Week 5 Summary
Foundations of Algorithm Design and Machine Learning
Intractable (Hard) Problems

- Dinesh Kulkarni
- 24BM6JP26

Introduction to Complexity and Problem Reduction

There are two main aspects to complexity: **Problem Complexity** and **Algorithm Complexity**. Although related, these two concepts focus on different things.

Problem Complexity

Problem complexity refers to how difficult a problem is at its core, no matter which algorithm we use to solve it. It sets the minimum amount of time or space required to solve the problem.

Examples:

- **Maximum Finding Problem**
  - Given an array of n numbers, the minimum number of comparisons required to find the maximum is **n−1.**
  - **Lower Bound: Ω(n)**
- **Sorting Problem**
  - Any comparison-based sorting algorithm must perform at least **n logn** comparisons in the worst case.
  - **Lower Bound: Ω(nlogn)**

Algorithm Complexity

Algorithm complexity focuses on how efficiently a particular algorithm solves a problem. It's measured by the amount of time and space the algorithm requires, especially as the input size grows. Refer below image for time and space complexities of a few algorithms.

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

If an algorithm's complexity matches the problem's lower bound, it's considered **optimal**.

Tractable vs. Intractable Problems

A tractable problem has a polynomial-time solution **(O(n^k))**. Examples include sorting (O(n log n)) and Dijkstra's shortest path algorithm ($O(n^2)$ or better with optimizations).

Intractable problems lack polynomial-time solutions, often requiring exponential time **($O(2^n)$)**. An example is the Traveling Salesman Problem (TSP), where the number of possible routes grows exponentially as cities increase.

P vs NP: The Big Question

- **P ⊆ NP:** Every problem in P is also in NP since a solution that can be computed efficiently can also be verified efficiently.
- **Is P = NP?** No one knows! This is one of the biggest unsolved questions in computer science. If proven true, many currently intractable problems would suddenly become solvable in polynomial time.

Class	Description
P	Problems that can be solved efficiently (in polynomial time) by a computer.
NP	Problems that can be solved by a computer, but not necessarily efficiently. Solutions can be verified efficiently.
NP-Hard	At least as hard as any problem in NP. Solving an NP-Hard problem would solve all NP problems.
NP-Complete	Both in NP and NP-Hard. The "hardest" problems in NP.

3-SAT to K-Clique Reduction

Building on the concepts of problem complexity and algorithmic efficiency,let's explore NP problems, NP-completeness, and their significance in computational theory.

The **SAT Problem** (Boolean satisfiability problem)

- Given a Boolean formula in **Conjunctive Normal Form (CNF)**, the task is to find a truth assignment that makes the entire formula true.
    - **Variables**: Boolean variables (x1,x2,x3) that can be either true or false.
    - **Literals**: Variables or their negation (x1 or -x1).
    - **Clauses**: Disjunction (OR) of literals (e.g., x1 ∨ -x2 ∨ x3).
    - **Formula**: A conjunction (AND) of clauses (e.g., (x1 ∨ -x2∨ x3)∧(-x1 ∨ x2))

K-SAT Problem

The K-SAT problem is a generalized form of SAT where each clause contains exactly **K literals**. The task is to determine if there is a way to assign truth values such that the entire formula is satisfied.

Examples:

- **2-SAT:** (x1 ∨ −x2) ∧ (x2 ∨ x3)
- **4-SAT:** (x1 ∨ x2 ∨ −x3 ∨ x4) ∧ (x2 ∨ −x4 ∨ x3 ∨ −x1)

Reduction from 3-SAT to K-Clique Problem

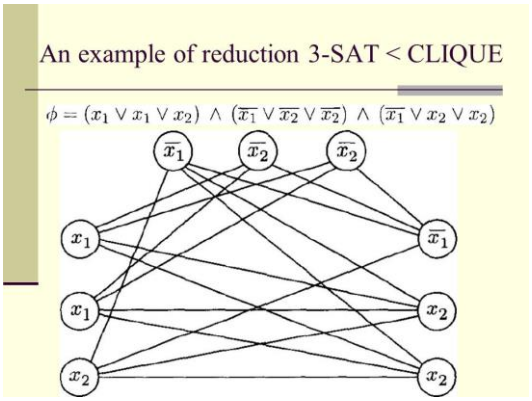
Let's explore how solving the **3-SAT problem** can be transformed into solving the **K-clique problem**.

The 3-SAT Problem:

- The formula consists of clauses with exactly 3 literals.
- The goal is to find a truth assignment that makes the formula true (i.e., at least one literal in each clause must be true).

The K-Clique Problem:

- This is a graph problem where we need to find a **clique** of size **K** (think of clique as a group of connected points in a graph where every point is linked to every other point in the group).
- The task is to determine if such a clique exists in the given graph.



Graph Construction Process:

1. **Vertices for Literals**:
   - Each literal in the 3-SAT formula corresponds to a vertex in the graph.
2. **Clause Representation**:
   - For each clause in the 3-SAT formula, a **triangle** (fully connected subgraph) is created, with one vertex for each literal in the clause.
3. **Incompatibility Handling**:
   - **Incompatible literals** (e.g., xi and -xi) are not connected.
4. **Edges for Compatibility**:
   - Vertices representing literals from different clauses are connected if they are **compatible** (i.e., they don't contradict each other).
5. **Clique Size (K)**:
   - The size of the required clique equals the number of clauses in the 3-SAT formula.

Why the Reduction Works

The reduction ensures that finding a **K-clique** in the constructed graph corresponds to finding a satisfying truth assignment for the 3-SAT formula:

- Each vertex in the K-clique represents a **true literal** from one clause.
- Since the clique includes one vertex from every clause, it guarantees that at least one literal in each clause is true.
- Thus, solving the K-clique problem provides a solution for the 3-SAT problem.

Longest Path to Hamiltonian Cycle Reduction

Let's explore how the **Longest Path Problem** can be reduced to the **Hamiltonian Cycle Problem**.

The Longest Path Problem

The task is to find the **longest simple path** between two specified vertices s and t in a given graph G. A **simple path** is one that visits each vertex at most once.

Problem Definition:

- **Input**: Graph G(V,E) start vertex s, and end vertex t.
- **Output**: The longest simple path from s to t.

Why it's NP-hard: This problem is NP-hard because there is no known polynomial-time algorithm to find the longest path in a general graph.

The Hamiltonian Cycle Problem

The task is to determine if there exists a **cycle** that visits every vertex in a given graph G exactly once and returns to the starting vertex.

Problem Definition:

- **Input**: Graph G(V,E).
- **Output**: True if there is a Hamiltonian cycle, False otherwise.

This problem is known to be NP-complete.

Reduction from Longest Path to Hamiltonian Cycle

To solve the **Longest Path Problem** using the **Hamiltonian Cycle Problem**, we transform the input graph for the Longest Path into a graph suitable for the Hamiltonian Cycle.

Graph Construction Process

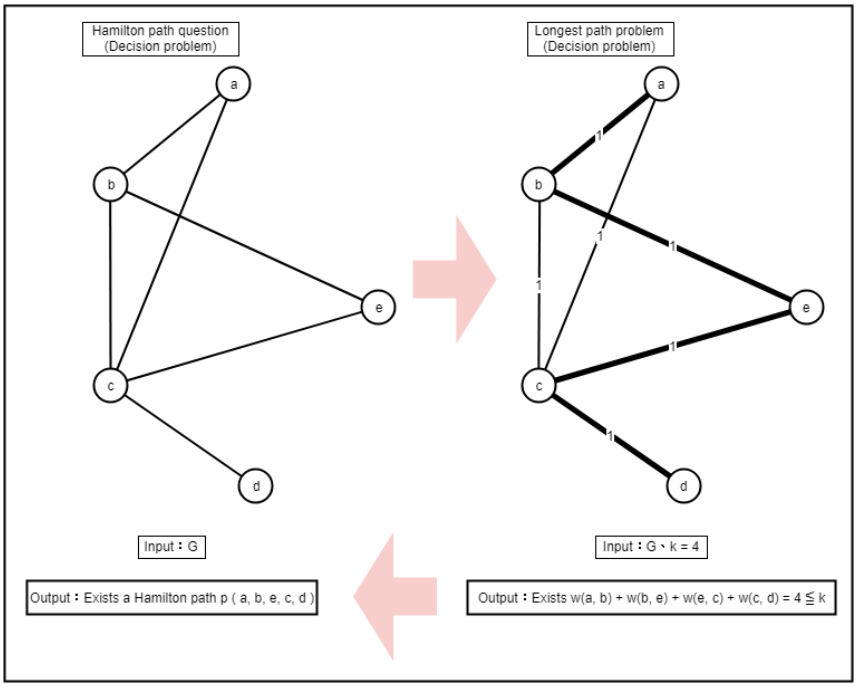
1. **Clone the Input Graph**:
    - Use the original graph G from the Longest Path problem as the base.
2. **Add Auxiliary Vertices and Edges**:
    - Introduce **dummy vertices and edges** that create a cycle-like structure between s and t, which forces the solution to include every vertex along the longest path.
    - This ensures that any Hamiltonian cycle in the new graph corresponds to a simple path between s and t in the original graph.
3. **Hamiltonian Cycle Transformation**:
    - Convert the task of finding the longest path between s and t into finding a Hamiltonian cycle in the newly constructed graph.
    - The resulting Hamiltonian cycle will contain all vertices along the longest path from s to t.

Why the Reduction Works

1. **Path to Cycle Correspondence**:

- Any Hamiltonian cycle in the transformed graph corresponds to a longest path in the original graph.

2. **Length Maximization**:
   - The auxiliary vertices and edges force the cycle to pass through every vertex, ensuring that it represents the longest possible simple path.
3. **Equivalence**:
   - If we can find a Hamiltonian cycle in the constructed graph, we've effectively solved the Longest Path problem.



Test Your Understanding:

1. Check out these 10 mcqs on NP-completeness: geeksforgeeks quiz
2. Show that the Hamiltonian path problem is NP complete