CSCI 4470 Algorithms

Part VII Selected Topics

- 26 Parallel Algorithms
- 27 Online Algorithms
- 28 Matrix Operations
- 29 Linear Programming
- 30 Polynomials and the FFT
- 31 Number-Theoretic Algorithms
- 32 String Matching
- 33 Machine-Learning Algorithms
- **34 NP-Completeness**
- 35 Approximation Algorithms

Chapter 34: NP-Completeness

```
34 NP-Completeness
  34.1 Polynomial time
  34.2 Polynomial-time verification
  34.3 NP-completeness and reducibility
  34.4 NP-completeness proofs
  34.5 NP-complete problems
```

- Tutorial Video

34.1 Polynomial Time

- Definition of polynomial time algorithms.
  - **Polynomial Time Complexity:** Algorithms whose running time is a polynomial function of the size of the input.
  - Polynomial time is represented as $O(n^k)$ where n is the size of the input and k is a constant.
- P, the class of problems that can be solved in polynomial time.
  - **P (Polynomial):** The complexity class of decision problems that can be solved in polynomial time by a deterministic Turing machine.
  - Problems in P are considered “easy” in terms of computational complexity.
- Importance of polynomial time algorithms.
  - Polynomial time algorithms are efficient and practical for real-world problem-solving.
  - Many important problems fall into the P category, making them feasible for computation.

34.2 Polynomial-Time Verification

- Introduction to the concept of verification.
  - **Verification vs. Computation:** Verification focuses on checking whether a solution is correct rather than finding the solution.
  - Problems that allow polynomial-time verification are in NP.
- Polynomial-time verification and decision problems.
  - **Decision Problems:** Problems with a yes/no answer.
  - Certain problems have polynomial-time verification algorithms even though finding a solution may not be polynomial.
- Examples of problems with polynomial-time verification algorithms.
  - Graph problems where a proposed solution can be quickly verified.
  - Certain mathematical problems with efficiently checkable solutions.

34.3 NP-Completeness and Reducibility

- Introduction to NP (nondeterministic polynomial time) class.
  - **NP (Nondeterministic Polynomial):** The class of decision problems for which a proposed solution can be checked quickly.
  - NP includes problems that, if a solution is guessed, can be verified in polynomial time.
- Definition of NP-completeness.
  - **NP-Complete Problems:** A subset of NP problems that are believed to be among the hardest in NP.
  - If any NP-complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time.
- Reducibility and its role in understanding NP-completeness.
  - **Reducibility:** The concept of transforming one problem into another.
  - If Problem A can be reduced to Problem B in polynomial time, and B is solved in polynomial time, A can also be solved efficiently.
- The Cook-Levin theorem and the concept of NP-complete problems.
  - Formally proves the existence of an NP-complete problem.
  - SAT (Boolean satisfiability problem) is the first problem proven to be NP-complete.

34.4 NP-Completeness Proofs

- Understanding the process of proving NP-completeness.
  - **Reduction Techniques:** Showing that if you can solve problem A in polynomial time, you can solve problem B in polynomial time.
  - Transformation from one NP-complete problem to another.
- Techniques for proving a problem is NP-complete.
  - **Proof by Reduction:** Demonstrating that a known NP-complete problem can be reduced to the problem in question.
  - Specific techniques depend on the problems being considered.
- Examples of common NP-complete problems.
  - **SAT (Boolean Satisfiability):** Given a Boolean formula, is there an assignment of truth values that makes it true?
  - **3SAT:** A variation of SAT where each clause has exactly three literals.
  - **Subset Sum:** Given a set of integers, is there a subset that sums to zero?

34.5 NP-Complete Problems

- Introduction to some classic NP-complete problems.
  - **SAT:** The foundational NP-complete problem.
  - **Traveling Salesman Problem (TSP):** Finding the shortest possible tour that visits a given set of cities and returns to the origin city.
- Examples such as the Boolean satisfiability problem (SAT), traveling salesman problem (TSP), and more.
- Implications of a problem being NP-complete.
  - A problem being NP-complete implies it is among the hardest problems in NP.
  - No known polynomial-time algorithm exists for solving NP-complete problems, but none is proven not to exist.
- Relationship between NP-complete problems.
  - The concept of NP-completeness creates a hierarchy among problems in terms of difficulty.
  - If a polynomial-time algorithm is found for one NP-complete problem, it would imply a polynomial-time algorithm for all NP problems.

Before NP-Completeness

Algorithm Type	Algorithm	Best Case	Average Case	Worst Case	Stability	Notes
Sorting Algorithms	Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	-
	Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	-

Algorithm Type	Algorithm	Best Case	Average Case	Worst Case	Stability	Notes
	Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	-
	Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Efficient for small or nearly sorted data
Searching Algorithms	Linear Search	$O(1)$	$O(n)$	$O(n)$	-	Scanning each element sequentially
	Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	-	Requires sorted input
	Selection Algorithm	-	$O(n)$	$O(n^2)$	-	Example: Quickselect
Dynamic Programming	Knapsack Problem	-	$O(nW)$	$O(nW)$ / $O(2^n)$	-	Variance in complexity
	Matrix Chain Multiplication	-	$O(n^3)$	$O(n^3)$	-	-
	Edit Distance	-	$O(mn)$	$O(mn)$	-	m and n are the lengths of two strings
Greedy Algorithms	General Approach	-	-	-	-	Optimal choice at each step
	Huffman Coding	-	$O(n \log n)$	$O(n \log n)$	-	Involves sorting the frequencies
	Fractional Knapsack	-	$O(n \log n)$	$O(n \log n)$	-	Sorting step involved
Graph Algorithms	BFS & DFS	-	$O(V + E)$	$O(V + E)$	-	V and E are vertices and edges

Algorithm Type	Algorithm	Best Case	Average Case	Worst Case	Stability	Notes
	Dijkstra's Algorithm	-	$O(V^2)$	$O(V + E \log V)$	-	With priority queue
	Prim's and Kruskal's Algorithms	-	$O(E \log V)$	$O(E \log V)$	-	Use of a priority queue or sorting of edges
	Bellman-Ford Algorithm	-	$O(VE)$	$O(VE)$	-	For shortest paths in weighted graphs

Additional Considerations

- Algorithm selection depends on context, such as input data size and whether data is already sorted.
- For some algorithms like Quick Sort, the average case is more efficient than the worst case.
- There is often a trade-off between time complexity and factors like memory usage or ease of implementation.

NP-Completeness and NP-Hardness

- **Proof Approach**: Different from previous proofs, focusing on reasoning rather than mathematical equations.
- Class P and Class NP
- **The concept of polynomial time algorithms**, with a focus on complexity classes $\{O(1), n, n\log(n), \{n^2, n^3, ..., n^{10}, \}, 2^n, n!, n^n\}$
  - **tractable**, $\{O(1), n, n\log(n)\}$
  - Polynomial complexities are **tractable**, $\{n^2, n^3, ..., n^{10}, \}$.
  - Non-polynomial (Exponential, Factorial) are **intractable**. $\{2^n, n!, n^n\}$

Class P (Polynomial Time Algorithms)

多項式時間算法

- **Definition**: Class P primarily includes decision problems that can be solved in polynomial time relative to input size.
  Class P 主要包括可以根據輸入大小在多項式時間內解決的決策問題。
- **Characteristics**: Problems in Class P are efficient to solve, making them practically feasible for large inputs.
  Class P 中的問題解決起來效率高，即使對於大型輸入也是實際可行的。
- **Example**: Sorting algorithms like Merge Sort, which sort n items in $O(n \log n)$ time, exemplify problems in Class P.
  如合併排序這樣的排序算法，能在 $O(n \log n)$ 時間內對 n 個項目進行排序，是 Class P 問題的典範。
- **Key Point**: If a problem is in Class P, we have an efficient way to solve it.
  如果一個問題在 Class P 中，我們有一種有效的解決方法。
- Think of Class P as a "fast-solving" group. If a problem is in Class P, it's like having a direct and fast route to your destination. 將 Class P 想像為一個「快速解決」的組。如果一個問題在 Class P 中，就像有一條直接且快速的路徑到達你的目的地。

Examples and Comparison

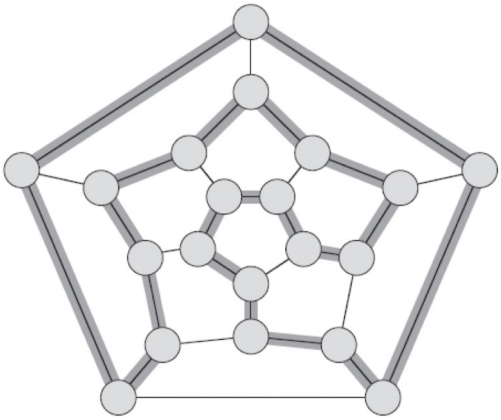
- **Euler's Rule Problem**: Like checking a road map to see if you can tour a city without retracing any road. It's doable quickly.
  就像檢查一張道路地圖，看看你是否可以遊覽一個城市而不重複任何道路。這可以迅速做到。

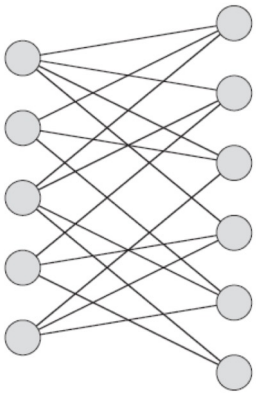
Euler's Rule: 歐拉規則 Polynomial-time problem (多項式時間問題).

- **Description**: Finding an Eulerian Path/Circuit in graphs (connected, directed $G(V, E)$), traversing each edge once. 在圖中尋找歐拉路徑/循環，每條邊遍歷一次。
- **Criteria**: Eulerian Circuit if all vertices have even degrees; Eulerian Path if two vertices have odd degrees. 如果所有頂點度數為偶數則為歐拉循環；如果兩個頂點度數為奇數則為歐拉路徑。
- **Complexity**: $O(V + E)$, efficient and practical.$O(V + E)$，高效且實用。
- **Efficient Solution**: The problem can be solved efficiently, making it a notable example in polynomial-time computable problems. 此問題可以高效解決，使其成為多項式時間可計算問題中的顯著例子。

Hamiltonian Cycle: 哈密頓循環 NP-complete problem (NP 完全問題).

- **Overview**: Searches for a cycle in a graph $G(V, E)$ visiting each vertex once. More complex than Euler's Rule, lacking a polynomial-time solution.
  在圖 $G(V, E)$ 中尋找訪問每個頂點一次的循環。比歐拉規則更複雜，缺乏多項式時間解決方案。
- Like trying to find a specific route that visits every city exactly once in a large country. It's much more complex and harder to do. 就像試圖在一個大國家中找到一條恰好只訪問每個城市一次的特定路線。這要複雜得多，也難做得多。
- **NP-Complete**: Represents one of the hardest problems in NP, with no efficient solution for large graphs. 代表 NP 中最困難的問題之一，對於大型圖沒有高效的解決方案。
- **Challenge**: Equivalent to finding a specific route visiting every city exactly once in a large country, demonstrating its complexity. 相當於在一個大國家中找到一條恰好只訪問每個城市一次的特定路線，展示了其複雜性。





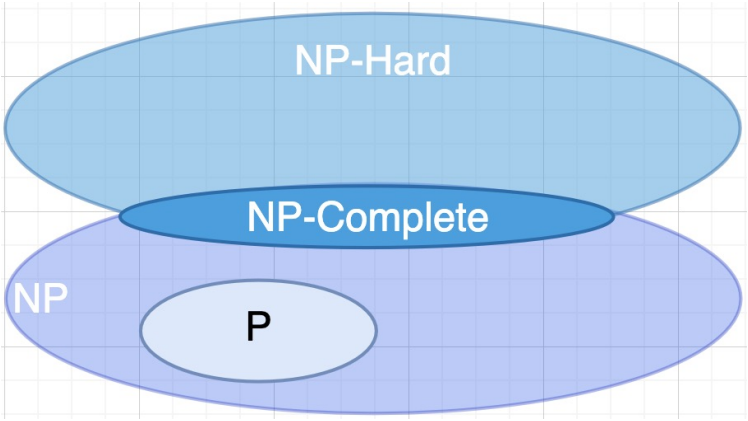
Class NP (Non-deterministic Polynomial Time)

非確定性多項式時間

- **Class NP**: Known for solution verifiability rather than easy solvability. 以解決方案的可驗證性而非易解性著稱。
- **Class NP Definition**: Class NP encompasses decision problems whose solutions can be verified quickly in polynomial time. Class NP 包括那些其解決方案可在多項式時間內快速驗證的決策問題。
- **Nondeterministic and O(1)**: Theoretically, a nondeterministic machine can guess a correct solution instantly (in O(1) time) and verify it in polynomial time. 理論上，非確定性機器可以瞬間（在 O(1) 時間內）猜出正確的解決方案並在多項式時間內進行驗證。
- **Example**: Sudoku puzzles demonstrate NP; solutions are quickly verifiable, but not necessarily quickly found. 數獨謎題展示了 NP 的特性；解決方案可以快速驗證，但不一定快速找到。
- **Key Point**: Class NP emphasizes solution verification over discovery. Class NP 強調解決方案的驗證而非發現。
- **NP-Complete and P in NP**: NP-complete problems, the most challenging subset of NP, also intersect with NP-hard. While all solvable problems in P are verifiable in NP, not all verifiable problems in NP are easily solvable.

NP-完全問題是 NP 中最具挑戰性的子集，也與 NP-困難相交。雖然 P 中所有可解決的問題在 NP 中都可驗證，但並非所有在 NP 中可驗證的問題都容易解決。

- **Subset of NP**: NP-complete problems are both in NP and NP-hard.



- Think of Class NP as a “fast-checking” group. Solving the problem might be like navigating a maze without a map, but once you’re told the exit’s location, you can quickly verify if that’s correct. 將 Class NP 想像為一個「快速檢查」的組。解決問題可能像是在沒有地圖的情況下穿越迷宮，但一旦你被告知出口的位置，你就可以迅速驗證那是否正確。

Understanding the Relationship of P & NP

- **P & NP Relationship**: All P problems are in NP, as solving quickly implies quick verification. However, not every NP problem is necessarily in P, as quick verification doesn’t always mean quick solving.
  - **P 與 NP 關係**：所有 P 類問題也在 NP 中，因快速解決意味著可以快速驗證。但並非所有 NP 問題都在 P 中，因為快速驗證並不總是意味著快速解決。
- All problems in Class P are also in Class NP because if we can solve a problem quickly, we can certainly check a solution quickly. However, not all problems in Class NP are in Class P (as far as we know), because being able to check a solution quickly doesn’t guarantee we can find it quickly.
  Class P 中的所有問題也在 Class NP 中，因為如果我們可以快速解決一個問題，我們當然可以快速檢查一個解決方案。然而，並非所有 Class NP 中的問題都在 Class P 中（據我們所知），因為能夠快速檢查一個解決方案並不保證我們可以快速找到它。

Decision Problem vs. Optimization Problem

決策問題與最佳化問題

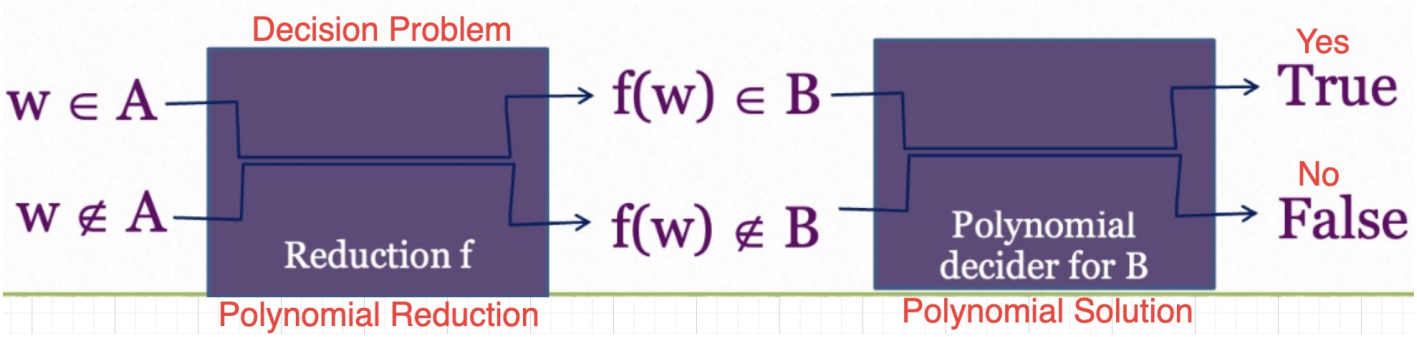
- **NP-Completeness**: Mainly applies to decision problems. NP is the set of problems where solutions can be guessed from many options quickly. 主要適用於決策問題。NP 是指解決方案可以從許多選項中迅速猜測出來的問題集合。
- **Decision Output**: These problems yield a YES or NO answer, with “nondeterministic” meaning a solution is guessed among many in polynomial time. 這些問題產生是或否答案，「非確定性」意味著在多項式時間內從許多選項中猜測出解決方案。
- **Optimization to Decision**: Optimization problems, focused on finding the best solution, can often be reformulated as decision problems. 專注於找到最佳解決方案的最佳化問題，通常可以重新表述為決策問題。

Polynomial Time Reduction & NP-Completeness

多項式時間簡化與 NP 完全性

Polynomial Reduction: Language A is polynomial-time reducible to language B, written $A \leq_p B$, if a polynomial time computable function $f : \sum_A^* \to \sum_B^*$ exists, where for every w, $w \in A$ iff $f(w) \in B$

- **Implications: If $B \in P$, then $A \in NP$, If $B \in NP$, then $A \in NP$, If A is NP-hard, then B is NP-hard.**



Polynomial Time Reduction

- **Concept**: Transforming one problem (A) into another (B) efficiently, specifically in polynomial time. 將一個問題 (A) 高效地轉化為另一個問題 (B)，特別是在多項式時間內。
- **Purpose**: Demonstrates that solving B efficiently implies solving A efficiently. 證明高效地解決 B 意味著也能高效地解決 A。
- **Process**: Conversion of problem A's input into problem B's, ensuring identical YES/NO answers. 將問題 A 的輸入轉換為問題 B 的輸入，確保得到相同的是或否答案。
- ***"Reduction in polynomial time" focuses on the process's efficiency, while "reduction in NP-complete" is about using this process to establish NP-completeness.*** "多項式時間內的簡化"關注過程的效率，而"NP 完全性中的簡化"則是使用此過程來確立 NP 完全性。

Reduction & NP-Completeness

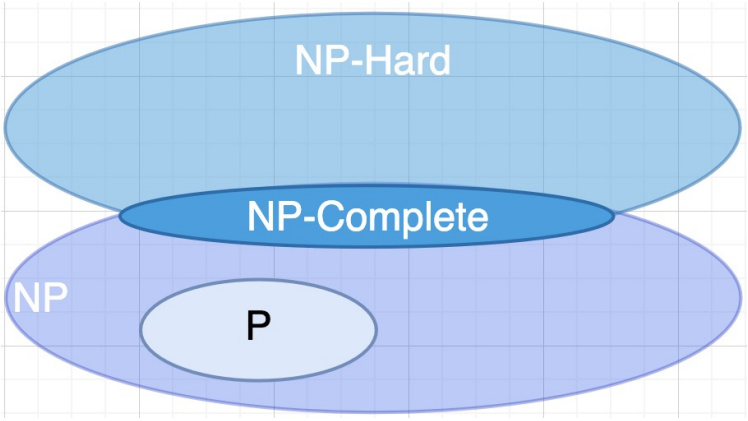
- **Application**: Used to prove NP-completeness by reducing a known NP-complete problem (B) to another problem (A). 用於通過將已知的 NP 完全問題 (B) 簡化為另一個問題 (A) 來證明 NP 完全性。
- **Criteria**: If A is in NP and can be reduced from B, A is also NP-complete. 如果 A 屬於 NP 且可以從 B 簡化，則 A 也是 NP 完全的。
- **Example**: Reducing 3SAT to Graph Coloring; if Graph Coloring is in NP and the reduction is in polynomial time, it's NP-complete. 將 3SAT 簡化為圖著色問題；如果圖著色問題在 NP 中且簡化是在多項式時間內，則它是 NP 完全的。
- Understanding these reductions is crucial in computational complexity to classify problem difficulties and solvability. 理解這些簡化在計算複雜性中至關重要，用於分類問題的難度和可解性。
- ***The relationship between them lies in using polynomial-time reductions to classify the complexity of problems, especially in determining if a problem is in P or NP.***
- **Reduction in NP-Completeness**: Proving NP-completeness often involves reducing a known NP-complete problem B to another problem A in polynomial time. If A is in NP and can be reduced from B, then A is also NP-complete.
  證明 NP 完全性通常涉及將已知的 NP 完全問題 B 在多項式時間內簡化為另一個問題 A。如果 A 屬於 NP 並且可以從 B 簡化，則 A 也是 NP 完全的。
- **NP-Hard and Polynomial Reduction**: If an NP-complete problem C can be polynomially reduced to problem A, then A is NP-hard. A polynomial-time solution for A implies a solution for C, potentially proving P=NP. If A is also in NP, it's NP-complete.
  如果一個 NP 完全問題 C 可以以多項式方式簡化為問題 A，則 A 是 NP 困難的。A 的多項式時間解決方案意味著 C 的解決方案，可能證明了 P=NP。如果 A 也在 NP 中，則它是 NP 完全的。

NP-Hard and NP-Complete Classes

NP-困難與 NP-完全類別

- **NP-Hard**: A problem C is NP-hard if a polynomial-time solution for C means P=NP.
  如果問題 C 的多項式時間解決方案意味著 P=NP，則 C 是 NP-Hard。
- **NP-Complete**: If C is NP-hard and also in NP, then C is NP-complete.
  如果 C 是 NP-Hard 並且也在 NP 中，則 C 是 NP-Complete。
- **Definition of NP-Complete:** Problems with no known polynomial-time solutions but have polynomial-time verifiers.

    - Examples include the Hamiltonian cycle problem.
- **A problem C is NP-hard if finding a polynomial-time solution for C would imply P=NP. If C is also in NP, then C is NP-complete.**
    - If solving problem C in polynomial time means all NP problems can be solved efficiently, C is NP-hard.
      如果多項式時間解決問題 C 意味著所有 NP 問題都可以高效解決，則 C 是 NP-困難的。
    - If C is both in NP (verifiable in polynomial time) and as hard as any NP problem, it's NP-complete.
      如果 C 同時在 NP 中（可在多項式時間內驗證）且像任何 NP 問題一樣困難，則它是 NP-完全的。



- So, if a problem C is NP-hard and also in NP, then it is classified as NP-Complete. This classification is fundamental in computational complexity theory, as it combines both the verifiability of solutions and the complexity of solving the problem itself.
  因此，如果一個問題 C 既是 NP-困難又在 NP 中，那麼它就被分類為 NP-完全。這一分類在計算複雜性理論中是基礎性的，因為它結合了解決方案的可驗證性和解決問題本身的複雜性。

Theory of NP-Completeness

NP-完全性理論

- **Concept**: Demonstrates the absence of polynomial-time solutions for certain problems.
  證明某些問題不存在多項式時間解決方案。
- **Importance**: Helps understand why some algorithms can't be optimized further.
  幫助理解為什麼某些算法無法進一步優化。

Satisfiability Problems in NP-Completeness

- Satisfiability refers to finding an input combination that satisfies the given Boolean expression.
- These problems involve different forms of Boolean expressions and operators.

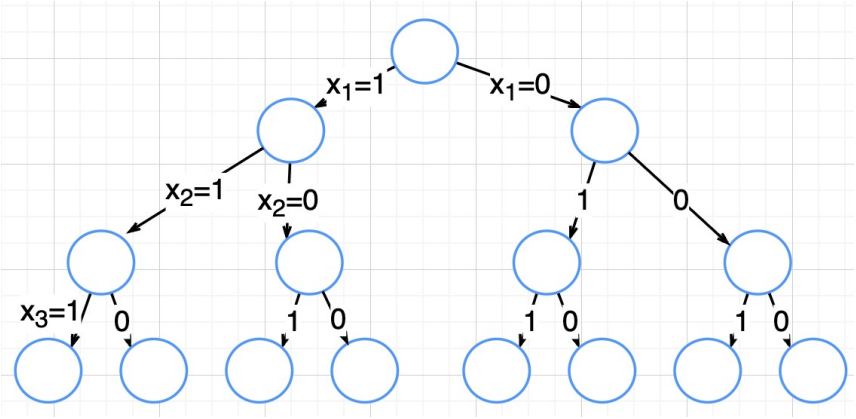
Satisfiability Problems `CIRCUIT-SAT`, `SAT`, and `3-CNF-SAT (3SAT)`: are NP-complete due to their ability to be quickly verified and transformed from other NP problems, showcasing their computational complexity significance.
這些問題之所以是 NP-完全的，是因為它們能夠快速驗證並從其他 NP 問題轉換過來，展示了它們在計算複雜性中的重要性。

1. **Circuit SAT**: **Quick Verification**: Solutions are easy to verify. 解決方案易於驗證。
    - **Transformability**: Any NP problem can be reduced to Circuit SAT in polynomial time. 任何 NP 問題都可以在多項式時間內簡化為 Circuit SAT。
2. **SAT**: **In NP**: Verification of solutions is quick. 解決方案的驗證是快速的。
    - **NP-Complete**: Circuit SAT problems transformable into SAT. Circuit SAT 問題可轉換為 SAT。
3. **3CNF Satisfiability**: **Specific SAT Case**: Limited to three literals per clause. 限於每個子句三個文字。
    - **NP-Complete**: Transformable from general SAT in polynomial time. **NP-完全**：可從一般 SAT 在多項式時間內轉換。

Example 3SAT and NP-Completeness

CNF-Satisfiability Example: - 2^n

$x_i = \{x_1, x_2, x_3, ...\}$ $CNF = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$

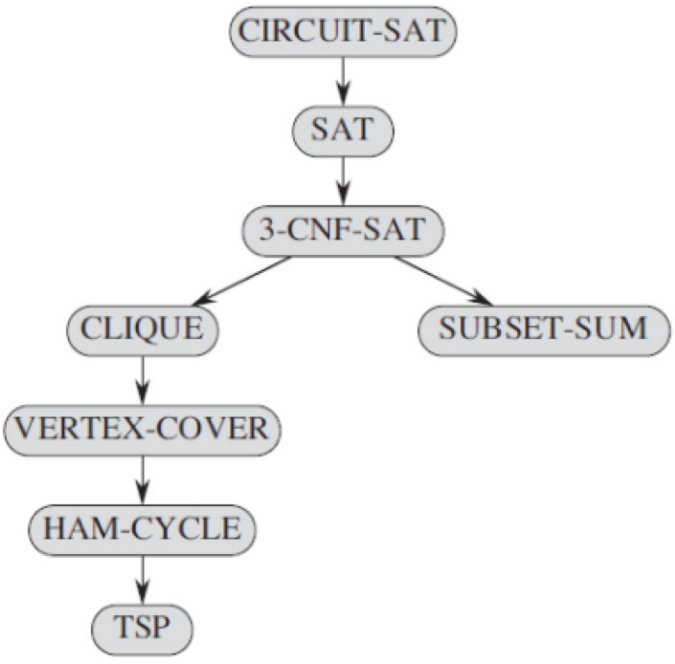


1. **3SAT Overview**: 3SAT, a type of Boolean satisfiability problem, involves determining if there's a true/false assignment for variables in a 3-CNF formula (three literals per clause) that makes the formula true.
2. **Formula Structure**: In 3SAT, formulas like

$$(x_1 \vee x_3 \vee \overline{x}_6) \wedge (\overline{x}_2 \vee x_3 \vee \overline{x}_7) \wedge \ldots$$

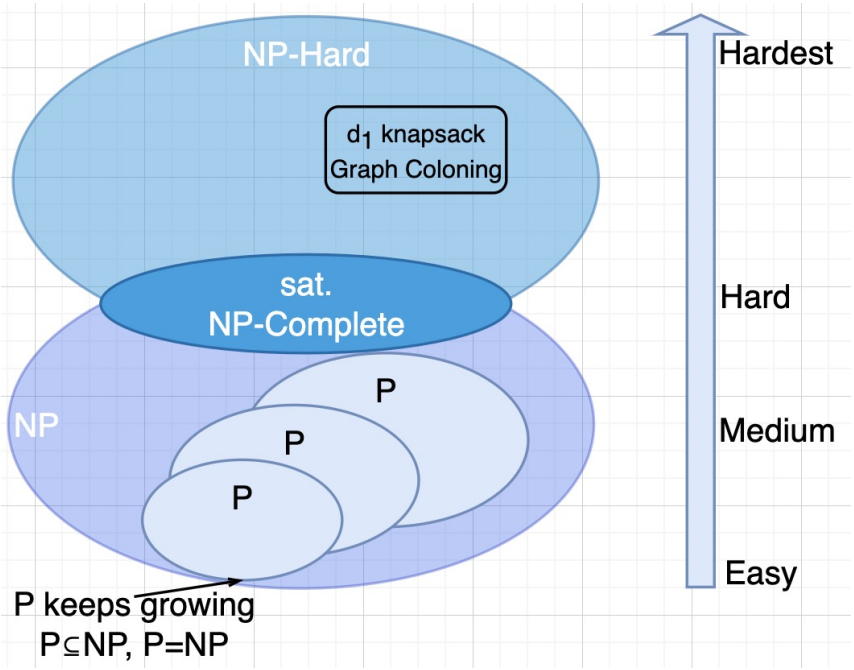
   consist of clauses (grouped literals) connected by AND ($\wedge$).
3. **Components of 3SAT**:
   - **Literal**: Either a variable x_i or its negation $\overline{x}_i$.
   - **Clause**: A combination of three literals connected by OR ($\vee$).
   - **Formula**: An AND ($\wedge$) of several clauses.
4. **3SAT in NP**: Verifying a 3SAT solution is quick (polynomial time), making it a part of NP. A solution 'certificate' can be checked by a verifier for correctness.
5. **NP-Hardness**: 3SAT is NP-hard because any NP problem's solution can be transformed into a 3SAT format. This transformation uses logical circuits (AND, OR, NOT) to create an equivalent 3SAT problem.
6. **NP-Completeness**: 3SAT is NP-complete as it's in NP and all NP problems can be reduced to it. Solving 3SAT in polynomial time would imply solving all NP problems likewise, leading to P=NP.

- `CLIQUE`
- `SUBSET-SUM`
- `VERTEX-COVER`
- `HAM-CYCLE`
- `TSP`



2. **Reduction in NP-Completeness Theory:**

- Concept of reduction: Transforming one problem into another to utilize known solutions.
- Reduction used as a backbone in NP-Completeness proofs.
- Example given of reducing a linear equation problem to a quadratic equation problem.



1. **Practical Implications of Satisfiability:**

- Importance of satisfiability in practical scenarios, like simplifying circuits in digital electronics.

4. **Further Discussion on Reduction and NP-Complete Problems:**

- Reductions used to show that various problems are NP-Complete.
- Example: Showing that Circuit SAT reduces to SAT, which in turn reduces to 3CNF.

2. **Summary and Upcoming Discussions:**

- Planned discussion on specific NP-Complete problems like Vertex Cover and Subset Sum.
- Further explanation of reduction and its role in NP-Completeness theory.

CNF Satisfiability and NP-Completeness

1. Understanding CNF Satisfiability:

- **Definition**: CNF Satisfiability, often referred to as SAT, is the problem of determining if there exists an assignment of truth values (true/false) to variables that makes a CNF formula true.
  定義：CNF 可滿足性問題（通常稱為 SAT），是判斷是否存在一種對變量的真值分配（真/假），使得 CNF 公式為真的問題。
- **CNF Formula**: A formula in CNF is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals (variables or their negations).
  CNF 公式：CNF 中的公式是子句的連接（AND），每個子句是文字（變量或其否定）的分離（OR）。

2. Connection to NP-Completeness:

- **NP-Completeness**: CNF Satisfiability was one of the first problems to be proven NP-complete. This means it is as hard as the hardest problems in NP, and if a polynomial-time algorithm exists for SAT, it exists for all problems in NP.
  NP 完全性：CNF 可滿足性是首批被證明為 NP 完全的問題之一。這意味著它和 NP 中最難的問題一樣困難，如果存在一個對 SAT 的多項式時間算法，那麼它也適用於 NP 中的所有問題。

- **Reduction Concept**: As discussed in your transcript, NP-completeness often involves the concept of reduction. For SAT, any problem in NP can be reduced to SAT in polynomial time.
  化簡概念：正如您的講義中所討論，NP 完全性通常涉及化簡的概念。對於 SAT，NP 中的任何問題都可以在多項式時間內化簡為 SAT。

3. Implications in Computer Science:

- **Hardness**: The NP-completeness of SAT implies that it is unlikely that a polynomial-time algorithm exists for this problem, making it a central problem in computational complexity theory.
  難度：SAT 的 NP 完全性意味著這個問題不太可能存在多項式時間算法，使其成為計算複雜性理論中的核心問題。
- **Uses**: Despite its hardness, SAT solvers are used in various fields like software verification, artificial intelligence, and more.
  用途：儘管 SAT 問題很難，但 SAT 求解器在軟件驗證、人工智能等多個領域中得到應用。

4. Proof Techniques in NP-Completeness (Related to Your Transcript):

- **Reasoning-Based Proofs**: Unlike mathematical proofs, proofs of NP-completeness, including for SAT, involve logical reasoning and reduction from known NP-complete problems.
  基於推理的證明：與數學證明不同，NP 完全性的證明，包括 SAT，涉及邏輯推理和從已知 NP 完全問題的化簡。

Example of Reduction: To show a problem is NP-complete, one typically demonstrates a polynomial-time reduction from a known NP-complete problem (like SAT) to the problem in question.
化簡示例：為了證明一個問題是 NP 完全的，通常需要展示從一個已知的 NP 完全問題（如 SAT）到所討論問題的多項式時間化簡。

5. Efficiency and Tractability (Related to Transcript Discussion):

- **Polynomial vs. Exponential**: While CNF SAT does not have a known polynomial-time solution, efficient heuristics and algorithms exist for certain instances, but they are not guaranteed to work efficiently in all cases.
  多項式與指數：雖然 CNF SAT 沒有已知的多項式時間解決方案，但對某些實例存在有效的啟發式算法，但它們不保證在所有情況下都有效。

6. Real-World Relevance (Tying back to the Transcript):

- **Practical Algorithms**: Despite its theoretical hardness, advancements in SAT solvers have made it a practical tool in various real-world applications.
  實用算法：儘管理論上很困難，但 SAT 求解器的進步使其成為各種實際應用中的實用工具。

NP-Complete Problems:

1. **0/1 Knapsack**: Brute-force is exponential (2^n). Dynamic programming offers a pseudo-polynomial solution, but no true polynomial solution exists.

- The decision version is NP-complete. (決策版本是 NP-完全問題。)
- The optimization version is not in NP. (最佳化版本不屬於 NP。)

2. **Traveling Salesperson (TSP)**: Solutions are either exponential or approximate. No exact polynomial time solution is known.

- The decision version is NP-complete. (決策版本是 NP-完全問題。)

3. **Subset Sum**: Search involves 2^n possibilities. A pseudo-polynomial solution exists through dynamic programming, yet no polynomial solution.

- This is an NP-complete problem. (這是一個 NP-完全問題。)

4. **Graph Coloring**: Exponential complexity in testing color combinations. Lacks a polynomial time solution, especially challenging for more than two colors.

- The decision version is NP-complete. (決策版本是 NP-完全問題。)

5. **Hamiltonian Cycle**: Checking all vertex sequences leads to exponential growth (2^n). No polynomial time solution; problem is NP-complete.

- Determining the existence of a Hamiltonian Cycle is NP-complete. (決定漢密爾頓循環的存在是 NP-完全問題。)

6. **In summary**, these problems have solutions that are either exponential or heuristic in nature, but exact polynomial time solutions remain elusive, marking them as challenging in NP-complete studies.

- In summary, the decision versions of these problems are NP-complete. (總結來說，這些問題的決策版本是 NP-完全的。)
- Optimization versions of such problems are typically not classified as NP. (這類問題的最佳化版本通常不被分類為 NP。)

SET PARTITION VERIFIER

```
SP-Verify(S.c)
1. Let s = ∑_{x∈S}(x)
2. Let t = ∑_{y∈c}(y)
3. If t = s/2
      accept
4. Else
      reject
```

ND ALGORITHM FOR SET PARTITION

```
SP-Verify (S.c)
1. Non-deterministically select A ⊆ S
2. Let s = ∑_{x∈S}(x)
3. Let t = ∑_{y∈c}(y)
4. If t = s/2
      accept
5. Else
      reject
```

Before class

Detailed on Algorithm Complexities

Sorting Algorithms

- **General Complexity**: Sorting algorithms typically range from $O(n \log n)$ to $O(n^2)$ in complexity.
- **Examples**:
  - **Quick Sort**: Average complexity is $O(n \log n)$, but worst-case is $O(n^2)$.
  - **Heap Sort**: Complexity is $O(n \log n)$ in all cases.
  - **Merge Sort**: Guarantees $O(n \log n)$ complexity. √
  - **Insertion Sort**: Efficient for small or nearly sorted data. Best case complexity is $O(n)$ for already sorted data, while average and worst-case complexities are $O(n^2)$. Simple implementation.

Searching Algorithms

- **Linear Search**: Simplest form, $O(n)$ complexity, scanning each element sequentially. √

- **Binary Search**: Requires sorted input, $O(\log n)$ complexity, repeatedly dividing the search interval in half iteratively. √
- **Selection Algorithm**: Used to find the k-th smallest/largest element in a list. Example: Quickselect with $O(n)$ average and $O(n^2)$ worst-case complexity. √

Dynamic Programming

- **Knapsack Problem**: Complexity varies. The 0/1 Knapsack problem has $O(nW)$ complexity, where n is the number of items and W is the weight capacity.
- **Matrix Chain Multiplication**: Complexity is $O(n^3)$ using a dynamic programming approach to find the most efficient way to multiply a chain of matrices. √
- **Edit Distance**: Typically solved with dynamic programming, having $O(mn)$ complexity, where m and n are the lengths of the two strings being compared.

Notice: The **0/1 Knapsack problem** has two solution approaches:

- **Dynamic Programming**: Time complexity is $O(nW)$, more efficient, uses a table to store subproblem results.
- **Recursive Approach**: Time complexity is $O(2^n)$, less efficient, explores all possible combinations.

Greedy Algorithms

- **General Approach**: Make the locally optimal choice at each step, aiming for a globally optimal solution.
- **Complexity**: Often involves sorting as a preliminary step, leading to complexities like $O(n \log n)$. The subsequent steps usually add linear time complexity, $O(n)$.
- **Examples**:
  - **Huffman Coding**: $O(n \log n)$ due to the need for sorting the frequencies of elements.
  - **Fractional Knapsack**: Also $O(n \log n)$ because of the sorting step.

Graph Algorithms

- **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**: Both have $O(V + E)$ complexity, where V is the number of vertices and E is the number of edges in the graph.
- **Dijkstra's Algorithm** (Shortest Path): Complexity is $O(V^2)$ but can be reduced to $O(V + E \log V)$ with the use of a priority queue.
- **Prim's and Kruskal's Algorithms** (Minimum Spanning Tree): Both are $O(E \log V)$, primarily due to the use of a priority queue or sorting of edges.
- **Bellman-Ford Algorithm**: Used for shortest paths in weighted graphs, with complexity $O(VE)$.

Additional Considerations

- **Algorithm Selection**: Choosing the right algorithm depends on the context, such as the size of input data and whether the data is already sorted.
- **Worst vs. Average Case**: For some algorithms like Quick Sort, the average case is more efficient than the worst case.
- **Trade-offs**: There is often a trade-off between time complexity and other factors like memory usage or ease of implementation.

Understanding 3SAT's role in computational complexity highlights the challenges in finding polynomial-time solutions for certain complex problems.
The 3SAT was discovered to be NP-complete by Cook in 1971.
Definition 1. 3SAT: Given a boolean formula of the form:

$$(x_1 \vee x_3 \vee \overline{x}_6) \wedge (\overline{x}_2 \vee x_3 \vee \overline{x}_7) \wedge ...$$

is there an assignment of variables to True and False, such that the entire formula evaluates to True?

We note that a literal is of the form $\{x_i, \overline{x}_i\}$, and both forms of the literal correspond to the variable xi. A clause is made up of the OR of 3 literals, and a formal is the AND of clauses.

$3SAT \in NP$ because we can create a verifier for a certificate. For a given instance of 3SAT, a certificate corresponds to a list of assignments for each variable, and a verifier can compute whether the instances is satisfied, or can be evaluated to true.
Thus the verifier is polynomial time, and the certificate has polynomial length.
It is important to note that this verifier only guarantees that a 3SAT instance is verifiable. To ensure that a 3SAT instance is not verifiable, the algorithm would have to check every variable assignment, which cannot be done in polynomial time.
3SAT is also NP-hard. We give some intuition for this result. Consider any problem in NP. Because it belongs in NP, a nondeterministic polynomial time algorithm exists to solve this problem, or a verifier to check a solution. The verifier is an algorithm that can be implemented as a circuit. Now, the circuit consists of AND, OR and NOT gates, which can be represented as a formula. This formula can be converted to 3SAT form, where each clause has 3 literals, which is equivalent to the original formula. Thus all problems in NP can be converted to 3SAT, and the inputs to the original problem are equivalent to the converted inputs to 3SAT, thus 3SAT is NP-complete.