NP-Complete

Classification of Problem

- **Optimization Problem**
  - Problem for which the objective is to maximize or minimize some values.
  - Example,
    - Finding the minimum number of colors needed to color a given graph.
    - Finding the shortest path between two vertices in a graph.
- **Decision Problem**
  - problems for which the answer is a Yes or a No.
  - For example,
    - Whether a given graph can be colored by only 4-colors.
    - Whether a Path exists between 2 nodes with cost <= C.
    - Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

Which problems will we be able to solve in practice?

- Those with polynomial-time algorithms.

- Generally, we think of problems that are solvable by **polynomial-time** algorithms as being **tractable**, or easy,

- and problems that require **superpolynomial** time as being **intractable**, or hard.

Some example

- **Shortest vs. longest simple paths:**
  - **Shortest – P** (can solve in Polynomial time)
  - **Longest – NP** (cannot solve in Polynomial time)
- **Euler tour vs. hamiltonian cycle:**
  - An ***Euler tour of a connected, directed graph*** G =(V, E) is a cycle that traverses each *edge of G exactly once, although* it is allowed to visit each vertex more than once.
    - O(E) time- P problem
  - A ***hamiltonian cycle of*** a directed graph G=(V,E) is a simple cycle that contains each *vertex in V* .
    - Determining whether a directed graph has a hamiltonian cycle is NP-complete. (can solve in Polynomial time)

3 CLASSES/TYPES

- P problems (Polynomial Class Problem)
- NP problems (Non-deterministic Polynomial Class Problem)
- NPC problems (NP Complete class Problem)

P PROBLEMS

- The class P consists of those problems that are solvable in **polynomial** time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k,
  - where n is the size of the input to the problem.
- Most of the problems we examined so far are in P.

NP CLASS

- NP – Nondeterministic Polynomial time.

- The class NP consists of those problems that are "**verifiable**" in polynomial time.

- What do we mean by a problem being **verifiable**?
  - If we were somehow given a guess (Known as "**certificate**") of a solution,
  - then we could **verify** that the **certificate is correct** in polynomial time.

NP class

- So, NP is the class of **decision** problems
  - As we are **verifying** which is a Yes/No answer.
  - So, it is easy to check the correctness of a claimed answer, with the aid of a little extra information.
  - Hence, we aren't **asking for a way to find a solution**, but only to **verify** that **an alleged solution really is correct**.

- Every problem in this class can be **solved** in **exponential** time using exhaustive search.

NON-DETERMINISTIC

- **D**ecision problem solvable in **N**on-deterministic **P**olynomial time.

  - **N**on-deterministic
    - can guess out of polynomially many options in O(1) time.
      - If I provide polynomially many guesses to computer, machine will magically return me a good guess.
    - Good guess - If any guess leads to yes, return that guess
      - If I get a no guess, that means there is absolutely no path that will lead to yes.

Example - Hamiltonian Cycle

- Determining whether a directed graph has a Hamiltonian cycle does not have a polynomial time algorithm (yet!)
- However if someone was to give you a sequence of vertices, **determining whether or not** that sequence forms a Hamiltonian cycle can be done in polynomial time
- Therefore Hamiltonian cycles are in NP

Example : 3-CNF satisfiability

- 3 SAT: A boolean formula is ***satisfiable*** if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.
- CNF – Conjunctive Normal Form. ANDing of clauses of ORs
- Example: $(x_1 \vee x_3 \vee \overline{x_6}) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_7}) \wedge \ldots$
  - Literal – x_1, x_2 , each variable
- Can you set the variables x_1, x_2, x .. ->(T,F) such that the formula returns true.
  - Each clause should return true as the clauses are **ANDed**
- No known polynomial time algorithm (P class) but there is a NP algorithm.

Example : 3-CNF satisfiability

- Lucky Guesses:
  - Guess x_1 = T or F
  - Guess x_2 = T or F
  - Guess x_3 = T or F
  - .
  - .
  - Check Formula
    - Return Yes if True
    - Return No if False
- Guessing at the beginning
- Then Checking
- Verification
  - Satisfy the claim – prove the formula return true in polynomial answer
  - Only works for Yes.
  - For No answer there is no polynomial solution
    - Need exponential time

Example : 3-CNF satisfiability

- Guessing part – result of guessing is called certificates
  - Guess x_1 = T or F
  - Guess x_2 = T or F
  - Guess x_3 = T or F
  - .
  - .
- Verifier step
  - Check Formula

- NP = {Decision problems with poly-size certificates & poly-time verifiers for Yes inputs}

- Summary so far:
  - **P** = problems that can be solved in polynomial time.
  - **NP** = problems for which a solution can be verified in polynomial time.
  - **When** can we say, **P = = NP**? (NP problems are polynomial)
    - When we can find a polynomial solution of an NP problem.
- Hamiltonian-cycle problem is in **NP class**:
  - Cannot be solved in polynomial time.
  - But, a potential solution can be verified in polynomial time.

"NP-COMPLETE" PROBLEMS

- "NP-complete" problems
  - Whose status is unknown.
  - No polynomial-time algorithm has yet been **discovered**.
  - Nor has anyone yet been able to **prove that no polynomial-time** algorithm can exist for any one of them.

"NP-complete" problems

"I can't find an efficient algorithm, but neither can all these famous people."

“NP-COMPLETE” PROBLEMS

- NP-Complete class problem - The problem is in NP, and is as “hard” as any problem in NP – the hardest problems in NP.

- Most believe that NP-Complete problems are *intractable*.
  - As they grow large, we are unable to solve them in reasonable time.
  - Nor has anyone yet been able to prove that no reasonable time algorithm can exist for any one of them.

“NP-complete” problems

- What constitutes <u>reasonable time</u>?
  - Standard working definition: *polynomial time*

- Several NP-complete problems **seem** to be solvable in polynomial time, in **reality** they are not.

NP-Complete Problems Seemed P

- Shortest Paths Problem
  - Given a weighted graph *G* and a source vertex *s*, find the minimum weight path from *s* to each vertex *v*.
  - The running time is ***O*(*VE*)** [Bellman-Ford Algorithm]

    [V=# of vertices, E =# of edges]
- Longest Paths Problem
  - Find the longest path between two vertices
  - The problem seems polynomial, but is **NP-Complete.**

NP-Complete Problems Seemed P

- **Euler Tour Problem**
  - An Euler tour (of a connected graph *G)* is a **cycle** that traverses each **edge** of *G* exactly **once**.
  - The Euler Tour Problem is to determine an Euler tour in a connected graph.
  - The running time is $O(E)$.

- **Hamiltonian Cycle Problem**
  - A Hamiltonian cycle (of a connected graph *G)* is a simple **cycle** that contains each **vertex** of *G* exactly **once**.
  - The Hamiltonian cycle problem: given a graph *G*, does it have a Hamiltonian cycle?
  - The problem seems polynomial, but is **NP-Complete**.

"NP-COMPLETE" PROBLEMS

- The well-known **Traveling Salesman Problem**:
  - **Optimization variant**: a salesman must travel to n cities, visiting each city exactly once and finishing where he begins. How to minimize travel cost?
  - We are given a weighted complete undirected graph G, and we must find a Hamiltonian cycle of G with minimum cost.
  - It is an **NP-Complete problem**.

- ***Decision variant:***
  - If there exists a TSP with cost $\leq k$.

“NP-complete” problems

- NP-Complete problems are the “hardest” problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  - Thus: solve Hamiltonian-cycle problem in $O(n^{100})$ time, you’ve proved that **P = NP**.

“NP-complete” problems

- When we demonstrate that a problem is NP-complete,
  - we are making a statement about **how hard** it is (or at least how hard we think it is), rather than about how easy it is.
  - We are **not** trying to prove **the existence** of an efficient algorithm, but instead that **no efficient algorithm is likely to exist**.

To Become a Good Algorithm Designer

- Become familiar with this remarkable "NP-Complete" class of problems.
- When you need to design an algorithm for any problem,
  - If you can, **try to establish the problem as NP-complete**.
- If you can establish that,
  - Then you will not be able to find a fast algorithm that exactly solves the problem.
- You would then do better to spend your time developing
  - an approximation algorithm, or
  - Solving a tractable special case.

Why Prove NP-Completeness?

- Though nobody has proven that **P != NP**, if you prove a problem NP-Complete, most people accept that it is probably intractable.
- Therefore it can be important to prove that a problem is NP-Complete
  - Don’t need to come up with an efficient algorithm
  - Can instead work on *approximation algorithms*

Reducibility

- A problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q
- Is a linear equation reducible to a quadratic equation?
  - Sure! Let coefficient of the square term be 0

Reducibility

- Can a optimization problem reducible to decision problem?
  - We usually can cast a given optimization problem as a related decision problem by **imposing a bound on the value to be optimized**.
  - For example, a decision problem related to SHORTEST-PATH is PATH:
    - given a directed graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

Reduction

- **Reduction** from problem A ---> problem B = poly-time algorithm converting A inputs ---> equivalent B inputs.
  - Equivalent means same YES/NO answer.
- So, if I know
  - how to solve B and
  - can covert A to B
- Then solving B will solve A as A and B has same Yes/No answer.
  - So, if B ε P then A ε P
  - if B ε NP then A ε NP

Reduction

- Example:
  - Find lcm(m, n): it is unknown.
  - ***But, lcm***$(m, n) = m * n$ / ***gcd***(m, n) <- this relationship is known
    - *And gcd is known, it is polynomial.*
    - *The transformation from lcm to gcd is also polynomial.*
    - *lcm*(m, n) problem is reduced to *gcd*(m, n) problem polynomially.
    - If gcd is polynomial, lcm is also polynomial.
- A problem R can be *reduced* to another problem Q
  - if any instance of R can be rephrased to an instance of Q,
  - the solution to the instance of Q provides a solution to the instance of R.
  - This rephrasing is called "*Reducibility*"
- Intuitively: If R reduces in polynomial time to Q,
  - R is "no harder to solve" than Q.

How to Prove NP-Completeness?

- Objective is to make a statement about how hard a problem is.
  - Only to say that no efficient algorithm is likely to exist.
- **In case of any optimization problem**, we can **transform** it to related decision problem by imposing a bound on the value to be optimized.
  - Why to a decision problem? - A decision problem is in a sense "easier", or at least "no harder" than optimization problem.

Optimization Problem --- transform to→ Decision Problem

How to Prove NP-Completeness? (cont..)

- A decision problem is in a sense "easier", or at least "no harder" than optimization problem. So, we can say,
  - **If, a decision problem is hard**, its related optimization problem is also hard.
- This notion is also applicable when both problems are decision problems.
  - If we can prove a decision **problem A** is easy, its related another decision **problem B** is also easy.

How to Prove NP-Completeness? (cont..)

- NP-completeness is about showing how hard a problem is rather than how easy it is,
- Hence, we use polynomial-time reductions in the opposite way to show that a problem is NP-complete.
  - Suppose we have a decision **problem A** for which we already know that no polynomial-time algorithm can exist.
  - Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B.
  - Now we can say that no polynomial time algorithm can exist for B.

References

- Chapter 34 (Intro part before section 34.1) (Cormen)