A Comprehensive Guide to the Classical NP-Complete Problems

Nargiz Aghayeva

ADA University, School of IT and Engineering, Baku, Azerbaijan,
naghayeva16042@ada.edu.az

Abstract

NP-completeness lies at the core of theoretical computer science, shaping our understanding of computational hardness, algorithm design, and complexity theory. This paper presents a comprehensive and structured guide to classical NP-complete problems, designed for both students and educators. It begins with a formal introduction to the theory of NP-completeness and proceeds to detailed reductions for fifteen fundamental NP-complete problems, each accompanied by illustrative examples and correctness proofs. Additionally, the practical relevance of NP-complete problems is explored through their applications in real-world contexts, bridging the gap between theory and practice. This resource is particularly suited for learners, instructors, and anyone interested in deepening their understanding of the landscape of NP-complete problems.

Keywords

NP-completeness, Karp's 21 problems, Polynomial-time reductions, Computational complexity, 3SAT, Vertex Cover, CLIQUE, Hamiltonian Path, Chromatic Number

1 Introduction

The theoretical foundations of computer science were laid by Alan Turing, one of the most prominent mathematicians of the twentieth century. In 1936, he published his groundbreaking paper titled *"On Computable Numbers, with an Application to the Entscheidungsproblem"*, in which he introduced the concept of a *Turing Machine*, an abstract computational model capable of simulating any algorithmic process (Turing et al., 1936). This work laid the foundation for the modern theory of computation and formalized the notion of *computability*, giving rise to what we now understand as algorithms or computers.

Building on this foundation, complexity theory emerged as a branch of theoretical computer science that studies the computational resources, specifically, time and space, required to solve computational problems (Sipser, 2012). The primary goal of this field is to classify problems into complexity classes based on how efficiently they can be solved.

The most fundamental of these classes is **P** (polynomial time), which consists of all problems that can be solved by a deterministic algorithm in polynomial time. In contrast, the class **NP** (nondeterministic polynomial time) contains problems for which a given solution can be verified in polynomial time. While it is clear that every problem in P is also in NP, it remains unknown whether every problem in NP can also be solved in polynomial time; this is the core of the famous P vs NP (Fortnow, 2009).

Within NP, there exists a particularly significant subclass known as **NP-complete** problems. These problems are important because solving any one of them in polynomial time would imply that all NP problems can be solved in polynomial time. The concept of NP-completeness was first introduced by Stephen Cook in 1971, who established the Boolean Satisfiability Problem (*SAT*) as the first known NP-complete problem (Cook, 1971). This result is now formally known as the *Cook-Levin Theorem*. In particular, Cook's contribution was not only in proving *SAT*'s NP-completeness but also in raising the P vs NP question, a problem that has remained one of the most important open questions in computer science.

Shortly after, Richard Karp significantly expanded the scope of NP-completeness by identifying 21 classic combinatorial problems as NP-complete in his seminal 1972 paper, *"Reducibility Among Combinatorial Problems"* (Karp, 1972).

Since then, countless researchers have worked to determine whether problems in NP truly require super-polynomial time. While no polynomial-time algorithm has been found for any NP-complete problem, polynomial-time reductions between them have become a central tool for demonstrating NP-completeness.

Given the central role of reductions in demonstrating NP-completeness, understanding how to construct such reductions is essential for anyone studying computational complexity. In this paper we focused on clarifying various NP-complete problems using formal polynomial-time reductions between them.

While these topics may initially appear complex, with time and exploration, their underlying structure and coherence become increasingly apparent. The reductions, in particular, serve not only as powerful computational tools, but also uncover elegant and often unexpected relationships between problems, highlighting the depth and intellectual beauty of theoretical computer science.

2 Background

Modern computer science classifies problems based on their computational properties. In particular, a central concept is the **decision problem**, which asks whether a given input meets certain conditions, with the output being either *Yes* or *No* (Garey and Johnson, 1979). These problems form the basis of many important complexity classes, including **P**, **NP**, and **NP**-complete.

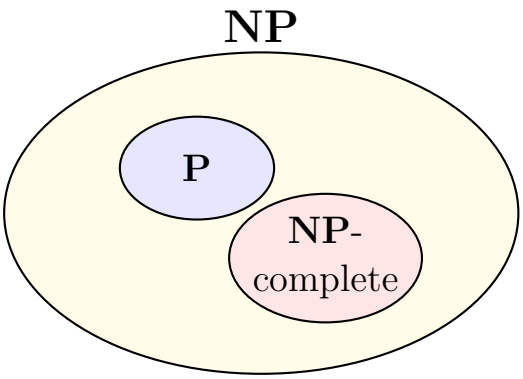
Decision problems are often defined in formal terms as *languages* over a finite alphabet Σ (Sipser, 2012). The language $L \subseteq \Sigma^*$ is constructed from all input strings for which the response to the decision problem is "yes." As a result, the terms "language" and "decision problem" are often used interchangeably in complexity theory (Papadimitriou, 2003).

Definition 2.1. *The class* **P** *contains all languages L for which there exists a determin-*

istic Turing machine M that decides membership of any input $x \in \Sigma^*$ in time polynomial in the length of x.

Definition 2.2. *The class **NP** contains decision problems for which a proposed certificate can be verified in polynomial time on a deterministic Turing machine.*

In this context, a *certificate* refers to a proposed solution, and the algorithm that checks the correctness of the solution is called a *verifier* (Sipser, 2012). The key insight for the class NP is that finding the solution to these problems might be hard, but once we have a solution, the verification takes polynomial time.



A common point of confusion is the relationship between the classes **P** and **NP** when people first learn about these concepts. Remember that the class P is a subset of NP, meaning all problems in P are also in NP. This does not mean that problems in NP do not have polynomial-time solutions. Some problems in NP, like sorting, are already known to be solvable in polynomial time; they belong to P and therefore also to NP. The big question is whether every problem whose solutions can be verified quickly also has a quick way to find those solutions. For many important NP problems, no polynomial-time algorithm is known so far, but no one has proven that such an algorithm cannot exist. So, in short, it is certain that $\mathbf{P} \subseteq \mathbf{NP}$, but whether $\mathbf{P} = \mathbf{NP}$ remains an open question (Fortnow, 2009).

Definition 2.3. *A language B is* NP-complete *if it satisfies the following two properties:*

1. *$B \in NP$, and*
2. *for every language $A \in NP$, $A \leq_P B$.*

It is also important to distinguish between NP-complete and NP-hard problems. While all NP-complete problems are NP-hard, the converse is not necessarily true. NP-complete problems are the hardest problems in the NP class for which polynomial-time solutions are currently not known. A problem is called NP-hard if every problem in NP can be reduced to it in polynomial time. However, an NP-hard problem does *not* have to be in the class NP itself (Welch, 1982). For example, the *Halting problem* is not even decidable, so it does not belong to any complexity class, nor does it belong to NP. But it is as hard as NP-complete problems.

We have been talking about reduction, but we have not yet formally defined what it is and how it is used.

Definition 2.4. *A polynomial-time reduction transforms one problem into another such that a solution to the second yields a solution to the first, in polynomial time.*

Polynomial-time reducibility involves two problems, say A and B. If we can transform instances of A into instances of B in polynomial time such that solving B also solves A,

then we say that A reduces to B (denoted $A \leq_P B$) (Sipser, 2012). To illustrate the idea of reduction in a real-life context, consider the following analogy.

Problem A: Reaching Barcelona from Baku.

Problem B: Purchasing a valid plane ticket for the journey.

Clearly, if you can solve Problem B, then solving Problem A becomes easy; that is, once you have the ticket, the journey is just a matter of execution. Therefore, we can say that *Problem A reduces to Problem B*. This mirrors the concept of polynomial-time reductions in complexity theory. In complexity theory, reductions are the primary tool used to demonstrate that a new problem is at least as hard as known NP-complete problems. If a known NP-complete problem can be reduced to a new problem in polynomial time, this establishes the new problem as NP-complete.

NP-complete problems often appear in two versions: the decision problem and the optimization problem. For instance, the Vertex Cover decision problem asks whether there exists a vertex cover of size at most k, while the optimization version seeks the smallest such cover. Since NP-completeness is formally defined in terms of decision problems, our focus will remain on these.

With these foundations in place, we now turn to a series of classical NP-complete problems. In the following sections, we formally define each problem, explain its significance, and demonstrate its NP-completeness via polynomial-time reductions, thereby highlighting the structural depth of the NP-complete class.

3 Classical NP-Complete Problems

3.1 ***3SAT***

The satisfiability problem, or SAT for short, was the first NP-complete problem established by Stephen Cook in 1971. The special form of the SAT problem, called *3SAT,* is one of the most foundational NP-complete problems, which is used to prove the NP-completeness of many well-known problems.

Definition 3.1. *Given a Boolean formula ϕ in conjunctive normal form such that each clause contains exactly three literals, the 3SAT decision problem asks whether there exists a certain assignment of 0 or 1 to the variables that makes the formula ϕ true.*

Theorem 3.1. *3SAT decision problem is NP-Complete.*

Proof:

The problem is, in fact, in class NP. Given the proposed 3CNF formula (the certificate), we can verify in polynomial time whether the formula is satisfiable. The NP-completeness of the problem is proved by Karp via giving the reduction from the *SAT* problem. The reduction was established by Karp as follows:

Replace a clause $\sigma_1 \cup \sigma_2 \cup \ldots \cup \sigma_m$, where the σ_i are literals and $m > 3$, by

$$(\sigma_1 \cup \sigma_2 \cup u_1)(\sigma_3 \cup \ldots \cup \sigma_m \cup \neg u_1)(\neg\sigma_3 \cup u_1) \ldots (\neg\sigma_m \cup u_1),$$

where u_1 is a new variable. Repeat this transformation until no clause has more than three literals.

Later, this reduction was also presented and formalized in the book *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Garey and Johnson, 1979).

Construction of the reduction:

Suppose you are given a Boolean formula $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ in conjunctive normal form, where each clause $C_i = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_m$ is a disjunction of literals with $m \geq 1$.

1. *Case 1:* $m = 1$

   If the clause has 1 literal, say ℓ, then:

   - add 2 new dummy variables that do not exist in the original formula, say x and y.
   - construct 4 new clauses by combining ℓ with all possible combinations of x, y, and their negations.
   - replace the original clause with the conjunction of the 4 new clauses in the formula.

2. *Case 2:* $m = 2$

   If the clause has 2 literals, say $(\ell_1 \vee \ell_2)$, then:

   - add one new dummy variable that does not exist in the original formula, say z.
   - construct 2 new clauses by combining $(\ell_1 \vee \ell_2)$ with z and $\neg z$: $(\ell_1 \vee \ell_2 \vee z)$ and $(\ell_1 \vee \ell_2 \vee \neg z)$.
   - replace the original clause with the conjunction of the 2 new clauses in the formula.

3. *Case 3:* $m = 3$

   - If the clause has 3 literals, it remains unchanged.

4. *Case 4:* $m > 3$

   Suppose we are given a clause $C = (l_1 \vee l_2 \vee l_3 \vee \cdots \vee l_m)$, where $m > 3$. We will construct $m - 2$ new clauses as follows:

   - add $m - 3$ new dummy variables that do not appear in the original formula.
   - the first clause will have the form: $(l_1 \vee l_2 \vee y_1)$
   - for the clauses $i = 2, \ldots, m-2$, the clauses will have the form: $(\neg y_{i-1} \vee l_{i+1} \vee y_i)$
   - the final clause will have the form: $(\neg y_{m-3} \vee l_{m-1} \vee l_m)$
   - replace the original clause with the conjunction of the new clauses in the formula.

Correctness of the reduction:

$\Rightarrow$ If ϕ is satisfiable, then ϕ' is satisfiable

Assume we are given a satisfying assignment A for the original CNF formula ϕ. We construct a satisfying assignment for ϕ' by extending A to the dummy variables used in the transformation. For **Case 1**, where the clause has 1 literal, (ℓ), we replace it with the 4 clauses: $(\ell \vee x \vee y)$, $(\ell \vee x \vee \neg y)$, $(\ell \vee \neg x \vee y)$, $(\ell \vee \neg x \vee \neg y)$. If ℓ is true, then no matter which value is assigned to x and y, all 4 clauses will be true. For the same reason, **Case 2** is also correct. **Case 3** is trivial. **Case 4:** If the original clause has more than 3 literals, it is replaced with a chain of clauses using new dummy variables. If any original literal is true, then we can choose values for the dummy variables to satisfy the whole chain. If all literals are false, then no assignment to the dummy variables can satisfy all clauses. Hence, satisfiability is preserved.

$\Leftarrow$ If ϕ' is satisfiable, then ϕ is satisfiable

Assume ϕ' is satisfiable under some assignment A' (which includes dummy variables). We restrict A' to the original variables to construct an assignment that satisfies ϕ. For **Case 1**, the 4 clauses can all be satisfied only if ℓ is true, since no assignment to x and y can satisfy all 4 when ℓ is false. So ℓ must be true, and the original clause is satisfied. For the same reason, **Case 2** is also correct: both clauses can only be satisfied if at least one of ℓ_1 or ℓ_2 is true. **Case 3** is trivial. For **Case 4**, suppose all l_i's are false. Then the first clause requires $y_1 = \text{true}$, the second requires $y_2 = \text{true}$, and so on. The final clause becomes $(\neg y_{m-3} \vee \text{false} \vee \text{false}) = \neg y_{m-3}$. So for the final clause to be true, $\neg y_{m-3}$ must be true, which contradicts the fact that we assigned a true value to y_{m-3} earlier. Therefore, at least one l_i must be true under A', and the original clause is satisfied.

Conclusion

$$\boxed{SAT \text{ is satisfiable } \iff 3SAT \text{ is satisfiable}}$$

Example 3.1.

We start with a satisfiable SAT formula that contains clauses with 1, 2, 3, and 4 literals:

$$\phi = (a) \wedge (b \vee c) \wedge (\neg a \vee d \vee \neg e) \wedge (f \vee g \vee h \vee i)$$

Transformation is applied as follows:

- **1-literal clause:** (a) is converted using new variables z_1, z_2:

$$(a \vee z_1 \vee z_2) \wedge (a \vee z_1 \vee \neg z_2) \wedge (a \vee \neg z_1 \vee z_2) \wedge (a \vee \neg z_1 \vee \neg z_2)$$

- **2-literal clause:** $(b \vee c)$ is converted using new variable z_3:

$$(b \vee c \vee z_3) \wedge (b \vee c \vee \neg z_3)$$

- **3-literal clause:** $(\neg a \vee d \vee \neg e)$ is already in 3-CNF, so we keep it as is:

$$(\neg a \vee d \vee \neg e)$$

- **4-literal clause:** $(f \vee g \vee h \vee i)$ is converted using a new variable z_4:

$$(f \vee g \vee z_4) \wedge (\neg z_4 \vee h \vee i)$$

The final 3SAT formula equivalent to ϕ is:

$$\begin{aligned}\phi' = &(a \vee z_1 \vee z_2) \wedge (a \vee z_1 \vee \neg z_2) \wedge (a \vee \neg z_1 \vee z_2) \\ &\wedge (a \vee \neg z_1 \vee \neg z_2) \wedge (b \vee c \vee z_3) \wedge (b \vee c \vee \neg z_3) \\ &\wedge (\neg a \vee d \vee \neg e) \wedge (f \vee g \vee z_4) \wedge (\neg z_4 \vee h \vee i)\end{aligned}$$

3.2 ***CLIQUE***

The *CLIQUE* problem is one of the most engaging and well-studied problems in graph theory and combinatorial mathematics. The problem is NP-complete, and it was among the 21 problems first established by Richard M. Karp in 1972 in his seminal paper.

Definition 3.2. *Given an undirected graph $G = (V, E)$ and an integer $k \leq |V|$, the CLIQUE problem asks whether the graph G contains a subset of k vertices such that every pair of vertices in the subset is connected by an edge.*

Theorem 3.2. *CLIQUE decision problem is NP-complete.*

Proof:

The *CLIQUE* problem is in class NP, because given an undirected graph G and an integer k (the certificate), we can verify whether it contains a clique of size k in polynomial time. To show that *CLIQUE* is NP-complete, we represent a reduction through *3SAT*.

The key insight into the reduction is that given any 3-CNF formula, we can build a graph such that the formula is satisfiable *if and only if* the graph has a k-clique, where k is the number of clauses in the formula.

Intuitive Analogy:
Imagine that you are forming a team for your senior year design project of k people by choosing each person from *different majors*, but you can only include people who *get along with each other* for productive teamwork. Imagine x_i and $\neg x_i$ as people who do not get along. In the graph, each person is represented by a vertex, and an edge between two people indicates that they get along.

- A k-clique in this graph represents a group of k people, each of whom comes from a different major. If it is a clique, then all pairs of vertices must be connected, meaning that everyone gets along.
- This corresponds exactly to selecting one literal from each clause in the 3-CNF formula such that no two selected literals contradict each other.

In logical terms, we have successfully chosen one literal from each clause. None of these literals contradict one another; we did not pick both x and $\neg x$. Therefore, they can all be made true at the same time, forming a satisfying assignment for the formula.

Now it is time to give the formal reduction to show the NP-Completeness of *CLIQUE*.

Construction of the reduction:

Let ϕ be a Boolean formula in 3-CNF form with k clauses, $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$, where each clause C_i contains exactly three literals.

- For each clause C_i, create a triangle formed by three vertices, each representing one literal in that clause. The total number of vertices is $3k$, with 3 vertices per clause.
- Add an edge between two vertices if allowed. You cannot connect the vertices:
  - within the same clause
  - contradictory literals as x and $\neg x$

Correctness of the reduction:

Let ϕ be a Boolean formula in 3-CNF form with k clauses, and let G be the graph constructed from ϕ as described above.

$\Rightarrow$ If ϕ is satisfiable, then G has a k-clique.

Suppose ϕ is satisfiable. Then there exists a certain truth assignment such that each clause C_i has at least one literal that is true under this assignment. For each clause C_i, choose one such satisfied literal ℓ_i, and let v_i be the vertex in G corresponding to ℓ_i.

We now show that the set $\{v_1, v_2, \ldots, v_k\}$ forms a clique in G. First, since we selected exactly one literal from each clause, the vertices $v_1, v_2, \ldots, v_k$ are from distinct clauses. Second, since the assignment satisfies all ℓ_i, and no variable is assigned both true and false, it follows that no two literals ℓ_i and ℓ_j are contradictory. Therefore, the pair (v_i, v_j) is connected by an edge for all $i \neq j$. Thus, the selected vertices form a k-clique in G.

$\Leftarrow$ If G has a k-clique, then ϕ is satisfiable.

Now, assume G has a clique of size k. Since vertices within the same clause are not connected in G, any k-clique must contain exactly one vertex from each clause. Let $\{v_1, v_2, \ldots, v_k\}$ be such a clique, and let ℓ_i be the literal corresponding to vertex v_i.

We now construct a truth assignment that satisfies ϕ. For each ℓ_i, assign the corresponding variable in a way that makes ℓ_i true. Since the literals in the clique are all pairwise connected, and contradictory literals are not connected in G, this assignment is consistent.

Each clause C_i contains one selected literal ℓ_i that is satisfied by the assignment, so all clauses are satisfied. Hence, ϕ is satisfiable.

Conclusion

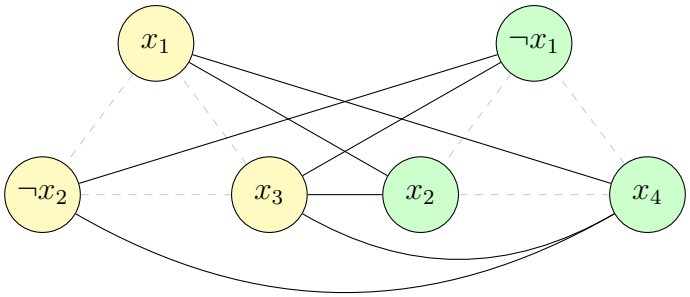
$$\boxed{\phi \text{ is satisfiable } \iff G \text{ has a } k\text{-clique.}}$$

Example 3.2. *Reduction from 3SAT to CLIQUE*

Let us now give an example of this construction with a 2-clause 3-CNF formula as follows:

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

The resulting graph is illustrated below:



Note that in the example above, I connected vertices within a clause with gray dashed lines just to illustrate the idea of a *triangle*. However, in the actual construction, *do not* connect vertices that belong to the same clause.

3.3 ***INDEPENDENT SET***

Now we will explore a similar problem to *CLIQUE*, which is the *Independent Set* problem. These two problems are computationally equivalent under graph complementation.

Definition 3.3. *Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, the Independent Set decision problem asks whether there exists a subset $S \subseteq V$ such that $|S| \geq k$ and $\forall u, v \in S,\ (u, v) \notin E$. In other words, S is an independent set of size at least k.*

Theorem 3.3. *The Independent Set decision problem is NP-Complete.*

Proof:

Given a graph $G = (V, E)$ and an integer k, a certificate is a subset $S \subseteq V$ with $|S| \geq k$. We can verify in polynomial time that no pair of two vertices in S are connected, meaning, $\forall u, v \in S,\ (u, v) \notin E$. Therefore, the problem is in NP. To prove that the *Independent Set* problem is NP-Complete, we provide a reduction from the *CLIQUE* problem.

Construction of the reduction:

Given an undirected graph $G = (V, E)$ and an integer k:

1. Initialize G' with the same vertex set V of G
2. For each pair of vertices (u, v) in graph G' :
    - Add edge between the vertices u and v only if (u, v) is not an edge in G. Otherwise, do not add the edge.
3. Output the instance (G', k).

Correctness of the reduction:

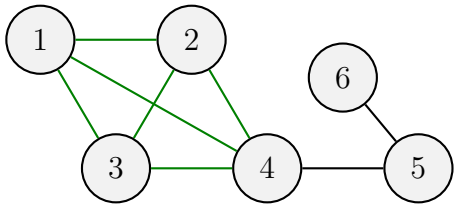
$\Rightarrow$ Suppose G contains a clique C of size k. By definition, every pair of vertices in C is connected in G. Therefore, in the complement graph G', no two vertices in C are connected. Hence, C forms an independent set of size k in G'.

$\Leftarrow$ Suppose G' contains an independent set S of size k. By definition, no two vertices in S are connected in G'. This implies that in G, every pair of vertices in S is connected. Therefore, S forms a clique of size k in G.

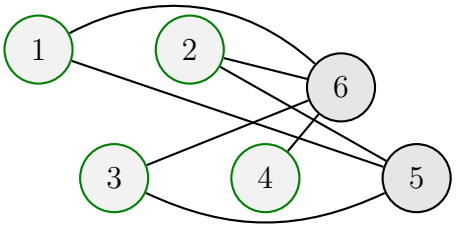
Conclusion

$$\boxed{G' \text{ has an independent set of size } k \iff G \text{ has a } k\text{-clique.}}$$

Example 3.3. *Reduction from CLIQUE to Independent Set*



Graph G with a clique of size $k = 4$.



Graph G' with an independent set of size $k = 4$.

3.4 ***HAMPATH***

Hamiltonian Path problem, also known as *HAMPATH*, is one of the most studied problems among NP-complete problems.

Definition 3.4. *Given a directed graph $G = (V, E)$, and two vertices $s, t \in V$, the HAMPATH Problem asks whether there exists a path from s to t that visits every vertex in V exactly once.*

Theorem 3.4. *The HAMPATH decision problem is NP-complete.*

Proof:

The *HAMPATH* problem is in class NP, because given a directed graph and a path from s to t (the certificate), we can verify the solution in polynomial time. To show that *HAMPATH* is NP-complete, we will provide a constructive proof by reducing *3SAT* to it. Note that the overall idea of reduction is based on the book *Introduction to the Theory of Computation by Michael Sipser* (Sipser, 2012). Before diving into the formal construction, let us first build an intuitive understanding of the connection between these two problems so you can truly appreciate the elegance behind it.

Intuitive Analogy:

Imagine you are planning a trip and need to visit a bunch of cities. Each *city* you visit requires making a key decision. At a certain point, you must choose between two roads: right and left. At every *decision gate*, you pick one road and commit to it. The left road leads you down one path; the right road takes you in a different direction. Once you have chosen, you can not go back. Along the way, there are *checkpoints* that allow passage only if certain roads have been taken earlier. These checkpoints are designed in such a way

that, if you have made at least one of the right decisions before arriving, the gate opens and you can pass through. Your goal is to find a route that passes through every city and checkpoint *exactly once*, without turning back or skipping any part of the journey. If such a route exists, then your journey is considered successful.

Now, let us give the formal reduction to show the NP-completeness of *HAMPATH*.

Construction of the reduction:

Suppose we have a *3SAT* formula ϕ with variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$. Our goal is to construct a directed graph G and two vertices s, t such that there exists a Hamiltonian path from s to t in G if and only if ϕ is satisfiable.

1. **Variable gadgets:**
   For each variable x_i, create a diamond-shaped gadget. Inside this gadget:
   - Add special clause nodes, one for the start node and the other for the end node for each C_i. C is the total number of clauses. So you will have $2C$ clause nodes inside each diamond-shaped gadget.
   - Connect these clause nodes using backward and forward arrows to allow flexible traversal within the variable gadget.
2. **Clause gadgets:**
   - On the right-hand side, create clause gadgets $C_1, C_2, \ldots, C_m$, one for each clause. Each clause gadget is a single node. These are separate from the clause nodes inside the variable gadgets. In total, there are exactly C clause gadgets.
3. **Connecting literals to clause gadgets:**
   For each literal that appears in a clause C_j, do the following:
   - If the literal x_i appears positively in clause C_j:
     - Draw a forward arrow from the **start node** of the corresponding clause gadget (in the x_i diamond) to the clause gadget C_j.
     - Then, draw a backward arrow back to the **end node** of that same clause gadget.
   - If the literal x_i appears negated in clause C_j:
     - Draw a forward arrow from the **end node** of the corresponding clause node to clause gadget C_j.
     - Then, return with a backward arrow to the **start node**.
   - Repeat this process for every literal in every clause (note that a literal may appear in multiple clauses).

Conclusion

$$\boxed{\textit{3SAT} \text{ is satisfiable} \iff G \text{ has a } \textit{HAMPATH}}$$

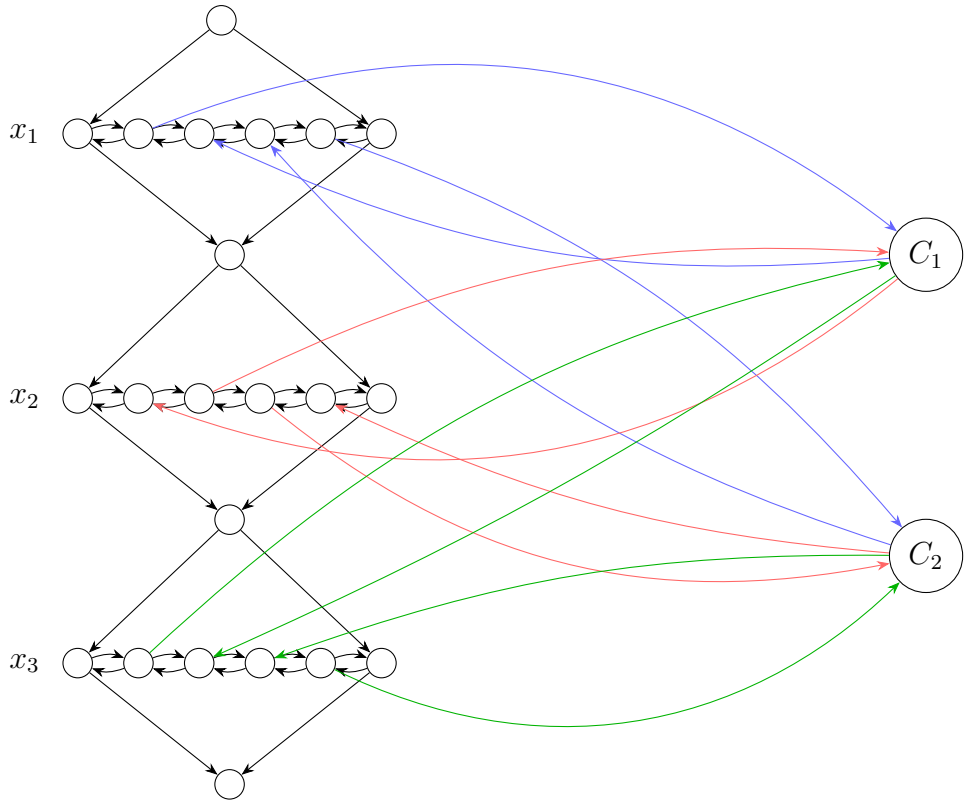
For a rigorous explanation of the correctness of this reduction, see Michael Sipser's *Introduction to the Theory of Computation*.

Example 3.4. *Reduction from 3SAT to HAMPATH.*

You are given a 2-clause 3-CNF satisfiable formula as follows:

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

The resulting graph is illustrated below:



3.5 ***UHAMPATH***

The *UHAMPATH* problem is a special form of the *HAMPATH* problem, where we consider an undirected version of the Hamiltonian path problem. This problem is notable because *UHAMPATH* is one of Karp's 21 NP-complete problems.

Definition 3.5. *Given an undirected graph $G = (V, E)$ and vertices $s, t \in V$, the UHAMPATH Problem asks whether there exists a path from s to t that visits every vertex in V exactly once.*

Theorem 3.5. *UHAMPATH decision problem is NP-complete.*

Proof:

UHAMPATH is in class NP because, given a sequence of vertices, we can verify in polynomial time whether it is a valid Hamiltonian path from s to t. We will show the NP-completeness of the *UHAMPATH* by giving the reduction from the *HAMPATH* problem. I will follow the same construction procedure as explained in the course textbook by Michael Sipser. (Sipser, 2012)

The key idea of the reduction is in *HAMPATH*, edges have directions, so you can traverse from vertex u to vertex v only if the directed edge $u \to v$ exists. However, in *UHAMPATH*,

all edges are undirected. So, we design the undirected edges in a way that the path is naturally forced to move forward.

Construction of the reduction:

Suppose you are given a directed graph G. Then G has a Hamiltonian path from node s to node t if and only if the corresponding undirected graph G' has a Hamiltonian path from s' to t'.

1. For each node u in G, except s and t, create three new nodes: u_{in}, u_{mid}, and u_{out}. For the special nodes s and t, create only s_{out} and t_{in}, respectively.
2. Add the following edges to construct the undirected graph G':
   - For each node u, add edges $(u_{\text{in}}, u_{\text{mid}})$ and $(u_{\text{mid}}, u_{\text{out}})$.
   - If there is an edge (u, v) in G, add an undirected edge $(u_{\text{out}}, v_{\text{in}})$ in G'.

Correctness of the reduction:

$\Rightarrow$ If G has a Hamiltonian path from s to t, then G' has one from s_{out} to t_{in}:

We can build a path in G' of the form: $s_{\text{out}}, u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, \ldots, t_{\text{in}}$ following the same order of vertices as the Hamiltonian path in G. By the construction, to move from u^{out} to v^{in} in G' you have to make sure that there is a directed edge (u, v) in G. The three nodes for each vertex make sure we visit it exactly once. So, this gives us a valid Hamiltonian path in G'.

$\Leftarrow$ If G' has a Hamiltonian path from s_{out} to t_{in}, then G has a Hamiltonian path from s to t:

We follow the path in G' starting from s_{out}. The next node must be some u_{in} (only such nodes are adjacent to s_{out}). Then, to visit all three nodes of the triple, the path must go through u_{mid} and then u_{out}. From there, it can go only to another v_{in} such that $(u_{\text{out}}, v_{\text{in}})$ is an edge, meaning (u, v) was an edge in G. This structure forces the path to simulate a valid directed Hamiltonian path in G.

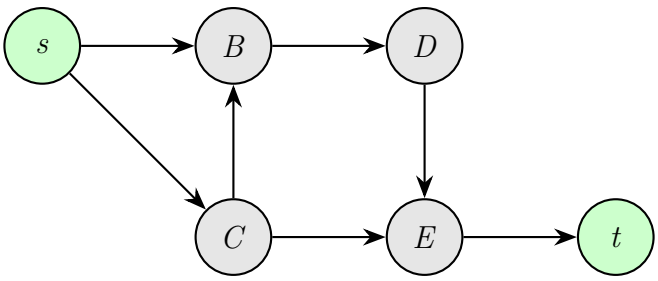
Conclusion

$$\boxed{G' \textit{ has UHAMPATH} \iff G \text{ has a } \textit{HAMPATH}}$$

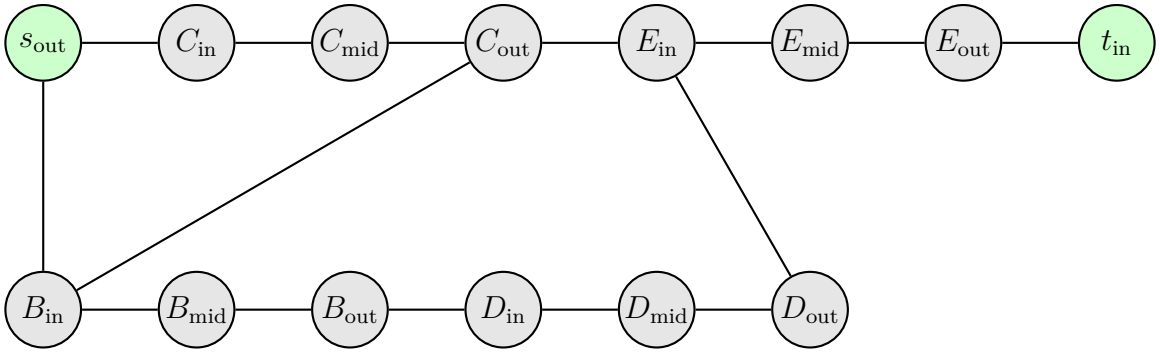
Example 3.5. *Reduction from HAMPATH to UHAMPATH*

We are given the following directed graph with a hamiltonian path:

$$s \to C \to B \to D \to E \to t$$



The undirected graph with a Hamiltonian Path from s to t is the following graph:



3.6 ***VERTEX COVER***

Another interesting problem, *Vertex Cover* (also known as *Node Cover* in Karp's list of 21 NP-complete problems), is NP-complete.

Definition 3.6. *Given an undirected graph $G = (V, E)$ and an integer k, the Vertex Cover decision problem asks whether there exists a subset of vertices $C \subseteq V$ with size at most k such that every edge in E has at least one endpoint in C.*

Theorem 3.6. *The Vertex Cover decision problem is NP-complete.*

Proof:

Given a vertex cover of size k (the certificate), we can easily verify the solution in polynomial time, so the problem is in NP. To demonstrate that the problem is NP-complete, we will introduce two reductions: first from *3SAT*, and then from *CLIQUE* to *Vertex Cover*. Note that while Michael Sipser presents the reduction from *3SAT*, Karp originally suggested the reduction from *CLIQUE*.

Reduction from *3SAT*

Suppose we are given a *3SAT* formula ϕ with variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$. We construct a graph G such that G has a vertex cover of size $k = n + 2m$ if and only if ϕ is satisfiable.

Construction of the reduction:

1. **Variable gadgets:** For each variable x_i, create two nodes labeled x_i and $\neg x_i$, and connect them with an edge.
2. **Clause gadgets:** For each clause $C_j = (\ell_{j1} \vee \ell_{j2} \vee \ell_{j3})$, where each ℓ_{ji} is a literal (e.g., x_k or $\neg x_k$), create three nodes for the literals in the clause, and connect them in a triangle.
3. **Connecting clause to variable nodes:** For each literal node ℓ_{ji} in a clause gadget, add an edge between it and the corresponding variable gadget node with the same label.

4. **Total nodes and vertex cover size:**

   - There are $2n$ nodes from variable gadgets.
   - There are $3m$ nodes from clause gadgets (each clause adds a triangle of 3 nodes).
   - The size of the vertex cover we aim for is $k = n + 2m$, since:
     - We pick 1 node from each variable pair (to satisfy the variable edge),
     - and 2 nodes from each triangle (to cover the clause triangle edges).

If you would like a clear and detailed explanation of why this reduction works, It is highly recommended to check out the book *Introduction to the Theory of Computation* by M. Sipser (Sipser, 2012).

Conclusion

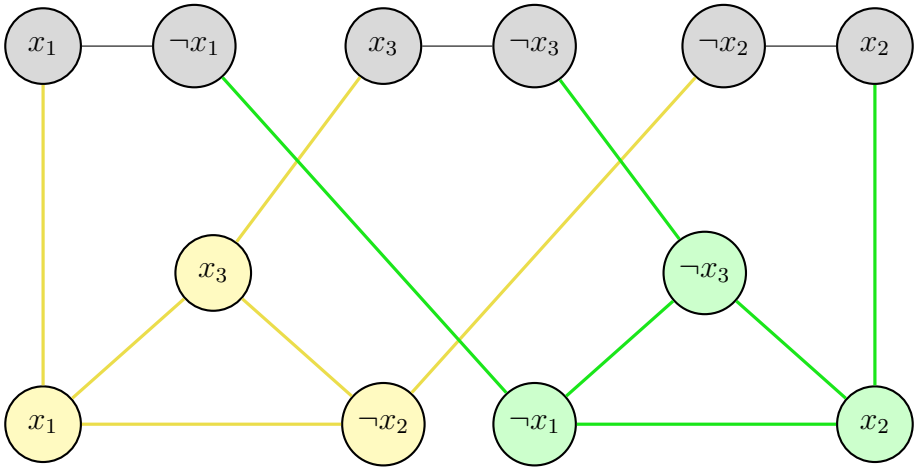
$$\boxed{G \text{ has a Vertex Cover of size } |V| - k \iff \textit{3SAT} \text{ is satisfiable.}}$$

Example 3.6. *Reduction from 3SAT to Vertex Cover*

Consider the following *3SAT* formula with three variables and two clauses:

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

The resulting graph is illustrated below:



Reduction from *CLIQUE*

The following proof was presented by Karp in his seminal 1972 paper. The concise proof is given as follows:

G' is the complement graph of G,

$l = |V| - k$

Construction of the reduction:

Given a graph $G = (V, E)$ and an integer k

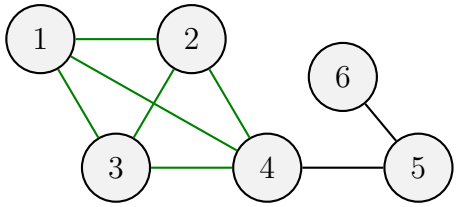
1. Initialize G' with the same vertex set V of G
2. For each pair of vertices (u, v) in graph G' :
   - Add an edge between the vertices u and v if (u, v) is not an edge in G. Otherwise, do not add the edge.
3. Output the pair $\left(\overline{G}, |V| - k\right)$ as the instance for the *Vertex Cover* problem.

If you closely follow the reductions, you might notice that the reduction we just gave is very similar to the reduction from *CLIQUE* to *Independent Set*. This is not a coincidence; indeed, these two problems are complementary problems. Finding an independent set of size k is equivalent to finding a vertex cover of size $|V|$- k. The reduction works for the same reasons as we explained earlier for the *Independent Set* problem.

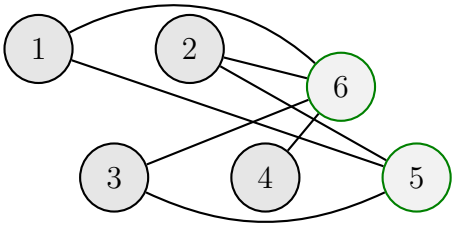
Conclusion

$$\boxed{G' \text{ has a Vertex Cover of size } |V| - k \iff G \text{ has a } k\text{-clique.}}$$

Example 3.7. *Reduction from CLIQUE to Vertex Cover*



Graph G with a clique of size $k = 4$.



Graph G' with a vertex cover of size $l = 2$.

3.7 ***SET COVERING***

The *Set Covering* decision problem asks whether it is possible to cover a given set using at most k subsets from a collection. It is interesting because it is NP-complete, which is a fundamental problem in computational complexity theory.

Definition 3.7. *Given a finite set U, a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U, and an integer k, the problem asks whether there exists a subcollection $\mathcal{S}' \subseteq \mathcal{S}$ with $|\mathcal{S}'| \leq k$ such that $\bigcup_{S \in \mathcal{S}'} S = U$.*

Theorem 3.7. *The Set Covering decision problem is NP-complete.*

Proof:

Given a collection of at most k subset indices (the certificate) whose union covers the universe set, we can verify this certificate by checking the union and the number of subsets in polynomial time. Therefore, the *Set Covering* problem is in class NP. By reducing the

Vertex Cover problem to the *Set Covering* problem, Karp demonstrated that the *Set Covering* problem is also NP-complete. He proposed the following reduction:

Assume $N' = \{1, 2, \ldots, n\}$. *The elements are the arcs of* G'.

S_j *is the set of arcs incident with node* j. $k = l$

Now we will give a clear algorithm and an explanation for this reduction.

Construction of the reduction:

Given a Vertex Cover instance, $G = (V, E)$ and k , to reduce it to Set Cover:

1. Construct the universe set $U = E$, that is all edges in the graph.
2. For each node $j \in V$, define a set $S_j = \{e \in E : e \text{ is incident to } j\}$.
3. The collection $\mathcal{S} = \{S_j : j \in V\}$ forms the sets in the Set Cover instance.

Correctness of this reduction:

$\Rightarrow$ Suppose there exists a vertex cover of size $\leq k$. Let $V' \subseteq V$ be a vertex cover such that $|V'| \leq k$. That means every edge $e \in E$ is incident to at least one vertex in V'. Consider the sets $\{S_j \mid j \in V'\}$. Since each edge is covered by at least one vertex in V', every edge $e \in E$ is contained in at least one of the sets S_j for $j \in V'$. Therefore, the union of these sets covers $U = E$. Hence, $\{S_j \mid j \in V'\}$ is a set cover of size at most k.

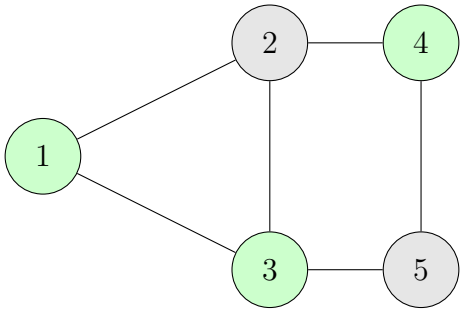
$\Leftarrow$ Suppose there exists a set cover of size $\leq k$ Let $\mathcal{S}' \subseteq \mathcal{S}$ be a collection of sets such that $|\mathcal{S}'| \leq k$ and $\bigcup \mathcal{S}' = U$. Each set $S_j \in \mathcal{S}'$ corresponds to a vertex $j \in V$. Let $V' \subseteq V$ be the corresponding set of vertices. Since the union of the sets in $\mathcal{S}'$ covers all edges in E, it means every edge is incident to at least one vertex in V'. Therefore, V' is a vertex cover of size at most k.

Conclusion

Set instance has a set cover of size $k \iff G$ has a vertex cover of size k

Example 3.8. *Reduction from Vertex Cover to Set Covering*

Consider the following graph G with the vertex cover size of k = 3.



By the reduction, we define the universe set as:

$$U = \{a = \{1, 2\},\ b = \{1, 3\},\ c = \{2, 3\},\ d = \{2, 4\},\ e = \{3, 5\},\ f = \{4, 5\}\},$$

which corresponds to all the edges of the given graph.

For each node j, we define a subset $S_j \subseteq U$ which contains all edges incident to node j:

$$S_1 = \{a, b\}, S_2 = \{a, c, d\}, S_3 = \{b, c, e\}, S_4 = \{d, f\}, S_5 = \{e, f\}$$

The collection of subsets is going to be the union of all S_j:

$$\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5\}$$

By taking the union of $l = k = 3$ subsets, for example S_1, S_3, and S_4 we can cover the universe set U:

$$S_1 \cup S_3 \cup S_4 = \{a, b, c, d, e, f\} = U$$

3.8 ***CHROMATIC NUMBER***

Time to explore one of the most studied problems in graph theory, called *Chromatic Number*, which was proved to be NP-complete by Karp in 1972.

Definition 3.8. *Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, the Chromatic Number decision problem asks whether the vertices of G can be colored using at most k colors such that no two adjacent vertices share the same color.*

Theorem 3.8. *The Chromatic Number decision problem is NP-complete.*

Proof:

The problem is in NP because given a proposed coloring (a certificate), we can verify in polynomial time if each vertex is colored with one of the k colors, and no two connected adjacent vertices share the same color. To show that the *Chromatic Number* problem is NP-complete, Karp gave a reduction from *3SAT*. The concise proof is given as follows:

Assume without loss of generality that $m \geq 4$.

$$\begin{aligned} N &= \{u_1, u_2, ..., u_m\} \cup \{\bar{u}_1, \bar{u}_2, ..., \bar{u}_m\} \cup \{v_1, v_2, ..., v_m\} \\ &\quad \cup \{D_1, D_2, ..., D_r\} \\ A &= \{\{u_i, \bar{u}_i\} | i = 1, 2, ..., n\} \cup \{\{v_i, v_j\} | i \neq j\} \cup \{\{\bar{v}_i, v_j\} | i \neq j\} \\ &\quad \cup \{\{v_i, \bar{x}_j\} | i \neq j\} \cup \{\{u_i, D_f\} | u_i \notin D_f\} \cup \{\{\bar{u}_i, D_f\} | \bar{u}_i \in D_f\} \\ k &= r + 1 \end{aligned}$$

Construction of the reduction:

Suppose we are given a 3SAT formula ϕ with variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$

Create Nodes:

- For each variable x_i create:

1. u_i: node representing variable x_i
2. $\neg u_i$: node representing variable $\neg x_i$
3. v_i: node enforcing that each variable chooses one truth value.

- For each clause C_f, create D_f: node corresponding to clause f .
- So in total, you will have $3n$ variable-related nodes and r clause nodes.

Add Edges:

1. Add an edge between each pair $(u_i, \neg u_i)$ to force them to take different colors.
2. The set $\{v_1, \ldots, v_n\}$ forms a clique, so this requires n different colors.
3. Connect each v_i to all u_j and $\neg u_j$ for $i \neq j$, so only u_i and $\neg u_i$ are allowed to share a color with v_i.
4. Connect each clause node D_f to the variable nodes that are **not** in that clause.

Correctness of the reduction:. $\Rightarrow$ Suppose ϕ is satisfiable. As all v_i nodes form a clique, this ensures that we must use at least n different colors. Connecting each v_i to all variable nodes except u_i and $\neg u_i$ ensures that only those two are allowed to share its color. In this way, the truth value represented by the color of v_i can only be passed to the intended variable and not mistakenly influence other variables. Now consider a clause node D_f. Since the clause is satisfied, at least one literal in it is true, which means that its corresponding node (say u) shares a color with some v_i. As D_f is connected only to the nodes corresponding to literals *not* in the clause, it is not adjacent to u. Therefore, assigning D_f the same color as u is valid. This ensures that all D_f nodes can be colored without introducing a new color beyond the n used.

$\Leftarrow$ Suppose the graph is n-colorable. The clique among the v_i nodes forces them to all receive different colors. Doing this establishes a unique color for each variable. Since u_i and $\neg u_i$ are connected, they must receive different colors, so at most one of them can share the color of v_i. This implies that we can define a valid truth assignment: set x_i = true if u_i shares color with v_i, and false otherwise. Each clause node D_f must be colored using one of the colors n, but it is adjacent to all variable nodes that do not appear in its clause. This means it can only reuse the color of a literal that appears in the clause. Doing so implies that at least one literal in each clause is assigned true under the induced assignment. Therefore, the assignment satisfies ϕ.

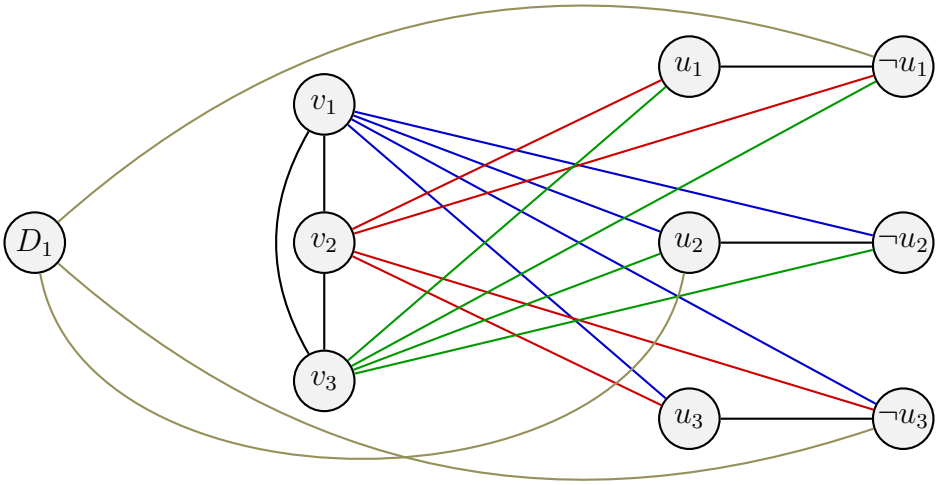
Conclusion

$$\boxed{\phi \text{ is satisfiable } \iff G \text{ is } n \text{ colorable}}$$

Example 3.9. *Reduction from 3SAT to Chromatic Number*

Note that even though in the formal proof Karp set $m \geq 4$ and that is typically for technical convenience. The construction holds for $m \geq 1$. We consider the following *3SAT* formula with three variables and 1 clause: $\phi = (x_1 \vee \neg x_2 \vee x_3)$

The resulting graph is illustrated below:



3.9 ***SET PACKING***

We will proceed with one of the most engaging problems among Karp's 21 NP-complete problems, called *Set Packing*. It was first proposed and proved to be NP-complete by Karp in 1972.

Definition 3.9. *Let U be a finite set of m elements, called the universal set, and let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be a collection of subsets of U. The Set Packing decision problem asks whether there exists a subcollection $\mathcal{S}' \subseteq \mathcal{S}$ of at least k pairwise disjoint sets, that is, sets such that for all $S_i, S_j \in \mathcal{S}'$ with $i \neq j$, we have $S_i \cap S_j = \emptyset$.*

Theorem 3.9. *The Set Packing decision problem is NP-complete.*

Proof:

Given a collection $\mathcal{S}$ and an integer k, a certificate is a subcollection of k subsets. We can verify in polynomial time that all selected subsets belong to $\mathcal{S}$ and are pairwise disjoint. Therefore, the problem is in NP. Karp provided the polynomial time reduction from *CLIQUE* to *Set Packing* to show the NP-Completeness of the problem as follows:

Assume $\mathcal{N} = \{1, 2, \ldots, n\}$. The elements of the sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$ are those two-element subsets of nodes $\{i, j\}$ not in A.
$\mathcal{S}_i = \{\{i, j\} \mid \{i, j\} \notin A\}, \quad i = 1, 2, \ldots, n.$
$l = k.$

Karp's key insight was, G contains a clique of size k if and only if there exists a collection of $l = k$ pairwise disjoint sets in $\mathcal{S}$.

Construction of the reduction:

Suppose the graph $G = (V, E)$ has vertex set $V = \{1, 2, \ldots, n\}$, and let $A = E$ be the set of edges in the graph.

1. Construct the universal (or universe) set:

$$\mathcal{N} = \{\{i, j\} \mid i < j \text{ and } \{i, j\} \notin A\}$$

These are the pairs of vertices (non-edges) that do not exist in G.

2. For each vertex $i \in V$, define a set S_i such that it contains all the non-edges from $\mathcal{N}$ that involve vertex i.

3. Let $l = k$, where l is the number of pairwise disjoint sets selected from the collection $\{S_1, S_2, \ldots, S_n\}$.

Correctness of the reduction:

$\Rightarrow$ Assume there exists a clique $C \subseteq V$ with $|C| = k$. That means for every pair $i, j \in C$, $\{i, j\} \in E$, in other words, they are connected. Now consider the sets $\{\mathcal{S}_i\}_{i \in C}$. Since $\{i, j\} \in E$, so $\{i, j\} \notin \mathcal{N}$, and therefore $\{i, j\} \notin \mathcal{S}_i$ and $\{i, j\} \notin \mathcal{S}_j$. This means, there is no pair $i, j \in C$ such that $\mathcal{S}_i$ and $\mathcal{S}_j$ share a common element. Hence, all $\mathcal{S}_i$ for $i \in C$ are pairwise disjoint. So, if G has a clique of size k, then the Set Packing instance has k disjoint sets.

$\Leftarrow$ Assume that we are given k disjoint sets $\mathcal{S}_{i_1}, \ldots, \mathcal{S}_{i_k}$. Let $C = \{i_1, \ldots, i_k\}$. We claim that C forms a clique in G. Suppose for contradiction that there exist $i, j \in C$ such that $\{i, j\} \notin E$. Then $\{i, j\} \in \mathcal{N}$, so $\{i, j\} \in \mathcal{S}_i$, because $\mathcal{S}_i$ includes all non-edges involving i, and $\{i, j\} \in \mathcal{S}_j$, for the same reason.

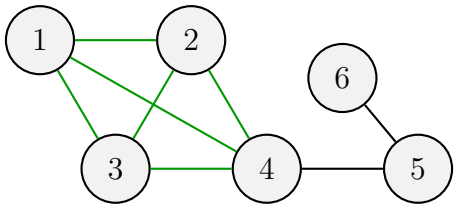
This means $\mathcal{S}_i \cap \mathcal{S}_j \neq \emptyset$, which contradicts our initial assumption that the sets are disjoint. So, all pairs in C must be connected which implies that C is a clique of size k. So, if the Set Packing instance has k disjoint sets, then G has a clique of size k.

Conclusion

$$\boxed{G \text{ has a clique of size } k \iff \mathcal{S} \text{ has a set packing of size } k}$$

Example 3.10. *Reduction from CLIQUE to Set Packing*

Consider the following graph G with k=4 clique:



We claim that the Set Packing instance must have 4 pairwise disjoint sets.

Let $V = \{1, 2, 3, 4, 5, 6\}$ and $A = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}, \{5, 6\}\}$.

1. $\mathcal{N} = \{\{1, 5\}, \{1, 6\}, \{2, 5\}, \{2, 6\}, \{3, 5\}, \{3, 6\}, \{4, 6\}\}$
   This is our universe set, which contains the non-edges in the graph G.

2. Now for each vertex $i \in V$, we create a subset of $\mathcal{N}$:

   - $\mathcal{S}_1 = \{\{1, 5\}, \{1, 6\}\}$

- $\mathcal{S}_2 = \{\{2,5\},\{2,6\}\}$
- $\mathcal{S}_3 = \{\{3,5\},\{3,6\}\}$
- $\mathcal{S}_4 = \{\{4,6\}\}$
- $\mathcal{S}_5 = \{\{1,5\},\{2,5\},\{3,5\}\}$
- $\mathcal{S}_6 = \{\{1,6\},\{2,6\},\{3,6\},\{4,6\}\}$

3. Try to pick $l = k = 4$ of these sets that are pairwise disjoint:
   $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, and $\mathcal{S}_4$ do not share any elements, so they are pairwise disjoint.

So, we found 4 pairwise disjoint sets, which implies that our graph contains a clique of size 4, and vice versa.

3.10 ***CLIQUE COVER***

This is an ideal point to introduce the *Clique Cover* problem, which is closely related to the previously discussed *Chromatic Number* problem. These two problems are known to be complementary equivalents of one another. The reduction between them is quite intuitive, especially if one is already familiar with the *Chromatic Number* decision problem. Let us now present the formal definition.

Definition 3.10. *Given an undirected graph $G = (V, E)$ and an integer k, the Clique Cover decision problem asks whether the vertex set V can be partitioned into at most k subsets $V_1, V_2, \ldots, V_k$ such that each V_i induces a clique; that is, every pair of distinct vertices $u, v \in V_i$ satisfies $(u, v) \in E$. In other words, the goal is to cover all vertices using at most k cliques.*

Theorem 3.10. *The Clique Cover decision problem is NP-complete.*

Proof:

It is easy to see that the *Clique Cover* problem belongs to the class NP. A certificate for an instance of the *Clique Cover* problem is a partition of the vertex set into at most k subsets, each of which is claimed to form a clique. Given such a certificate, we can verify in polynomial time whether each subset forms a clique and whether the number of subsets is at most k. To prove the NP-completeness, Karp came up with the brilliant idea that properly coloring a graph G with k colors is equivalent to covering the complement graph $\overline{G}$ with k cliques.

The original proof was given by Karp as follows:

- *G' is the complement graph of G,*
- *$l = k$.*

Construction of the reduction:

Given an undirected k colorable graph $G = (V, E)$:

1. Initialize $\overline{G}$ with the same vertex set V of G

2. For each pair of vertices (u, v) in graph G' :

   - add an edge between the vertices u and v if and only if (u, v) is not an edge in G. Basically, find the complement graph.

3. Output the pair $\left(\overline{G}, l = k\right)$ as the instance for the *Clique Cover* problem.

Correctness of the reduction:

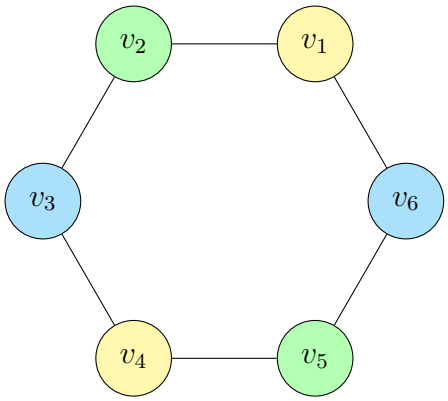
$\Rightarrow$ In a k-coloring of G, each color set forms an independent set (no two vertices in the same class are adjacent). When we take the complement graph $\overline{G}$, these independent sets correspond to cliques beacuse non-edges become edges. Since there are at most k color sets, the clique cover of $\overline{G}$ has size at most k. So, if G is k-colorable, then $\overline{G}$ has a clique cover of size at most k.

$\Leftarrow$ Each clique in $\overline{G}$ corresponds to an independent set in G, because this time edges become non-edges. Since the cliques cover all vertices of $\overline{G}$, the corresponding independent sets cover all vertices of G. Assigning a unique color to each independent set yields a valid k-coloring of G. So if $\overline{G}$ has a clique cover of size at most k, then G is k-colorable.

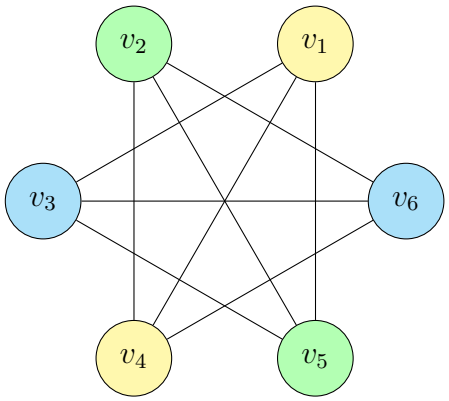
Conclusion

$$\boxed{G \text{ has a clique cover of size } k \iff \overline{G} \text{ is } k\text{-colorable}}$$

Example 3.11. *Reduction from Chromatic Number to Clique Cover*



Consider this $k = 3$ colorable graph G



A corresponding clique cover of size at most $l = k = 3$ in $\overline{G}$

3.11 ***SUBSETSUM***

We shift to another interesting problem, the *SubsetSum* decision problem.

Definition 3.11. *Given a set of positive integers $S = \{a_1, a_2, \ldots, a_n\}$ and a target integer T, the SubsetSum decision problem asks whether there exists a subset $S' \subseteq S$ such that $\sum_{x \in S'} x = t$.*

Theorem 3.11. *The SubsetSum decision problem is NP-complete.*

Proof:

The decision problem belongs to the NP class since, given a proposed subset S' and a target integer t, verifying the correctness of the solution can be done in polynomial time. The NP-completeness of the SubsetSum decision problem is proved by giving different polynomial time reductions through *3-Dimensional Matching, 3SAT*, etc. Following M. Sipser's approach in his book, we will perform a similar reduction of *3SAT* to the *Subset-Sum* problem. (Sipser, 2012)

Construction of the reduction:

Given a 3SAT instance with l variables and k clauses, the reduction algorithm proceeds as follows:

1. Initialize an empty set S.
2. For each variable x_i ($i = 1$ to l):
   (a) Create number y_i (represents x_i = TRUE):
      - Left part (length l): 1 at position i, 0 elsewhere.
      - Right part (length k): for each clause c_j:
        * If literal x_i appears in c_j, set bit $j = 1$.
        * Else, set bit $j = 0$.
      - Concatenate left and right parts to form a binary vector of length $l + k$.
      - Convert the binary vector to a decimal number and add it to S as y_i.

   (b) Create number z_i (represents x_i = FALSE):
      - Left part (length l): 1 at position i, 0 elsewhere.
      - Right part (length k): for each clause c_j:
        * If literal $\neg x_i$ appears in c_j, set bit $j = 1$.
        * Else, set bit $j = 0$.
      - Concatenate left and right parts to form a binary vector of length $l + k$.
      - Convert the binary vector to a decimal number and add it to S as z_i.
3. For each clause c_j ($j = 1$ to k):
   - Create two identical numbers g_j and h_j:
     - Binary vector of length $l + k$ with a 1 at position $l + j$ and 0 elsewhere.
     - Convert the binary vector to a decimal number.
     - Add both g_j and h_j to S.
4. Construct the target number t:
   - First l digits: all 1s (to ensure exactly one selection per variable).
   - Last k digits: all 3s (each clause must be satisfied by 1–3 literals and padded using g_j, h_j).
   - Concatenate to form a binary vector of length $l + k$, then convert to decimal $\rightarrow t$.
5. Return the pair (S, t).

M. Sipser gave a rigorous explanation of the correctness of the reduction in his book (Sipser, 2012). It is recommended to check the explanation if you are interested.

3.12 ***KNAPSACK***

We now turn our attention to the *Knapsack* problem, a fundamental topic in computer science and combinatorial optimization. Its classification as NP-complete was established by Karp in his influential 1972 work.

Definition 3.12. *Given a set of n items with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, a knapsack capacity W, and a target value V, the Knapsack decision problem asks if there exists a subset of items whose total weight is at most W and total value is at least V.*

Theorem 3.12. *The Knapsack decision problem is NP-complete*

Proof:

The *Knapsack* problem belongs to the class NP, as any proposed solution can be verified in polynomial time by computing the total weight and value and checking them against the given constraints. The NP-completeness of Knapsack is typically proven via a reduction from the *SubsetSum* problem, which is inherently related, as it represents a special case of Knapsack where each item's value equals its weight.

Construction of the reduction:

Given a SubsetSum instance $\{a_1, a_2, \ldots, a_n\}$ with target T, we construct an equivalent Knapsack instance as follows:

- For each element $a_i \in S$:
  - Set its weight: $w_i = a_i$
  - Set its value: $v_i = a_i$
- Set the knapsack capacity: $W = T$
- Set the target value: $V = T$

Correctness of the reduction:

$\Rightarrow$ If the Subset Sum instance has a solution, then the corresponding Knapsack instance also has a solution.

There exists a subset $S' \subseteq S$ such that

$$\sum_{a_i \in S'} a_i = T.$$

Then, the corresponding Knapsack solution uses items S' with

$$\sum_{a_i \in S'} w_i = \sum_{a_i \in S'} a_i = T = W,$$

and

$$\sum_{a_i \in S'} v_i = \sum_{a_i \in S'} a_i = T = V.$$

$\Leftarrow$ If the Knapsack instance has a solution, then the corresponding Subset Sum instance also has a solution.

There exists a subset S' such that

$$\sum_{a_i \in S'} w_i \leq W = T, \quad \text{and} \quad \sum_{a_i \in S'} v_i \geq V = T.$$

Since $w_i = v_i = a_i$, this implies

$$\sum_{a_i \in S'} a_i \leq T, \quad \text{and} \quad \sum_{a_i \in S'} a_i \geq T,$$

hence

$$\sum_{a_i \in S'} a_i = T,$$

Example 3.12. *Reduction from SubsetSum to Knapsack*

Consider the following *SubsetSum* instance:

Set $S = \{3, 4, 7, 9\}$ and the target sum $T = 13$.

There is a subset of S that sums to 13, for example, $\{4, 9\}$

Following the reduction procedure, we reduce this to a *Knapsack* instance as follows:

Item	Weight w_i	Value v_i
1	3	3
2	4	4
3	7	7
4	9	9

Now the *Knapsack* problem searches for a subset of items whose total weight ≤ 13 and total value ≥ 13. There exists such a subset, for example, choose items 2 and 4: total weight $= 4 + 9 = 13$, total value $= 4 + 9 = 13$

3.13 ***SEQUENCING***

The *Sequencing* problem is widely studied and has a significant impact across multiple domains. Its decision version was shown to be NP-complete by R.M.Karp.

Definition 3.13. *Given a set of jobs with processing times and constraints, the Sequencing decision problem inquires if there is an ordering of the jobs that satisfies all the constraints and meets a given objective threshold.*

Theorem 3.13. *The Sequencing decision problem is NP-complete.*

Proof:

Given a certificate, which is an ordering of the jobs, verifying that all scheduling constraints (such as precedence or deadlines) are satisfied and that the objective function (such as total completion time or makespan) meets a specified threshold can be done in polynomial time. Therefore, the *Sequencing* problem is in the NP class. R.M. Karp gave a reduction through the *Knapsack* problem to show the NP-completeness of the problem as follows:

$$p = r,\ T_i = P_i = a_i,\ D_i = b$$

Construction of the reduction:

We are given an instance of the knapsack problem, defined as follows:

- A set of n items, where each item i has profit $a_i \in \mathbb{N}$ and weight $b_i \in \mathbb{N}$
- A knapsack capacity b
- A required total profit threshold r

We construct the corresponding sequencing problem instance as follows:

1. Initialize an empty list to hold the jobs: `jobs` $\leftarrow [\,]$
2. For each item $i = 1$ to n, convert it into a job as follows:
   - Set the job's **processing time** $T_i = a_i$
   - Set the job's **profit** $P_i = a_i$
   - Set the job's **deadline** $D_i = b_i$
   - Add the job (T_i, D_i, P_i) to the list of jobs
3. Set the sequencing profit goal $p \leftarrow r$
4. Output a sequencing problem instance with a list of jobs $(T_1, D_1, P_1), \ldots, (T_n, D_n, P_n)$ and a profit threshold p

Correctness of the reduction:

$\Rightarrow$ If the Knapsack instance has a solution, then the Sequencing instance does too:

Assume there exists a subset of items S such that the total profit satisfies $\sum_{i \in S} a_i \geq r$ and the total weight satisfies $\sum_{i \in S} b_i \leq b$. We apply the reduction: for each item i, we construct a job with processing time $T_i = a_i$, deadline $D_i = b_i$, and profit $P_i = a_i$. This implies that the jobs corresponding to the items in S can be scheduled with total processing time $\sum_{i \in S} T_i \leq \sum_{i \in S} b_i$. Since each deadline $D_i = b_i$, the jobs can be scheduled such that they finish on time (for example, using earliest deadline first). This implies that the total profit from on-time jobs is at least $\sum_{i \in S} P_i = \sum_{i \in S} a_i \geq r$. Therefore, the Sequencing instance has a valid solution.

$\Leftarrow$ If the Sequencing instance has a solution, then the Knapsack instance does too:

Assume there is a valid schedule of jobs where the total profit of jobs completed on or before their deadlines is at least $p = r$. Let J be the set of such jobs. Each job $j \in J$ corresponds to an item with $a_j = T_j = P_j$, and deadline $D_j = b_j$. Since the jobs in J finish on time in the schedule, the total processing time of the jobs respects their deadlines. This implies the corresponding items can be packed into the knapsack with total weight $\sum_{j \in J} b_j$, which does not exceed the knapsack capacity if the same order is used. The total profit from the items in J is $\sum_{j \in J} a_j \geq r$, satisfying the profit threshold. Therefore, the corresponding Knapsack instance also has a solution.

Conclusion

Knapsack has a solution $\iff$ Sequencing has a solution

3.14 ***PARTITION***

The *Partition* problem, which we are about to examine, appears deceptively simple at first glance, making its NP-completeness somewhat surprising. It has even been described in the literature as *the easiest hard problem* (Hayes, 2002). The Partition problem has been extensively studied in both number theory and computer science. Without further delay, let us present its formal definition, as it is quite intuitive to understand.

Definition 3.14. *Given a set of positive integers S, the Partition decision problem asks whether it is possible to divide S into two subsets S_1 and S_2 such that the sum of elements in S_1 is equal to the sum of elements in S_2.*

Theorem 3.14. *The Partition decision problem is NP-complete*

Proof:

It is easy to see that the problem belongs to the class NP, since given subsets S_1 and S_2, we can verify in polynomial time whether the sum of elements in both subsets is equal. As for the NP-completeness of the problem, there exist polynomial-time reductions from problems such as *SubsetSum*, *Knapsack*, and *3-Dimensional Matching (3DM)*. In fact, the *Partition* problem can be seen as a special case of Subset Sum, where the target sum is exactly half of the total sum of the set S. Note that this reduction is folklore and commonly used in algorithms and complexity theory courses. We include it here for completeness and clarity. In the following, we explore the reduction from the *SubsetSum* problem.

Construction of the reduction:

Given a Subset Sum instance $\{a_1, a_2, \ldots, a_n\}$ with target T, we construct an equivalent Partition instance as follows:

1. Compute the total sum of elements in S. Let $\text{sum} = s_1 + s_2 + \cdots + s_n$.
2. Define a new integer x. Compute $x = \text{sum} - 2T$. Note that x may be negative.
3. Construct a new set S' for the Partition problem. Define $S' = S \cup \{x\}$.

4. Formulate the Partition problem instance. Ask whether the new set S' can be divided into two disjoint subsets A and B such that.$\sum_{a\in A} a = \sum_{b\in B} b$.

Correctness of the reduction:

$\Rightarrow$ The total sum of the new set S' is $\text{sum} + x = \text{sum} + (\text{sum} - 2T) = 2(\text{sum} - T)$. Therefore, a valid partition must split S' into two subsets, each summing to $\text{sum} - T$. Assume there exists a subset $S_1 \subseteq S$ such that $\sum_{s\in S_1} s = T$. Then the remaining elements $S \setminus S_1$ sum to $\text{sum} - T$. By placing the element $x = \text{sum} - 2T$ into S_1, we get:

$$\sum S_1 + x = T + (\text{sum} - 2T) = \text{sum} - T$$

Thus, both subsets sum to $\text{sum} - T$, forming a valid partition.

$\Leftarrow$ Conversely, assume S' can be partitioned into two equal-sum subsets, each summing to $\text{sum} - T$. Since $x \notin S$, it must be in one of the two subsets. Without loss of generality, let us assume it is in subset A. Then, the sum of the rest of A is $\text{sum} - T - x = T$. This implies that the subset $A \setminus \{x\} \subseteq S$ sums to T, solving the original *SubsetSum* instance.

Conclusion

$$\boxed{\textit{SubsetSum} \text{ has a solution} \iff \textit{Partition} \text{ has a solution.}}$$

Example 3.13. *Reduction from SubsetSum to Partition*

Given a Subset Sum instance: $S = \{3, 1, 5, 9, 12\}$ and $T = 9$, we reduce it to an instance of the Partition problem using the following steps:

1. Compute the total sum of S:
$$\text{sum} = 3 + 1 + 5 + 9 + 12 = 30$$
2. Define the new element $x = \text{sum} - 2T$
$$x = 30 - 2 \cdot 9 = 12$$
3. Create the new set for Partition:
$$S' = S \cup \{x\} = \{3, 1, 5, 9, 12, 12\}$$
4. Compute the new total sum:
$$\text{sum}(S') = 30 + 12 = 42 \quad \Rightarrow \quad \text{Target subset sum} = \frac{42}{2} = 21$$
5. Verify the correctness of the reduction.

   Assume subset $\{9\} \subseteq S$ sums to $T = 9$. The remaining elements in S are $\{3, 1, 5, 12\}$, which sum to $30 - 9 = 21$.

   Now place $x = 12$ in the subset with 9:
$$\text{Subset A: } \{9, 12\} \Rightarrow 21 \quad \text{Subset B: } \{3, 1, 5, 12\} \Rightarrow 21$$

   Thus, S' can be partitioned into two equal-sum subsets.

3.15 ***FEEDBACK VERTEX SET***

In many applications, removing cycles from a network is crucial, and the most efficient way to do this leads us to the Feedback Vertex Set problem, which focuses on finding the smallest set of vertices whose removal makes a graph acyclic.

Definition 3.15. *Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, the Feedback Vertex Set problem asks whether there exists a subset $S \subseteq V$ with $|S| \leq k$ such that removing all vertices in S from G, along with all edges connected to them, results in a graph with no cycles.*

Theorem 3.15. *The Feedback Vertex Set decision problem is NP-complete.*

Proof:

To begin with, we observe why the problem belongs to the complexity class NP. Given an undirected graph G and a vertex set $S \subseteq V(G)$ with $|S| \leq k$, we can verify in polynomial time whether removing all vertices in S and adjacent edges results in an acyclic graph. As for NP-completeness, Karp provided a polynomial-time reduction from *Vertex Cover* to *Feedback Vertex Set* in his seminal 1972 paper, thereby proving the problem to be NP-complete as follows:

$V = N'$
$E = \{\langle u, v\rangle \mid \langle u, v\rangle \in A'\}$
$\ell \leq k$

Construction of the reduction:

Given an undirected Vertex Cover instance of a graph $G = (V, E)$ and a vertex cover size k:

1. Set the vertex set of the new directed graph H to be V.
2. For each undirected edge $\{u, v\} \in E$, add a directed edge $\langle u, v\rangle$ in H.
3. Set the feedback vertex set size $\ell = k$.

   Note: There is a possibility that H is already acyclic even when the original graph G has a vertex cover of size k. In such a case, the feedback vertex set is trivially empty. To enforce the presence of cycles in H, we apply the following transformation:
4. For each edge $\{u, v\} \in E$, create a new node x_{uv}.
5. Add the following directed edges to form a cycle: $u \to x_{uv} \to v \to u$.

Correctness of the reduction:

$\Rightarrow$ If G has a vertex cover C of size k, then H has a feedback vertex set S of size $\ell = k$.

Let C be a vertex cover of G. Then, by definition, for every edge $\{u, v\}$ from G, one of u or v is in C.

By construction, we create H in such a way that each edge $\{u, v\}$ becomes a cycle $u \to x_{uv} \to v \to u$. To break this cycle, either u or v must be removed. Since C covers

all edges in G, removing all vertices in C will break all such cycles. Thus, $S = C$ and $|S| = |C| \leq k = \ell$.

$\Leftarrow$ If H has a feedback vertex set S of size ℓ, then G has a vertex cover C of size k.

Let S be a feedback vertex set of H with $|S| \leq \ell = k$. Note that each cycle in H is of the form $u \to x_{uv} \to v \to u$ for every edge $\{u, v\} \in E$. To break such a cycle, at least one of the vertices u or v must be included in S.

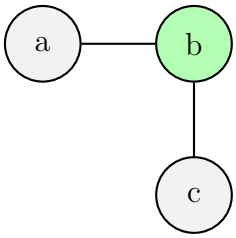
If S contains any intermediate nodes like x_{uv}, we can replace them with either u or v without increasing the size of the set, since x_{uv} only appears in one cycle. Hence, without loss of generality, we can assume that $S \subseteq V(G)$. For every edge $\{u, v\}$, since the corresponding cycle is broken, at least one of u or v is in S. This implies that S is a vertex cover of G. Let $C = S$, then $|C| = |S| \leq \ell = k$, so G has a vertex cover of size at most k.

Conclusion

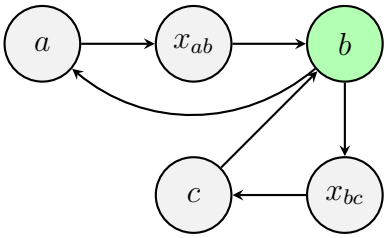
$$\boxed{G \text{ has a vertex cover of size } k \iff H \text{ has a feedback vertex set of size } \ell = k.}$$

Example 3.14. *Reduction from Vertex Cover to Feedback Vertex Set*

Acyclic

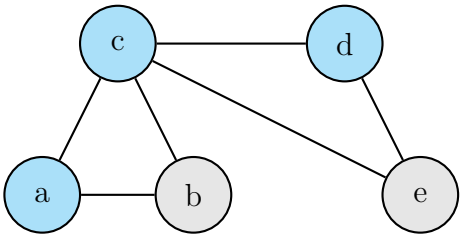


Acyclic Graph G

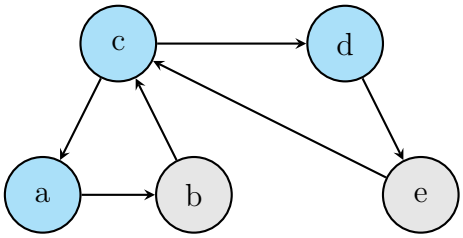


Graph H (Reduced Directed):

Cyclic



Cyclic Graph G



Graph H (Reduced Directed)

4 Applications of NP-Complete Problems

Having explored the definitions, proofs, and interrelationships of various NP-complete problems, we now turn to an equally significant aspect: their real-world applications. Although these problems are central to theoretical computer science, they also arise naturally in diverse practical settings. In this section, we will give a brief overview of how each NP-complete problem discussed in this document connects to real-world challenges.

The Boolean Satisfiability problem remains central to various practical domains. In hardware and software verification, SAT solvers are used to check if a system design meets its specification without bugs (Gong and Zhou, 2017). In artificial intelligence, SAT is at the heart of many planning and reasoning tasks, especially in constraint satisfaction and rule-based decision-making (Biere et al., 2009).

The Clique problem is well appreciated in domains such as bioinformatics, chemoinformatics, examination planning, financial networks, social network analysis, and many more (Wu and Hao, 2015). It is especially useful when trying to identify tightly connected subgroups, such as finding groups of similar compounds or social circles.

The Vertex Cover optimization problem is highly regarded in bioinformatics. It is used to compare DNA and protein sequences by identifying the smallest set of crucial nodes that collectively cover all edges in a genetic interaction graph. This helps in studying genetic differences, organizing bioinformatics workflows, and understanding relationships between genes (Angel, 2019).

One of the most well-known NP-complete problems, the Hamiltonian Path problem, plays a big role in robot motion planning, scheduling problems, network topology, and DNA assembly (Even, 2011), (LaValle, 2006), (Pevzner, 2000). It comes up in cases where visiting all points exactly once is crucial, for instance, designing optimal delivery routes or assembling DNA fragments efficiently.

The Chromatic Number problem, a classic graph coloring challenge, is highly valuable in real-world situations such as course timetabling, exam scheduling, circuit testing, register allocation in compilers, and matrix decomposition for parallel systems (Lewandowski, 1994). Its usefulness shines particularly in avoiding time slot clashes when scheduling classes.

The Knapsack and Subset Sum problems also appear in a wide range of applications, including investment planning, cargo loading, and project selection under budgetary constraints. They are also fundamental in cryptography, production scheduling, and quality-of-service management in network systems (Assi and Haraty, 2018).

In conclusion, NP-complete problems are not merely of theoretical interest; they underpin a wide range of practical tasks in science, engineering, logistics, and beyond. Their recurring presence in real-world challenges underscores their importance and motivates the development of approximation algorithms, heuristics, and SAT/ILP solvers. Even without exact solutions, modeling problems as NP-complete is often the first step toward effective decision-making in complex systems.

Visual Workflow of NP-Completeness Reductions

From	To
SAT	3SAT
3SAT	CLIQUE
3SAT	HAMPATH
3SAT	CHROMATIC NUMBER
3SAT	VERTEX COVER
CLIQUE	VERTEX COVER
CLIQUE	INDEPENDENT SET
CLIQUE	SET PACKING
HAMPATH	UHAMPATH
CHROMATIC NUMBER	CLIQUE COVER
VERTEX COVER	SET COVER
VERTEX COVER	FEEDBACK VERTEX SET
SUBSET SUM	PARTITION
SUBSET SUM	KNAPSACK
KNAPSACK	SEQUENCING

References

Angel, D. (2019). A graph theoretical approach for node covering in tree based architectures and its application to bioinformatics. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 8(1):12.

Assi, M. and Haraty, R. A. (2018). A survey of the knapsack problem. In *2018 International Arab Conference on Information Technology (ACIT)*, pages 1–6. IEEE.

Biere, A., Heule, M. J. H., van Maaren, H., and Walsh, T. (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM.

Even, S. (2011). *Graph algorithms*. Cambridge University Press.

Fortnow, L. (2009). The status of the p versus np problem. *Communications of the ACM*, 52(9):78–86.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Gong, W. and Zhou, X. (2017). A survey of sat solver. In *AIP Conference Proceedings*, volume 1836, page 020059. AIP Publishing LLC.

Hayes, B. (2002). Computing science: The easiest hard problem. *American Scientist*, 90(2):113–117.

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. W., editors, *Complexity of Computer Computations*, pages 85–103. Springer, Boston, MA.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

Lewandowski, G. C. (1994). *Practical implementations and applications of graph coloring*. The University of Wisconsin-Madison.

Papadimitriou, C. H. (2003). Computational complexity. In *Encyclopedia of computer science*, pages 260–265.

Pevzner, P. (2000). *Computational molecular biology: an algorithmic approach*. MIT press.

Sipser, M. (2012). *Introduction to the Theory of Computation*. Cengage Learning, Boston, MA, 3rd edition.

Turing, A. M. et al. (1936). On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5.

Welch, W. J. (1982). Algorithmic complexity: three np-hard problems in computational statistics. *Journal of Statistical Computation and Simulation*, 15(1):17–25.

Wu, Q. and Hao, J.-K. (2015). A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709.