# A Brief Introduction to Complexity Classes and NP-Completeness

MAD 3105: Discrete Mathematics II
Spring 2025

Adam Rumpf*

An important concept in theoretical Computer Science is the idea of *complexity classes*, which provide a precise way of categorizing problems according to the computational work required to solve them. Since our main textbook [3] does not include a thorough description of complexity classes, these supplementary notes will provide a brief description for use as a reference. These notes are based primarily on Cormen et al. [2].

As a caveat, note that the following descriptions omit many details and simplify many explanations. The precise definitions of concepts like decision problems, complexity classes, NP-completeness, and polynomial time reductions all involve concepts from theoretical Computer Science that go well beyond the scope of this course. Our primary focus will be on gaining an intuitive understanding of what NP-completeness means, and on understanding the structure of a typical NP-completeness proof.

## 1 Preliminaries

We will begin by establishing some of the basic definitions and concepts required to describe complexity classes.

### 1.1 Polynomial Time

The primary topic of interest in Complexity Theory is describing how computationally difficult a given problem is. We've seen through asymptotic analysis that there is a hierarchy of algorithmic run times ranging from slowly-increasing complexity (constant, logarithmic, linear, $n \lg n$, etc.) to quickly-increasing complexity (exponential, factorial, $n^n$, etc.). In Complexity Theory, a line of distinction is drawn between problems that can be solved in *polynomial time* and problems that cannot.

**Definition 1.1.** An algorithm is said to run in **polynomial time** if its worst-case time complexity is $T(n) = O(n^k)$ for some positive constant $k$. Here, $n$ is some measure of the "size" of the problem (length of a list, number of vertices in a graph, number of elements in a set, etc.).

Roughly speaking, we interpret a problem as being relatively "easy" if it is solvable in polynomial time, while it is relatively "hard" otherwise, in the sense that the problem is likely to become intractable as its size increases. While it is true that some polynomials (like $n^{100}$) grow very quickly, and while it is true that asymptotic notation could be hiding some very large constants,

*Florida Polytechnic University, Department of Applied Mathematics, arumpf@floridapoly.edu

in practice most polynomial time algorithms for common problems are of degree 4 or less, so for practical purposes polynomial time algorithms are substantially better than any exponential time algorithms for large problem instances.

When we refer to a "problem" in Complexity Theory, we are really referring to a general *type* of problem that can appear as many different specific instances. For example, finding the maximum matching in a bipartite graph is a type of problem, while finding a maximum matching in a particular graph $G = (V, E)$ is an instance of that problem. It is always true that certain specific instances of a problem may be easy to solve due to special structure, but determining complexity classes depends on the *general* version of the problem.

There are many different complexity classes, but for our purposes the most important include: P, NP, and NP-complete. We will define NP and NP-complete in Section 2, but for now we can define P as the set of problems solvable in polynomial time, which are generally thought of as the "easiest" types of problems we know about.

**Definition 1.2.** The complexity class P (for *Polynomial*) is the set of all problems that can be solved by a polynomial time algorithm. Given a problem $A$, showing that $A \in \mathrm{P}$ requires showing that any general instance of problem $A$ can be solved in polynomial time.

It should be mentioned that whether or not a problem belongs to class P is an intrinsic property of the problem, itself, independent of whether or not a polynomial time solution algorithm has been discovered by humans. Some problems currently thought not to belong to P might actually belong to P after all, but we cannot say for sure until a polynomial time algorithm is found (or until it is proved that one does not exist).

To define the other complexity classes, we need to restrict our scope to a special category of problem known as a *decision problem.*

**Definition 1.3.** A **decision problem** is one for which the only possible answers are "yes" or "no".

Some decision problems we've seen before include determining whether a given graph $G$ possesses various properties (e.g. whether $G$ is Eulerian, whether $G$ is bipartite, whether $G$ is Hamiltonian, whether $G$ is triangle-free, whether $G$ is connected, whether $G$ has a strong orientation, etc.). Many common optimization problems can also be phrased as analogous decision problems.

For example, the *maximum bipartite matching* problem asks us to find the maximum-cardinality matching in a given bipartite graph, while the *decision* version of the problem asks us whether there exists a bipartite matching of cardinality $k$ for some given $k$. The decision version of a problem is generally easier, and our definition for the NP-complete complexity class (in Section 2) will be restricted to only decision problems.

## 1.2 Polynomial Time Reduction

It is common throughout mathematics to solve one problem by translating it into an analogous instance of a different type of problem, thus enabling us to use a known solution method for the second problem to solve the first problem. For example, we've seen that the maximum bipartite matching problem on a graph $G$ can be solved by developing a related flow network $G'$ and finding a maximum flow in $G'$ that corresponds to a maximum matching in $G$ (see Figure 1.1).

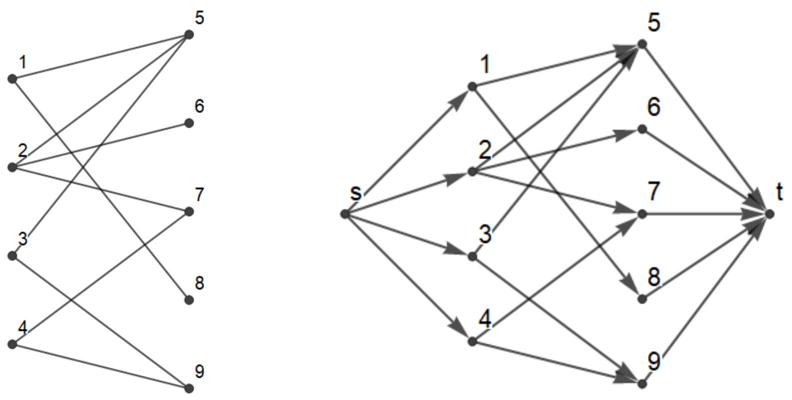

Figure 1.1: (Left) A graph $G$ for which we want to solve the maximum bipartite matching problem. (Right) A related flow network $G'$ whose maximum flow corresponds to the maximum bipartite matching of $G$.

Stated more precisely, consider the following two decision problems:

- **Maximum Bipartite Matching (Decision Version):** Given a bipartite graph $G$ and a number $k$, determine whether $G$ contains a matching of cardinality $k$ or greater.
- **Maximum Flow (Decision Version):** Given a flow network $G$ and a number $k$, determine whether $G$ has a feasible flow of value $k$ or greater.

The transformation discussed earlier in the course allows us to take any arbitrary instance of the first problem (maximum bipartite matching) and to answer it using a related instance of the second problem (maximum flow). The two instances correspond to each other in the sense that $G$ has a matching of cardinality $k$ or greater if and only if $G'$ has a flow of value $k$ or greater (i.e. both decision problems have the same answer, either both "yes" or both "no"). Moreover, the transformation from $G$ to $G'$ requires only a polynomial amount of work (orienting $|E(G)|$ edges, adding 2 vertices, adding $|V(G)|$ arcs, and defining $|V(G)| + |E(G)|$ capacities). Then this transformation is what we would refer to as a *polynomial time reduction* from the maximum bipartite matching problem to the maximum flow problem.

**Definition 1.4.** A decision problem $A$ is said to be **reducible in polynomial time** to another decision problem $B$, denoted $A \leq_P B$, if it is possible to translate an instance of $A$ in polynomial time to a corresponding instance of $B$ for which both decision problems have the same solution (i.e. the instance of $A$ is "yes" if and only if the corresponding instance of $B$ is "yes"). This transformation is called a **polynomial time reduction** of $A$ to $B$, or to $B$ from $A$.

The symbol $\leq_P$ is not really an inequality, since it would not make sense to make an inequality comparison between two problems. It can be read as "is reducible in polynomial time to". Note that this is a transitive relation: if problem $A$ reduces to problem $B$, and problem $B$ reduces to problem $C$, then problem $A$ reduces to problem $C$ (via $B$), i.e. $A \leq_P B$ and $B \leq_P C$ implies $A \leq_P C$.

Previously our reason for reducing the maximum bipartite matching problem to the maximum flow problem was because it allowed us to reuse a known efficient solution algorithm for finding maximum flows (the Ford-Fulkerson algorithm) to find maximum bipartite matchings. Since maximum flows can be found efficiently, this reduction immediately implies that maximum bipartite matchings can also be found efficiently. Importantly, the converse implication also holds.

**Observation 1.5.** If $A \leq_P B$, then $B$ is "at least as hard" as $A$. In particular:

(a) If $B \in \text{P}$, then $A \in \text{P}$. (That's because having a polynomial time algorithm for $B$ implies a polynomial time algorithm for $A$, consisting of reducing $A$ to $B$ in polynomial time and then solving $B$ in polynomial time.)

(b) If $A \notin \text{P}$, then $B \notin \text{P}$. (That's because, if there were a polynomial time algorithm for $B$, then by point (a) there would also be a polynomial time algorithm for $A$, which would contradict $A \notin \text{P}$.)

This observation also justifies the use of inequality-like notation ($\leq_P$) for denoting that one problem reduces to another, since the inequality indicates which problem's complexity class "bounds" the other.

## 2 NP-Completeness

Lots of problems have no known polynomial time solution algorithms, meaning that these problems cannot definitively be placed in the complexity class P. However, even if a decision problem cannot be *solved* in polynomial time, it is often easy to *verify* a solution that has already been provided. The decision problems that can be verified (but not necessarily solved) in polynomial time make up the complexity class NP.

**Definition 2.1.** The complexity class NP (for *Nondeterministic Polynomial*) is the set of all decision problems for which a "yes" instance can be verified in polynomial time given a proposed solution (or some other certificate). Saying that a problem "is NP" is shorthand for saying that it belongs to the complexity class NP.

For example, the problem of deciding whether a graph is Hamiltonian can be easily shown to be in NP. Given a graph $G$ that is indeed Hamiltonian, that fact can be verified in polynomial time by simply providing a Hamiltonian cycle and verifying that the cycle is valid (by checking that it is a simple cycle that includes every vertex).

It can be readily seen that P is a subset of NP.

**Observation 2.2.** Any decision problem in P is also in NP (i.e. $\text{P} \subseteq \text{NP}$).

This is because a "yes" instance of any decision problem in P can be verified in polynomial time without needing *any* solution certificate, since the problem can simply be solved from scratch in polynomial time.

One of the central open questions in modern Mathematics and theoretical Computer Science (and the subject of one of the Clay Mathematics Institute's Millennium Prize Problems[1]) is the *P vs NP Problem*, which concerns the relationship between problems that are easy to solve and problems that are easy to verify. It is an open question whether $\text{P} = \text{NP}$, though most Computer Scientists believe that $\text{P} \neq \text{NP}$. This would imply the existence of problems that are in NP but not in P, i.e. problems that are easy to verify but hard to solve. This brings us to the topic of the NP-complete complexity class.

[1] `https://www.claymath.org/millennium/p-vs-np/`

**Definition 2.3.** The complexity class NP-complete is the set of all decision problems $A$ such that:

(a) $A \in \text{NP}$ (i.e. $A$ is verifiable in polynomial time).

(b) *Any* other problem in NP is reducible in polynomial time to $A$ (i.e. $A$ is at least as hard as any other NP problem).

Saying that a problem "is NP-complete" is shorthand for saying that it belongs to the complexity class NP-complete.

Roughly speaking, this makes NP-complete the "hardest" set of problems in NP. If it turns out that P = NP, then it would also be the case that P = NP = NP-complete, but the far likelier scenario (and the one that most Computer Scientists regard to be true) is that P $\neq$ NP, in which case P and NP-complete are disjoint sets, with the NP-complete problems being problems that are verifiable but not solvable in polynomial time. For this reason, it is of practical importance to know that a given problem is NP-complete, since this shows that the problem cannot be solved in polynomial time (unless P = NP).

## 3 Proving NP-Completeness

It is not clear from the above definition how one would prove that a problem is NP-complete (or even that any such problems exist), but this result was demonstrated in the early 1970s by the Cook-Levin Theorem [1, 5].

**Theorem 3.1 (Cook-Levin).** The boolean satisfiability problem (SAT) is NP-complete.

More precisely, the theorem shows that the SAT problem (which will be explained in Section 3.2) is NP, and that *any* other problem in NP is reducible in polynomial time to SAT (i.e. that $A \leq_P \text{SAT}$ for any $A \in \text{NP}$). This result established the existence of NP-complete problems, since it was the first ever problem explicitly shown to be NP-complete, but it also provides a method for more easily "growing" a set of known NP-complete problems.

### 3.1 How to Prove a Problem is NP-Complete

The proof of the Cook-Levin Theorem is highly technical and well beyond the scope of our course, and in general it would be extremely difficult to prove that *any* problem is NP-complete directly from the definition. Fortunately, due to how the NP-complete complexity class is defined, once it is known that one problem is NP-complete it is much easier to prove that another is.

**Theorem 3.2.** For any NP-complete problem $A$ and any NP problem $B$, if $A$ is reducible to $B$ in polynomial time ($A \leq_P B$), then $B$ is NP-complete.

This follows because the NP-completeness of $A$ implies that any problem $C \in \text{NP}$ is reducible to $A$, and since $A$ is reducible to $B$, such $C$ is also reducible to $B$ (by transitivity). Since $B$ is NP, and since any $C \in \text{NP}$ is reducible to $B$, this shows that $B$ is by definition NP-complete.

With this in mind, the NP-completeness of one problem $A$ can be used to prove the NP-completeness of another problem $B$ by applying the following approach:

1. Prove that $B \in$ NP. This requires explaining how a "yes" instance of $B$ can be verified in polynomial time given a solution. (This step is usually fairly trivial and requires minimal explanation.)
2. Prove that some known NP-complete problem $A$ can be reduced in polynomial time to $B$. This requires explaining how to solve an arbitrary instance of $A$ using a black box solver for $B$ plus a polynomial number of extra steps.

   In practice, this generally consists of translating an instance of $A$ into an analogous instance of $B$ such that the two have equivalent decision results (both "yes" or both "no").

The Cook-Levin Theorem established that the SAT problem is NP-complete, and the above method was then applied to grow a "tree" of additional known NP-complete problems which could then be used to prove the NP-completeness of even more problems. The method was first used in a 1972 paper by Richard Karp [4], which established the NP-completeness of a set of 21 known hard problems from various branches of Discrete Mathematics, and are now known as *Karp's 21 NP-complete problems.* Some of these are listed in Section 3.2.

Another interesting implication of the definition of NP-completeness is that all NP-complete problems are reducible to each other in polynomial time, which makes them all in some sense "equally hard". It would also mean that finding a polynomial time algorithm for *one* NP-complete problem would provide a polynomial time algorithm for solving *all* NP-complete problems, and would thus establish that P = NP.

## 3.2 Some Common NP-Complete Problems

Many common problems from a diverse array of fields in Discrete Mathematics have been shown to be NP-complete, and today there are hundreds of known NP-complete problems. It is useful to have a library of some common NP-complete problems for use in proving that others are NP-complete. Some are listed below.

- **Boolean Satisfiability (SAT):** SAT was the first ever problem proved to be NP-complete (thus establishing the Cook-Levin Theorem). It concerns a *boolean formula* consisting of some number of boolean (0/1, or true/false) variables $x_1, \ldots, x_n$ combined using some number of boolean operations ($\vee$ OR, $\wedge$ AND, and $\neg$ NOT) and parentheses for grouping. For example, $\phi$ defined by

  $$\phi(x_1, x_2, x_3, x_4) = \big((x_1 \vee x_2) \vee \neg(\neg x_1 \wedge x_3) \vee x_4\big) \wedge \neg x_2$$

  is a boolean formula on 4 variables. A *satisfying assignment* for a formula is an assignment of truth values to all variables such that the overall expression evaluates to 1 (true). A formula is called *satisfiable* if it has a satisfying assignment. Given any boolean formula $\phi$, the SAT problem consists of deciding whether $\phi$ is satisfiable.

  For example, the formula $\phi$ defined above has a satisfying assignment $(0, 0, 1, 1)$ since

  $$\phi(0,0,1,1) = \big((0 \vee 0) \vee \neg(\neg 0 \wedge 1) \vee 1\big) \wedge \neg 0 = (0 \vee \neg 1 \vee 1) \wedge 1 = 1 \wedge 1 = 1$$

  but the formula $\psi$ defined by

  $$\psi(x_1, x_2) = \neg x_1 \wedge \neg x_2 \wedge (x_1 \vee x_2)$$

  has no satisfying assignment, evaluating to 0 no matter what $x_1$ and $x_2$ are.

- **Boolean 3-Conjunctive Normal Form Satisfiability (3-SAT):** 3-SAT is the special case of SAT for which the boolean formula $\phi$ consists of a conjunction (AND) of *clauses*, each of which consists of a disjunction (OR) of exactly 3 literals (variables or their negations). For example,

  $$\phi(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

  Here, the first clause is $x_1 \vee \neg x_1 \vee \neg x_2$, consisting of a disjunction of three literals: $x_1$, $\neg x_1$, and $\neg x_2$. A boolean formula with this structure is said to be in 3-conjunctive normal form (3-CNF). The 3-SAT problem consists of deciding whether a 3-CNF boolean formula is satisfiable.

  Given the restrictive structure of 3-SAT, it seems like the problem would be easier to solve than the more general problem SAT, but both are NP-complete, shown because any general boolean formula can be transformed into an equivalent 3-CNF formula. In practice, 3-SAT is an extremely important NP-complete problem because it is used in many other NP-completeness proofs.

- **Zero-One Integer Programming (ZOIP):** A common type of problem in mathematical optimization is the *linear program*, which consists of attempting to minimize a linear objective function subject to a set of linear inequality constraints. The constraint set for any linear program can be written in the form

  $$\mathbf{Ax} \leq \mathbf{b}$$

  for a given variable vector $\mathbf{x} \in \mathbb{R}^n$, requirement vector $\mathbf{b} \in \mathbb{R}^m$, and constraint matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, in which case the program has $n$ variables and $m$ constraints. A *zero-one integer program* is a linear program for which all variables are required to take binary values $\mathbf{x} \in \{0, 1\}^n$. The ZOIP problem consists of deciding whether a given zero-one integer program has any feasible solutions, i.e. whether there are any binary vectors that satisfy all $m$ inequality constraints.

- **Hamiltonian Cycle (HAM-CYCLE):** Given a graph $G = (V, E)$, the HAM-CYCLE problem consists of deciding whether $G$ contains a Hamiltonian cycle (a simple cycle that includes every vertex).

- **Clique:** Given a graph $G = (V, E)$ and a number $k$, the CLIQUE problem consists of deciding whether $G$ contains a clique of size $k$ or greater (see 3.3.1).

- **3-Colorability (3-COLOR):** Given a graph $G = (V, E)$, the 3-COLOR problem consists of deciding whether $G$ is 3-colorable.

- **Vertex Cover:** Given a graph $G = (V, E)$ and a number $k$, the VERTEX-COVER problem consists of deciding whether $G$ has a vertex cover (a set $Q \subseteq V$ of vertices such that each edge in $E$ has at least one endpoint in $Q$) of size $k$ or less (see 3.3.3).

- **Traveling Salesman Problem (TSP):** Given a weighted complete graph $G = (V, E)$ and a number $k$, TSP consists of deciding whether $G$ has a Hamiltonian cycle of weight $k$ or less.

- **Subset Sum:** Given a set $S$ of numbers and a target value $t$, the SUBSET-SUM problem consists of deciding whether there is a subset of $S$ whose sum is exactly $t$.

- **Set Cover:** Given a ground set $X$, a collection of subsets $S_1, \ldots, S_n$ of $X$, and a number $k$, the SET-COVER problem consists of deciding whether there is a collection of $k$ or fewer of the subsets $S_1, \ldots, S_n$ whose union is $X$.

## 3.3 Example NP-Completeness Proofs

The following examples illustrate the method for proving that a problem is NP-complete by applying the two-step method described in Section 3.1 (showing that the problem is in NP, then showing that a known NP-complete problem can be reduced to it).

### 3.3.1 Clique

We will show that CLIQUE is NP-complete by showing that it is NP, and that it is reducible in polynomial time to the NP-complete problem 3-SAT. For reference:

- **CLIQUE:** Given a graph $G = (V, E)$ and a number $k$, decide whether $G$ has a clique of size $k$ or greater.
- **3-SAT:** Given a boolean formula $\phi$ in 3-conjunctive normal form (a conjunction of clauses, each consisting of a disjunction of 3 literals), decide whether $\phi$ has a satisfying assignment.

*Proof.* We first show that CLIQUE $\in$ NP. Given an instance of CLIQUE with graph $G$ and number $k$, an affirmative result can be verified using the clique $S$, itself. It can be verified in polynomial time that $S$ has cardinality at least $k$ and that all of its vertices are pairwise adjacent.

We next show that 3-SAT $\leq_P$ CLIQUE. Consider an instance of 3-SAT with a boolean formula $\phi$ with $r$ clauses. We will define a corresponding instance of CLIQUE by constructing a graph $G$ based on $\phi$, and by searching for cliques of size at least $r$. Define a vertex in $G$ for each literal in $\phi$, and define an edge between vertices whose corresponding literals (a) belong to different clauses and (b) are not logical negations of each other (i.e. not between $x_i$ and $\neg x_i$). See Figure 3.1 for an example. Clearly such $G$ can be constructed in polynomial time.

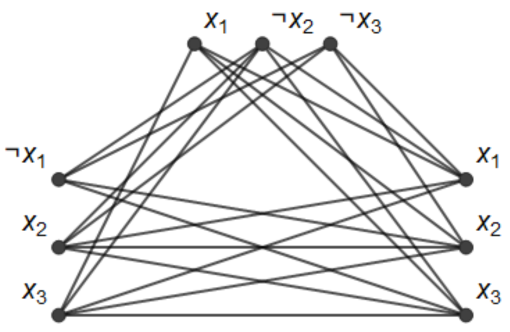

Figure 3.1: The graph $G$ corresponding to the boolean formula $\phi(x_1, x_2, x_3) = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$. An example satisfying argument is $(x_1, x_2, x_3) = (0, 0, 1)$, which corresponds to a clique of size 3 consisting of the $\neg x_1 = 1$ from the left set, the $\neg x_2 = 1$ from the top set, and the $x_3 = 1$ from the right set.

We claim that $\phi$ is satisfiable if and only if $G$ has a clique of size $r$ or greater. To show this, first suppose $\phi$ has a satisfying argument. This requires that at least one literal in each clause be assigned 1. Let $S$ be a set of vertices in $G$ containing one vertex from each clause corresponding to one of the literals from that clause assigned 1. All vertices in $S$ must be pairwise adjacent since all correspond to different clauses and none can correspond to a literal and its own negation (since we assume a satisfying argument), therefore $S$ is a clique of size $r$.

Next, suppose $G$ has a clique $S$ of size $r$. There are no edges between vertices corresponding to the same clause, so all $r$ vertices must correspond to different clauses. Then assigning a 1 to each literal in $S$ produces a satisfying argument of $\phi$ since it results in at least one 1 in each clause, and since the clique cannot contain any literal and its own negation.

This shows that 3-SAT is reducible in polynomial time to CLIQUE, and since 3-SAT is NP-complete, we conclude that CLIQUE is also NP-complete. □

### 3.3.2 Independent Set

We will show that INDEPENDENT-SET is NP-complete using a reduction from CLIQUE. For reference:

- **INDEPENDENT-SET:** Given a graph $G = (V, E)$ and a number $k$, decide whether $G$ has an independent set of size $k$ or greater.

*Proof.* We first show that INDEPENDENT-SET $\in$ NP. Given an instance of INDEPENDENT-SET with graph $G$ and number $k$, an affirmative result can be verified using the independent set $S$, itself. It can be verified in polynomial time that $S$ has cardinality at least $k$ and that none of its vertices are adjacent.

We next show that CLIQUE $\leq_P$ INDEPENDENT-SET. Consider an instance of CLIQUE with graph $G = (V, E)$ and number $k$. We will construct a corresponding instance of INDEPENDENT-SET using the complement $\overline{G}$ of $G$ and using the same number $k$. This complement can clearly be constructed in polynomial time.

A set $S \subseteq V$ is a clique in $G$ if and only if it is an independent set in $\overline{G}$ (since pairwise adjacent vertices in $G$ become pairwise nonadjacent in $\overline{G}$), therefore $\overline{G}$ has an independent set of size $k$ if and only if $G$ has a clique of size $k$. Then CLIQUE is reducible in polynomial time to INDEPENDENT-SET, and since CLIQUE is NP-complete, we conclude that INDEPENDENT-SET is also NP-complete. □

### 3.3.3 Vertex Cover

We will show that VERTEX-COVER is NP-complete using a reduction from CLIQUE. For reference:

- **VERTEX-COVER:** Given a graph $G = (V, E)$ and a number $k$, decide whether $G$ has a vertex cover (a set $Q \subseteq V$ of vertices such that each edge in $E$ has at least one endpoint in $Q$) of size $k$ or greater.

*Proof.* We first show that VERTEX-COVER $\in$ NP. Given an instance of VERTEX-COVER with graph $G$ and number $k$, an affirmative result can be verified in polynomial time using the vertex cover $Q$, itself. It can be verified in polynomial time that $Q$ has cardinality at most $k$ and that every edge in $E$ has at least one endpoint in $Q$.

We next show that CLIQUE $\leq_P$ VERTEX-COVER. Given an instance of CLIQUE with graph $G = (V, E)$ and number $k$, construct a corresponding instance of VERTEX-COVER on the complement $\overline{G}$ of $G$ with number $|V| - k$. The complement can clearly be constructed in polynomial time. We claim that $G$ contains a clique of size $k$ or more if and only if $\overline{G}$ contains a vertex cover of size $|V| - k$ or less.

Suppose $G$ contains a clique $S$ of size $|S| = k$. We claim that $V \setminus S$ is a vertex cover of $\overline{G}$. To show this, for any edge $\{u, v\} \in E(\overline{G})$, it must be the case that $u$ and $v$ are not adjacent in $G$, and thus at most one out of $u$ and $v$ is in the clique $S$. This implies that at least one out of $u$ and $v$ is in $V \setminus S$, and thus $V \setminus S$ is a vertex cover of $\overline{G}$ of size $|V| - |S| = |V| - k$.

Next, suppose $\overline{G}$ contains a vertex cover $T$ of size $|T| = |V| - k$. We claim that $V \setminus T$ is a clique in $G$. To show this, for any edge $\{u, v\} \in E(\overline{G})$, it must be the case that at least one out of $u$ and $v$ is in $T$. This is equivalent to saying that, for any $u, v \notin T$, there cannot be an edge $\{u, v\}$ in $E(\overline{G})$, in which case $\{u, v\}$ is an edge in $E(G)$. This means that all elements of $V \setminus T$ are pairwise adjacent in $G$, making $V \setminus T$ a clique in $G$ of size $|V| - |T| = |V| - (|V| - k) = k$.

This shows that CLIQUE is reducible in polynomial time to VERTEX-COVER, and since CLIQUE is NP-complete, we conclude that VERTEX-COVER is also NP-complete. □

### 3.3.4 4-Colorability

We will show that 4-COLOR is NP-complete using a reduction from 3-COLOR.

*Proof.* We first show that 4-COLOR $\in$ NP, which is straightforward, since a 4-coloring can be verified to be a proper coloring in polynomial time by checking the colors of both endpoints of each edge.

We next show that 3-COLOR $\leq_P$ 4-COLOR. Consider an instance of 3-COLOR with a graph $G$. Define a new graph $G'$ by starting with $G$ and adding one vertex $v$ adjacent to all other $u \in V(G)$. Such $G'$ can clearly be constructed in polynomial time.

Because $v$ is adjacent to all other vertices in $G'$, any proper coloring of $G'$ must assign a unique color to $v$. Then if $G'$ is 4-colorable, one color must be used for $v$ with the remaining 3 colors being used for $V(G)$, which implies that $G$ is 3-colorable. On the other hand, if $G$ is 3-colorable then $G'$ is certainly 4-colorable, since 3 colors can be used for $V(G)$ leaving a fourth color for $v$. We conclude that $G'$ is 4-colorable if and only if $G$ is 3-colorable.

Then 3-COLOR is reducible in polynomial time to 4-COLOR, and since 3-COLOR is NP-complete, we conclude that 4-COLOR is also NP-complete. □

This argument can be extended in a straightforward way (e.g. using induction) to show that it is NP-complete to decide whether a graph is $k$-colorable for any $k \geq 3$.

### 3.3.5 Hamiltonian Path

We will show that HAM-PATH NP-complete using a reduction from HAM-CYCLE. For reference:

- **HAM-PATH:** Given a graph $G = (V, E)$, decide whether $G$ has a Hamiltonian path (a simple path that includes every vertex in $V$).
- **HAM-CYCLE:** Given a graph $G = (V, E)$, decide whether $G$ has a Hamiltonian cycle (a simple cycle that includes every vertex in $V$).

*Proof.* First, we show that HAM-PATH is NP. Given a graph $G$ that contains a Hamiltonian path, the result can be verified in polynomial time by simply verifying that the path is simple and that it includes every vertex.

We next show that HAM-CYCLE $\leq_P$ HAM-PATH. Consider an instance of HAM-CYCLE on a graph $G$. Define a new graph $G'$ from $G$ by choosing any vertex $v \in V(G)$ and dividing it into two vertices $v$ and $v'$, both with neighbor set $N_G(v)$. Then add two more vertices $s$, adjacent to only $v$, and $t$, adjacent to only $v'$. Such $G'$ can clearly be constructed in polynomial time. We claim that $G$ contains a Hamiltonian cycle if and only if $G'$ contains a Hamiltonian path.

Suppose $G$ contains a Hamiltonian cycle $C$. Then $G'$ contains a Hamiltonian path, consisting of the path from $s$ to $v$, around the part of $C$ that includes every vertex in $V(G)$ except $v$, then to $v'$ (which must be possible since $v'$ has the same neighbors in $G$ as $v$ has), and finally to $t$. This is a valid Hamiltonian path since every vertex in $V(G')$ is used exactly once.

Next suppose $G'$ contains a Hamiltonian path $P$ from $s$ to $t$. Then $G$ contains a Hamiltonian cycle, consisting of the part of $P$ beginning at $v$ and ending at the predecessor of $v'$, and then from that predecessor to $v$, thus closing the cycle. This is a valid Hamiltonian cycle since every vertex in $V(G)$ (which does not include $v'$, $s$, or $t$) is used exactly once.

We conclude that $G$ contains a Hamiltonian cycle if and only if $G'$ contains a Hamiltonian path. Then HAM-CYCLE is reducible in polynomial time to HAM-PATH, and since HAM-CYCLE is NP-complete, we conclude that HAM-PATH is also NP-complete. □

### 3.3.6 Spanning Tree with Constrained Leaves

Consider a decision problem called TREE-LEAVES for which we are given a graph $G = (V, E)$ and a subset $S \subseteq V$ of vertices, and asked to determine whether $G$ has a spanning tree whose leaves are exactly the set $S$. We can show that this is NP-complete using a reduction from HAM-PATH.

*Proof.* We first show that TREE-LEAVES $\in$ NP. Given an instance of TREE-LEAVES with graph $G$ and set $S$, an affirmative result can be verified using the spanning tree $T$, itself. It can be verified in polynomial time that $T$ is a spanning tree of $S$ and that its leaves (its degree-1 vertices) exactly correspond to the set $S$.

We next show that HAM-PATH $\leq_P$ TREE-LEAVES. Consider an instance of HAM-PATH with graph $G = (V, E)$, and note that a Hamiltonian path is equivalent to a spanning tree with exactly two leaves. Therefore $G$ has a Hamiltonian path if and only if there is a pair of vertices $u, v \in V$ such that $G$ has a spanning tree whose leaves are $S = \{u, v\}$. There are $\binom{|V|}{2} = \frac{1}{2}|V|^2 + \frac{1}{2}|V|$ pairs of vertices in $G$ that could serve as the endpoints of a potential Hamiltonian path. In the worst case all of them would need to be checked, but the number of pairs to check is only polynomial in $|V|$.

We will define a solution algorithm for HAM-PATH as follows: Loop through every one of the $\binom{|V|}{2}$ pairs of vertices $u, v \in V$. For each pair, construct an instance of TREE-LEAVES with graph $G$ and leaf set $S = \{u, v\}$, and solve the TREE-LEAVES problem. If there is a spanning tree of $G$ with leaves $\{u, v\}$, then there is a Hamiltonian path in $G$. Otherwise, if all pairs $u, v \in V$ are exhausted and none yielded a spanning tree, then $G$ has no Hamiltonian path.

This shows that HAM-PATH is reducible in polynomial time to a polynomial number of TREE-LEAVES instances, and since HAM-PATH is NP-complete, we conclude that TREE-LEAVES is also NP-complete. □

Note that the structure of this proof is slightly different than some of the previous proofs, since the HAM-PATH problem was not reducible to just a *single* instance of TREE-LEAVES, but rather a collection of different TREE-LEAVES instances with different sets $S$. However, as long as the number of these instances is polynomial, we still have a polynomial time reduction from HAM-PATH to TREE-LEAVES and the conclusions still hold. The logic goes that being able to solve a polynomial number of TREE-LEAVES instances, each in polynomial time, would imply that HAM-PATH is solvable in polynomial time, which would contradict HAM-PATH being NP-complete (unless P = NP).

## References

[1] S. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971. `doi: 10.1145/800157.805047`.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, London, England, 3rd edition, 2009. ISBN 978-0-262-03384-8.

[3] R. Johnsonbaugh. *Discrete Mathematics*. Pearson, 8th edition, 2023. ISBN 9780137848577. URL `https://www.pearson.com/en-us/subject-catalog/p/discrete-mathematics/P200000006219`.

[4] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series. Springer, Boston, MA, 1972. `doi: 10.1007/978-1-4684-2001-2_9`.

[5] L. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3): 265–266, 1973.