Unit - 5

Design and Analysis of Algorithms

Limitations of Algorithm Power

* Lower Bounds: An estimate on the minimum amount of work needed to solve a problem.

→ Efficiency of algorithms can only be compared for the same problem

for eg. for comparison-based sorting, the lower bound is $\Omega(n \log n)$.

Need for Lower Bounds: If there is a gap between the efficiency of the fastest algorithm and the best known lower bound, then one can attempt to find algorithms whose time matches the best known lower bound.

→ If there is no gap between the efficiency of the fastest algorithm and the best lower bound ⇒ no better algorithm can be found.

* Trivial Lower Bounds

→ based on counting the number of items that must be processed in input & generated as output.

→ Since any algorithm must at least "read" all the items it needs to process and "write" all its outputs, such a count yields a trivial lower bound.

eg. Any algorithm for generating all permutations of n distinct items

must be in $\Omega(n!)$ because the size of the input is n!.

*Tight lower Bound : there exists an algorithm with the same efficiency as the lower bound

Problem	lower Bound	Tightness
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n-digit int multiplication	$\Omega(n)$	unknown
nxn matrix multiplication	$\Omega(n^2)$	unknown

Example → : Matrix multiplication requires an $O(n3)$ algorithm

→ Strassen's matrix multiplication has a time complexity of $O(n^{2.81})$

→ It is possible to prove that matrix multiplication requires an algorithm whose time complexity is at least quadratic

→ Whether matrix multiplication can be done in quadratic time remains an open question, since no one has created an algorithm, and also no one has proven that it is not possible to create such an algorithm.

* Methods for Establishing Lower Bounds

(i) trivial lower bounds
(ii) information - theoretic arguments - decision trees
(iii) adversary arguments
(iv) problem reduction

Decision Trees : a convenient models involving comparisons in which :

- internal nodes represent comparison
- leaves represent outcomes

Such a bound is called the information theoretic - lower bound.

Adversary Arguments: a method of proving a lower bound by playing the role of an adversary that makes the algorithm work the hardest by adjusting input.

* Complexity Classes

① Class P - The set of problems that are solvable in polynomial time.

→ Algorithms with time complexity $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^c)$ are considered efficient

→ A problem is efficiently solvable if it can be solved in $O(n^c)$.

→ Problems that can be solved in polynomial time are tractable, those that cannot be solved in polynomial time are intractable.

② Class NP - The set of decision problems solvable in non-deterministic polynomial time.

→ i.e These are the problems that are verifiable in polynomial time.

→ If one were given a 'certificate' of a solution, then ~~one~~ to be of class NP, one would have to verify that the certificate is correct in polynomial time.

→ eg. In the Hamiltonian cycle problem, a certificate would be a sequence $\{v_1, v_2, v_3 \ldots v_n\}$ of $|V|$ vertices. It can be checked in

polynomial time that the sequence contains each of the |V| vertices exactly one.

eg.② for 3-CNF satisfiability, it can be checked in polynomial time if an assignment satisfies the boolean formula.

→ Any problem in P belongs in NP, since if a problem belongs to P, then it is solvable in polynomial time without even being supplied a certificate.

* P vs. NP

→ This problem asks if every problem whose solution can be verified in polynomial time can also be solved in polynomial time.

→ If it turns out that $P \neq NP$, it would mean that there are problems in NP that are harder to compute than to verify.

* Non-Determinism

→ a computational model that is allowed to branch out to check many different possibilities at once.

*

* Class-NP-hard

→ This is the class of problems that are at least as hard as the hardest problems in NP.

→ A problem X is NP-hard if every problem $Y \in NP$ reduces to X.

→ NP-hardness is defined on the basis of polynomial time reductions. If problem A can be reduced to problem B in polynomial time, and the problem B is NP-hard, then problem A is also NP-hard.

→ Examples of NP-hard problems include the TSP, knapsack problem, the graph coloring problem and the satisfability problem.

* Reductions

→ A reduction is a transformation that converts one problem into another problem in a way that preserves the problem's inherent difficulty.

→ Formally, a polynomial-time reduction between 2 problems A and B is a polynomial-time algorithm that maps instances of problem A to instances of problem B, such that for any instance x of problem A:

(i) If x is a 'yes' instance of problem A then the transformed instance f(x) of problem B must also be a 'yes' instance.

Ill for no as well.

→ If $B \in P$ then $A \in P$

If $B \in NP$, then $A \in NP$

If A is NP-hard, then B is NP hard.

* NP- Complete

→ A decision problem is NP- complete if it satisfies 2 conditions:

(a) It belongs to the class NP, meaning that if a solution is provided, it can be verified in polynomial time

(b) It is NP hard - meaning that a NP problem can be polynomial -time reduced to it.

→ In other words, they are some of the hardest problems in NP, and are efficiently verifiable.

In order to prove the completeness of a problem, the following steps have to be followed:

(i) Verify that the solution is in NP - i.e a potential solution's correctness can be verified in polynomial time.

(ii) Show that X is NP-hard. - Reduce from a known NP-complete problem Y to X. This is sufficient to establish the NP- hardness of the problem.

This is because: any problem Z in NP can be reduced to Y and Y can be reduced to X.

⇒ X is at least as hard as the hardest problems in NP.

* 3-SAT

→ Given a Boolean formula of the form:

$(x_1 \vee x_3 \vee \bar{x_6}) \wedge (\bar{x_2} \vee x_3 \vee \bar{x_7}) \wedge \cdots$ the objective is to find an assignment of variables to True and False, such that the entire formula evaluates to True.

→ called the boolean satisfiability problem, it is the first problem shown to be NP-complete.

→ 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, and a verifier can compute whether the instances evaluate to true. The verifier runs in polynomial time.

* NP-completeness of 3-SAT

Intuition for NP-hardness:

→ Consider any problem in NP - where the validity can be checked in polynomial time.

→ Since the verifier runs in polynomial time, we can think of its execution as a program with a polynomial number of lines of code.

→ To analyze this program, we convert each line into machine code, into a boolean circuit representation, with AND, OR and NOT gates.

→ This boolean circuit can be represented as a logical fm,

In 3-SAT format (with 3 ANDs > ~~Q6~~ logical OR operations)

→ By converting the boolean circuit into a 3SAT formula, we are essentially expressing the original problem as an instance of 3-SAT

→ The intuition behind the NP hardness of 3-SAT is that we can transform any problem in NP to an equivalent instance of 3-SAT, allowing us to solve any problem in NP, by solving 3-SAT.

* P vs. NPC Problems

(i) Shortest path vs. longest path

↓ ↓

P np-complete

(ii) Euler Tour vs. Hamiltonian Cycle

↓ ↓

visits each edge exactly once → P

visits each vertex exactly once → NP-C

(iii) 2SAT vs. 3-SAT

↓ ↓

has only 2 literals → solvable in polynomial time

NP-C

* Polynomial - Time Reductions

→ A polynomial - time reduction provides a way to solve a problem A in polynomial time as follows

(i) Given an instance α of problem A, use a polynomial - time reduction algorithm to transform it into an instance β of problem B

(ii) Run the polynomial time - decision algorithm for B on the instance β.

(iii) Use the answer for β as the answer for α.

→ By reducing solving problem A to solving problem B, we use the easiness of B to prove the easiness of A.

→ The symbol $\leq_p$ denotes polynomial - time reducibility.

* Reductions to show hardness

→ Suppose there is a decision problem A for which one already knows that no polynomial - time algorithm can exist. This means that A is a difficult problem that cannot be efficiently solved.

→ Consider another problem B, for which the computational complexity has to be determined.

→ If it can be shown that B is at least as hard as problem A, then it can be concluded that B is also difficult to solve efficiently

→ To do this, assume that there exists a polynomial time algorithm

for problem B. This implies that B can be efficiently solved.

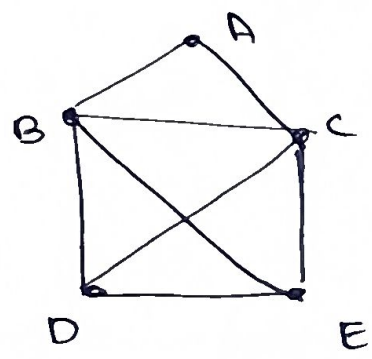
→ If the reduction is used to transform instances of problem A to instances of problem B, and since B can be solved efficiently, A can be solved efficiently as well.

→ This contradicts our initial assumption that problem A is difficult and has no polynomial-time algorithm.

→ This means that B also has no polynomial time solvable algorithm

* Decision Problem - Clique

→ A clique in a graph is a set Q of vertices such that each vertex in Q is connected to every other vertex in Q.



ABC & BCED are cliques.

→ The problem to find in a simple graph G, if there exists a clique of size k?

Theorem - CLIQUE is NP complete.

Step1: To prove that CLIQUE belongs to NP

We need to show that for a given graph $G = (V, E)$, we can efficiently verify whether a given set of vertices V' forms a clique.

→ A clique is a subset of vertices in a graph where each pair of vertices in the subset is connected by a edge

→ To verify whether V' is a clique, check for every pair of vertices u & v in V', if it belongs to E.

→ This verification process take only polynomial time.

Step2: Reduce CLIQUE from a known NP-complete problem (3SAT)

→ Let $\phi = C_1 \wedge C_2 \ldots C_k$ be a boolean formula in 3-CNF with k clauses.

→ For $r = 1, 2 \ldots k$, each clause Cr has exactly 3 distinct literals.

→ Construct a graph G such that ϕ is satisfiable iff G has a clique of size k

(i) For each clause Cr, place a triplet of vertices into V

(ii) Add an edge between 2 vertices v_i^r and v_j^s if

(a) v_i^r and v_j^s are in different triples.

(b) their corresponding literals are consistent, ie one is not the negation of the other.

→ This graph from ϕ can be built in polynomial time

→ It must be shown that this transformation of ϕ into G is a reduction.

Claim1: If ϕ has a satisfying assignment, then G has a clique of size k.

Proof: → Suppose ϕ has a satisfying assignment.

→ Then each clause has at least one literal assigned 1.

→ Picking one such 'true' literal from each clause yields a set V' of k vertices.

→ For any two vertices v_i^r and v_j^s $(r \neq s)$, both literals map to 1 by the satisfying arguments, and thus cannot be complements

→ Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Claim2: If G has a clique of size k, then ϕ has a satisfying assignment.

Proof: → Suppose that G has a clique V' of size k.

→ No edges in G connect vertices in the same triple, so V' contains exactly one vertex per triple.

→ 1 can be assigned to each literation w/o fear of assigning 1 to both a literal and its complement, since G has no edges between inconsistent literals.

→ Each clause is satisfied, so ϕ is satisfied.

* Approximation Algorithms

→ Many problems of practical significance are np- compiae, and only near - optimal solutions in polynomial time.

→ An algorithm that returns near - optimal solutions is called an approximation algorithms.

* Approximation Ratio

→ used to know how accurate an approximation

→ Let sa be an approximate solution and s* be an optimal solution.

Then $r(sa) = \max\left(\frac{f(sa)}{f(s*)}, \frac{f(s*)}{f(sa)}\right)$

f is the function that gives the value of the solution

→ For a minimization problem: $f(s*) < f(sa)$

→ For a maximization problem : $f(s*) > f(sa)$

* c- approximation algorithm

→ A polynomial -time approximation algorithm is said to be a c- approximation algorithm, where $c \geq 1$, if the the accuracy ratio of the approximation it produces does not exceed for any instance of the problem

$$r(sa) \leq c$$

→ Smallest value of c for which the inequality holds= performance ratio

→ Approx. algos should have performance ratios as close to 1 as possible

* Roving Performance Ratios

→ need to know the value of the optimal solution, which is the ans one is looking for

→ Since we do not have the optimal solution, start off with a lower bound for the value of the optimal solution.

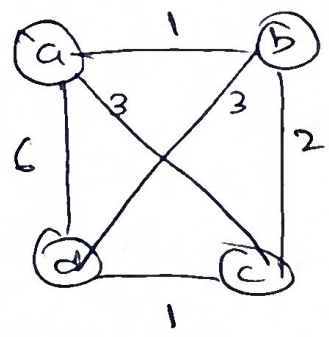
→ The approximation ratio is expressed as the ratio of the approximate solution & the lower bound.

* Approximation Algorithms for the Traveling Salesman Problem

Greedy Strategy : Choose an arbitrary city as the start

Repeat Visit the nearest city until all cities have been visited

Return to starting city.



$$\frac{\text{nearest neighbor algo}}{\text{best soln}} = \frac{10}{8} = 1.25$$

approx ratio depends on each instance

Greedy strategy: add edges in order of increasing edge weights

MST- based algo

(i) Construct an MST

(ii) Remove repeating vertices

~~a b c~~

(ab) (bc) (bd) (de)

~~a b c b d~~

~~a b c b b d e d a~~.

a b c b d e d b a

a b c d e a

Approximation ratio

The twice around the tree algo is a 2-approximation algo for the TSP w/ Euclidean distances

$$2f(s_a) \leq 2f(s^*)$$

remove an edge* from the MST

$$w(T) \geq w(T^*)$$

$$2f(s^*) \geq w(T) \geq w(T^*)$$

$$2f(s^*) > 2w(T^*)$$

shortcuts cannot increase total length

$$2f(s^*) \geq f(s_a)$$

* Approximation Algorithms for Knapsack

(i) compute value by weight ratio

(ii) sort items in decreasing order

(iii) place current item if fits, otherwise go to next item