian path problem to the Hamiltonian circuit problem. Thus, the Hamiltonian circuit problem is NP-complete.

4 More NP-complete Problems

We will see the proofs for more NP-complete problems in this section. In fact, all the problems in the left column of Table 1 are NP-complete. The web of reductions is presented in Fig. 2. In Fig. 2, an arrow from problem A to problem B represents $A \leq_k B$. Since all the problems in the figure are in NP and there is a path from the known NP-complete problem SAT to each of them, all the problems are NP-complete. On the other hand, since SAT is NP-complete, all the NP problems can be reduced to SAT. Therefore, there should be an arrow from each problem in the figure back to SAT, which is omitted in the figure.
In this section, we will show some of the reductions in Fig. 2.

Theorem 14. *SAT$\leq_k$ 3-SAT.*

Proof. Given a SAT instance ϕ, we will construct a 3-SAT instance ϕ' as follows. Consider a clause $(a_1 \vee a_2 \vee \cdots \vee a_k)$ in ϕ that contains more than three literals. Here, each literal a_i can be a variable, or its

$$\begin{tikzcd}
 & \text{SAT} \arrow[d] & & & & \\
 & \text{3-SAT} \arrow[r] \arrow[d] & \text{3-SAT } (\Delta \leq 3) \arrow[rr] & & \text{3D-Matching} \arrow[d] & \\
 & \text{Independent Set} \arrow[dl] \arrow[dr] & & & \text{ZOE} \arrow[dl] \arrow[dr] & \\
\text{Vertex Cover} & & \text{Clique} & \text{Subset-Sum} & & \text{Hamiltonian Circuit} \arrow[d] \\
 & & & & & \text{TSP}
\end{tikzcd}$$

Figure 2: The web of reductions. The problem “3-SAT ($\Delta \leq 3$)” will be defined in Problem 15.

negation. We convert it to $k-2$ clauses in ϕ' such that each clause contains three literals:

$$(a_1 \vee a_2 \vee y_1) \wedge (\bar{y}_1 \vee a_3 \vee y_2) \wedge (\bar{y}_2 \vee a_4 \vee y_3) \wedge \cdots \wedge (\bar{y}_{k-4} \vee a_{k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee a_{k-1} \vee a_k), \tag{1}$$

where $y_1, \ldots, y_{k-3}$ are new variables introduced. This defines a mapping from a SAT instance ϕ to a 3-SAT instance ϕ'. It can be clearly computed in a polynomial time.

If ϕ is satisfiable, then each clause $(a_1 \vee a_2 \vee \cdots \vee a_k)$ contains at least one literal a_i that evaluates to **true**. In (1), if we let $y_j =$ **true** for all $j \leq i-2$ and $y_k =$ **false** for all $k \geq i-1$, then all clauses in (1) evaluate to **true**. In particular, the clause containing a_i, which is $(\bar{y}_{i-2} \vee a_i \vee y_{i-1})$, is **true**, as a_i is **true**. For each clause before this, the last literal is **true**; for each clause after this, the first literal is **true**. Therefore, we have proved ϕ' is satisfiable.

If ϕ is not satisfiable, we aim to show that ϕ' is also not satisfiable. We prove the contra-positive of this: suppose ϕ' is satisfiable, we show that ϕ is also satisfiable. Consider a satisfying assignment to ϕ'. We aim to show that, under this assignment, each clause $(a_1 \vee a_2 \vee \cdots \vee a_k)$ in ϕ contains at least one literal that evaluates to **true**. In fact, if this is not the case, then there exists at least one clause in (1) that cannot be satisfied, which contradict to that the assignment to ϕ' is a satisfying assignment. To see this, if $a_1 = \cdots = a_k =$ **false**, then we must have $y_1 =$ **true** to make the first clause in (1) evaluate to **true**, and subsequently we must have $y_2 =$ **true** to make the second clause evaluate to **true**. Following these, we must have $y_1 = y_2 = \cdots = y_{k-3} =$ **true**. However, in this case, the last clause cannot be **true**. □

The theorem above says that 3-SAT is at least as hard as SAT. On the other hand, it is obvious that 3-SAT is at most as hard as SAT, as it is a special case of SAT. Therefore, we can conclude that 3-SAT is as hard as SAT. In fact, if we want to prove that a NP problem is NP-complete, it is often easier to reduce it from

3-SAT than from SAT. This is because 3-SAT is seemingly an easier problem compared to SAT, as it is a special case. Proving a special case being hard is a stronger hardness result, which makes it more powerful when used in a reduction to show other NP-complete/NP-hard problems.
We will further show that the following problem, which is even a special case of 3-SAT, is still NP-complete.

Problem 15 (3-SAT ($\Delta \le 3$)). Given a 3-CNF formula ϕ such that each variable x_i (including its negation $\bar{x}_i$) appears at most three times in ϕ which further satisfies:

- the literal x_i appears at most twice, and
- the literal $\bar{x}_i$ appears at most twice,

decide if ϕ has a satisfying assignment.

Theorem 16. *3-SAT ($\Delta \le 3$) is NP-complete.*

Proof. The problem is clearly in NP, as a satisfying assignment can be used as a certificate in the case the instance is a yes instance. To show it is NP-complete, we prove that 3-SAT $\le_k$ 3-SAT ($\Delta \le 3$).
Given a 3-SAT instance ϕ, we construct a 3-SAT ($\Delta \le 3$) instance ϕ' as follows. Consider an arbitrary variable x such that it appears m times in ϕ and its negation $\bar{x}$ appears n times in ϕ. We construct $m+n$ variables $x_1, \dots, x_{m+n}$ for ϕ'. ϕ' is obtained from ϕ by replacing the i-th appearance of x to x_i and the j-th appearance of $\bar{x}$ to $\bar{x}_{m+j}$. In addition, we construct $m+n$ extra clauses in ϕ' as follows:

$$(\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge \cdots \wedge (\bar{x}_{m-1} \vee x_m) \wedge (\bar{x}_m \vee \bar{x}_{m+1}) \wedge (x_{m+1} \vee \bar{x}_{m+2}) \wedge \cdots \wedge (x_{m+n-1} \vee \bar{x}_{m+n}) \wedge (x_{m+n} \vee x_1).$$

It is easy to check that the $m+n$ clauses ensure $x_1 = \cdots = x_m = \bar{x}_{m+1} = \cdots = \bar{x}_{m+n}$. Therefore, the $m+n$ variables $x_1, \dots, x_{m+n}$ in ϕ' simulate the value of x in ϕ. It is then easy to see that ϕ is satisfiable *if and only if* ϕ' is satisfiable, and we leave the remaining details as an exercise.
Finally, it is easy to see that ϕ' is a valid 3-SAT ($\Delta \le 3$) instance, and it can be constructed in a polynomial time. □

Next, the theorem below shows that the independent set problem is NP-complete.

Theorem 17. *3-SAT $\le_k$ Independent Set*

Proof. Given a 3-SAT instance ϕ, we construct an independent set instance $(G = (V, E), k)$ as follows. For each clause containing three literals, we construct three vertices, and build an edge between every pair of them. The clauses containing two literals are treated similarly. After this, for any two vertices corresponding to the same variable such that one of them is the negation of the other, we build an edge between them. Finally, k is set to m, the number of clauses in ϕ. It is clear that the construction can be done in a polynomial time. An example of this is shown in Fig. 3.
If ϕ is satisfiable, we aim to show that the graph contains an independent set with $k = m$ vertices. Consider a satisfying assignment of ϕ. For each clause, we choose a representative literal whose value is **true**, and

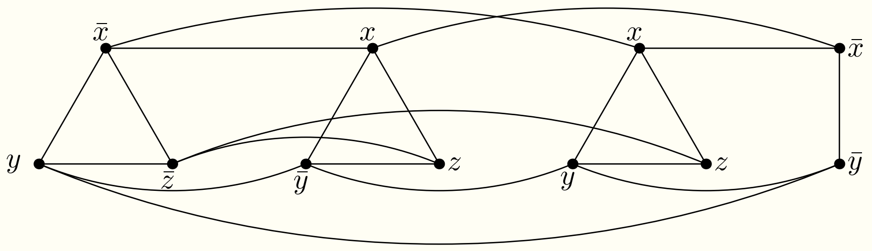

Figure 3: An example of the constructed independent set instance for the 3-SAT instance $(\bar{x} \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee z) \wedge (x \vee y \vee z) \wedge (\bar{x} \vee \bar{y})$.

we select the corresponding vertex in G. Since we have m literals, we have chosen $k = m$ vertices in G. Moreover, these k vertices form an independent set. To show that there is no edge between any two vertices among those k vertices, we first notice that there are two types of edges we have constructed: 1) the edge between two literals in the same clause, and 2) the edge connecting a variable to its negation. There cannot be any type 1 edge between any two vertices, as we have only chosen one vertex in the triangle representing each clause. There cannot be any type 2 edge between any two vertices, as the vertices we have chosen corresponding to the literals with value **true**, and we cannot have a variable and its negation being both **true**.
If ϕ is not satisfiable, we aim to show that all the independent sets in the graph have sizes at most $k-1$. We prove the contra-positive. Suppose the graph contains an independent set with k vertices. It must be that each triangle representing a clause contains exactly one vertex in the independent set (why?). We assign the values to the variables that are indicated by the vertices in the independent set. By doing this, the value of each clause in ϕ is guaranteed to be **true**. Notice that there may be variables whose values are not decided yet (there may be variables x such that both x and $\bar{x}$ are not selected as vertices), we can just assignment them with an arbitrary value, which will not make any clause **false**. Thus, we have proved that ϕ is satisfiable if the graph contains an independent set with k vertices. □

We leave it as an exercise to prove that the vertex cover problem and the clique problem are both NP-complete.

Exercise 18. Prove that Independent Set $\leq_k$ Vertex Cover.

Exercise 19. Prove that Independent Set $\leq_k$ Clique.

Finally, we will show that perfect 3D-matching is NP-complete. The reduction is slightly more tricky.

Theorem 20. *3-SAT ($\Delta \leq 3$) $\leq_k$ Perfect 3D-Matching.*

Proof. Given a 3-SAT ($\Delta \leq 3$) instance ϕ, we construct a perfect 3D-matching instance $G = (A, B, C, E)$ as follows. Firstly, for each variable x in ϕ, we construct a *variable gadget* that contains two vertices a_{x_1}, a_{x_2}

in A, two vertices b_{x_1}, b_{x_2} in B, and four vertices $c_{x_1}, c_{x_2}, c_{x_3}, c_{x_4}$ in C, and we construct four hyper-edges for this gadget: $e_1 = \{c_{x_1}, a_{x_1}, b_{x_1}\}$, $e_2 = \{c_{x_2}, a_{x_1}, b_{x_2}\}$, $e_3 = \{c_{x_3}, a_{x_2}, b_{x_2}\}$ and $e_4 = \{c_{x_4}, a_{x_2}, b_{x_1}\}$. We will keep $a_{x_1}, a_{x_2}, b_{x_1}, b_{x_2}$ be "internal" to this gadget, such that each of them will not be included in any other hyper-edges. We will view the four vertices $c_{x_1}, c_{x_2}, c_{x_3}, c_{x_4}$ be the four "terminals" that can "interact" with other gadgets or the remaining part of the graph. The gadget is shown in Fig. 4, and we will use the picture on the right-hand side to represent this gadget. We will use G_x to denote this gadget.

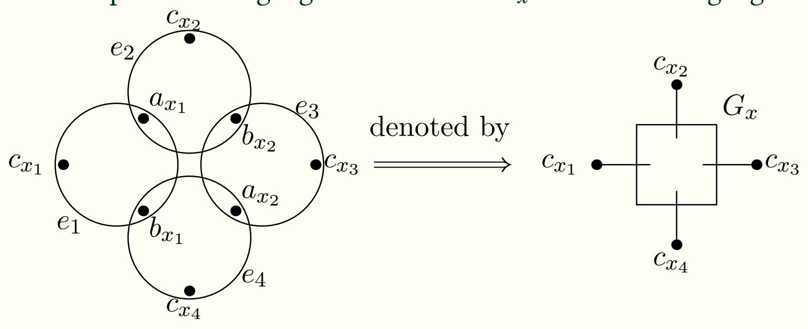

Figure 4: The variable gadget.

In this gadget, we can choose either $\{e_1, e_3\}$ or $\{e_2, e_4\}$ as a part of the perfect matching. We will make the two options to represent **true** and **false** for this variable. In particular, $x =$ **true** will correspond to $\{e_1, e_3\}$ being chosen, and $x =$ **false** will correspond to $\{e_2, e_4\}$ being chosen. We will call c_{x_2}, c_{x_4} the *true terminals* of G_x, and c_{x_1}, c_{x_3} the *false terminals* of G_x. Specifically, if x is assigned **true**, then $\{e_1, e_3\}$ are chosen, while the true terminals c_{x_2}, c_{x_4} are available to be matched by outside edges; if x is assigned **false**, then $\{e_2, e_4\}$ are chosen, while the false terminals c_{x_1}, c_{x_3} are available to be matched by outside edges.

Secondly, for each clause $s = (d_1 \vee d_2 \vee d_3)$, we construct two vertices $a_s \in A$ and $b_s \in B$. For each of the three literals d_1, d_2, d_3, if literal d_i $(i = 1, 2, 3)$ is a variable x, then build an edge that contains a true terminal of G_x, and the two vertices a_s, b_s; if literal d_i is a negation $\bar{x}$, then build an edge that contains a false terminal of G_x, and the two vertices a_s, b_s. Figure 5 gives an example of this.

We make sure that each terminal is used in at most one clause. Notice that, the requirement that each of x and $\bar{x}$ appears at most twice makes sure that, after constructing all the clause gadgets, the terminals are not running out (since each gadget contains two true terminals and two false terminals).

Lastly, we create $2n - m$ vertices $a_1, a_2, \ldots, a_{2n-m}$ in A and $2n - m$ vertices $b_1, b_2, \ldots, b_{2n-m}$ in B. For each of the $4n$ terminals, build an edge containing it and the two vertices a_i, b_i for each $i = 1, \ldots, 2n - m$. We have built a total of $4n(2n - m)$ edges. The reason for using the number $2n - m$ will be apparent soon. Before we finish the construction, we need to prove $2n \geq m$. Indeed, each clause contains at least two literals, so the total number of literals is at least $2m$. On the other hand, each variable appears at most three times, so the total number of literals is at most $3n$. Thus, $3n \geq 2m$, which implies $2n > m$.

Suppose ϕ has a satisfying assignment. We aim to show that there is a perfect 3D-matching in G. Consider

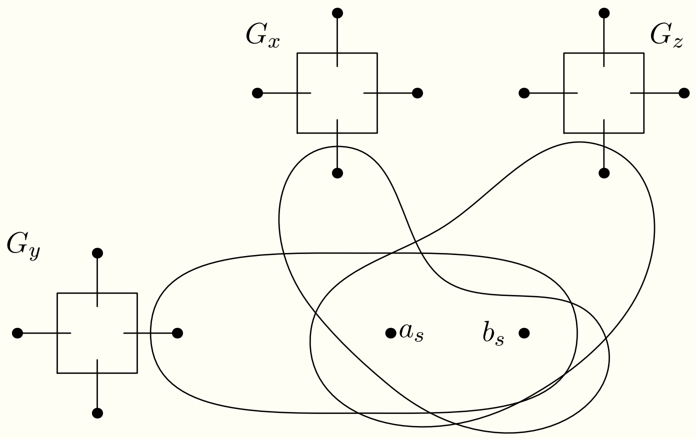

Figure 5: The gadget representing the clause $s = (x \vee \bar{y} \vee z)$

a satisfying assignment for ϕ. We select the two edges in each variable gadget corresponding to the assignment of this variable. For each clause s, we select a representative literal with value **true**, and select the edge containing the corresponding terminal and the two vertices a_s, b_s. To this point, we have selected $2n + m$ edges which contains $2n + m$ terminals. There are still $4n - (2n + m) = 2n - m$ unmatched terminals. We can "absorb" those unmatched terminals to the $2n - m$ vertices $a_1, a_2, \dots, a_{2n-m}$ and the $2n - m$ vertices $b_1, b_2, \dots, b_{2n-m}$ constructed in the last step. After this, we have a perfect 3D-matching. This is why we use the number $2n - m$ in our construction.

Suppose ϕ is not satisfiable. We aim to show that G does not have a perfect 3D-matching. We prove the contra-positive. Suppose G contains a perfect 3D-matching. Then, exactly two internal edges must be selected in each variable gadget: we cannot select more than two of course; if we select less than two, the four internal vertices from A and B will not be fully matched. Moreover, in each gadget G_x, we must select either $\{e_1, e_3\}$ or $\{e_2, e_4\}$. This corresponds to an assignment for ϕ. Moreover, this assignment must be a satisfying assignment. Otherwise, by our construction, there exists a clause s such that a_s, b_s are unmatched. $\square$