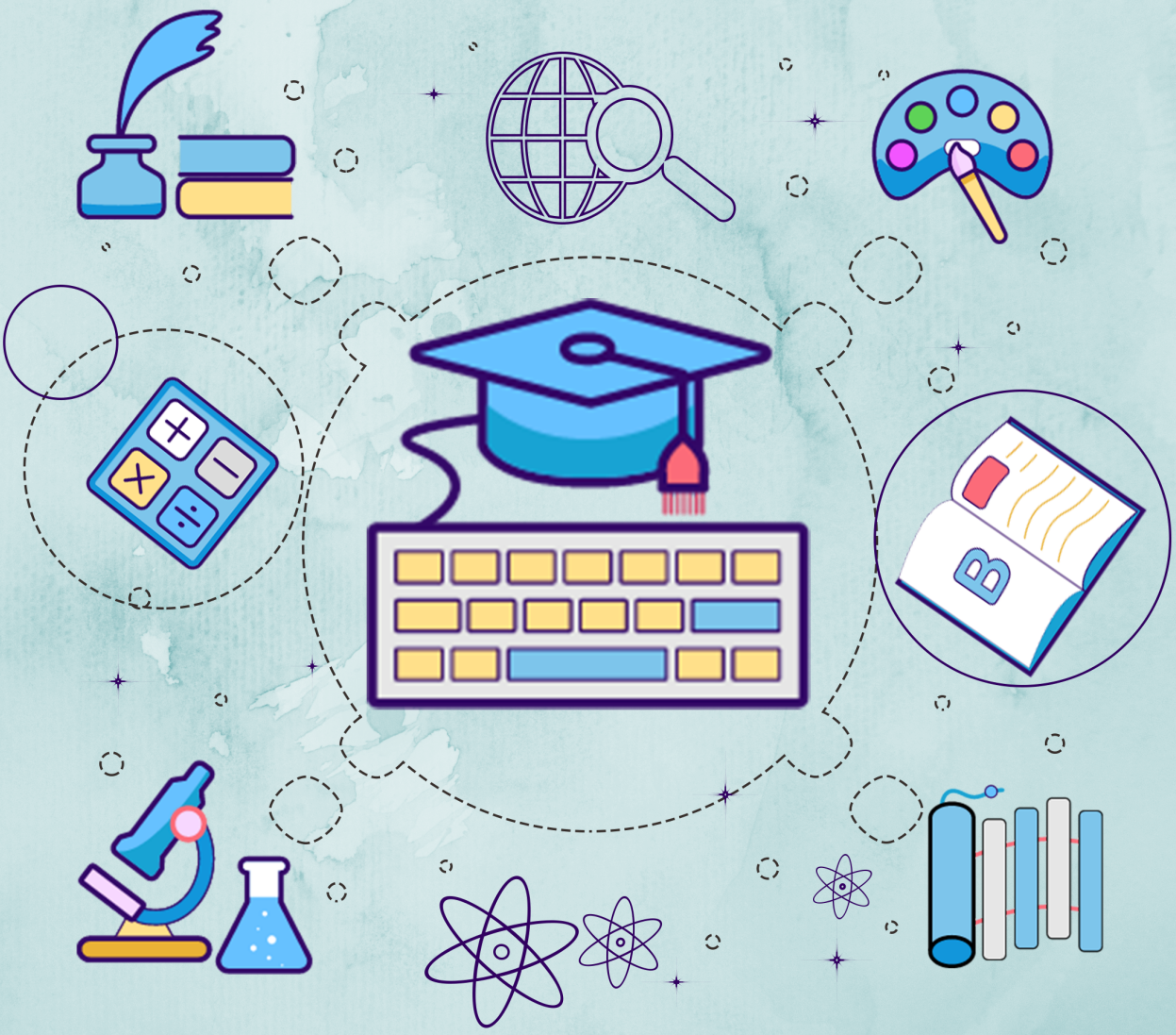
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

ALGORITHM ANALYSIS AND DESIGN

CST 306

Module 5

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

AAD

ALGORITHM ANALYSIS AND DESIGN - CST306

Module 5

Neethu Mathew , CSE Dept. EKCTC

Module-5 (Introduction to Complexity Theory)

Tractable and Intractable Problems, Complexity Classes – P, NP, NP- Hard and NP-Complete Classes- NP Completeness proof of Clique Problem and Vertex Cover Problem- Approximation algorithms- Bin Packing, Graph Coloring. Randomized Algorithms (Definitions of Monte Carlo and Las Vegas algorithms), Randomized version of Quick Sort algorithm with analysis.

Introduction to Complexity Theory

❑ Tractable and Intractable Problems

1. Tractable /easy Problems

- Problem that is solvable by a polynomial time algorithm is tractable
- There is an efficient algorithm to solve it in polynomial time
- Polynomial time => O(n^k) ,for some constant k

  Time taken by an algorithm which are solvable can be O(n^k) ,which is very much polynomial for some constant k
- Problems that can be solvable in a reasonable(polynomial) time.
- Examples (ones with known polynomial time algorithms)

  Sorting

  Searching

  Matrix multiplication

2. Intractable / hard Problems

- Problem that cannot be solvable by a polynomial time algorithm
- unable to solve them in a reasonable time. There is no efficient algorithm to solve it in polynomial time
- Exponential time => O(k^n)
- Examples (ones that have been proven to have non polynomial time algorithm)

  Tower of Hanoi

  TSP

  sudoku

  0/1 knapsack

  graph coloring

Exponential function vs Polynomial

- The function $p(x) = x^3$ is a **polynomial**
- Here the "variable", x, is being raised to some constant power
- The function $f(x)=3^x$ is an **exponential** function
- The variable is the exponent
- **Exponential functions**:
- E.g. $O(2^n)$, $O(n!)$, $O(n^n)$

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
n-log-n	$O(n \times \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(k^n)$, e.g. $O(2^n)$
factorial	$O(n!)$
super-exponential	e.g. $O(n^n)$

these functions are divided into two classes:

- Polynomial functions: Any function that is $O(n^k)$, i.e. bounded from above by n^k for some constant k.
- E.g. O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(n^3)$
- 'Polynomial' is function of the form $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$.
- Exponential functions: The remaining functions.
- E.g. $O(2^n)$, O(n!), $O(n^n)$

- **Deterministic algorithms**

- Algorithms with the property that result of every operation is uniquely defined
- Predictable in terms of output for a certain input

- **Nondeterministic algorithms:**

- Result of every operation is not uniquely defined.
- They are allowed to contain operations whose outcomes are limited to a given set of possibilities

- Many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not.

  Hence, the problems can be categorized as follows :

  - **Optimization Problems**
    - An optimization problem is one which asks, “What is the optimal solution to problem X?”
    - An optimization problem tries to find an optimal solution
    - Result is a number representing an objective value
    - Optimization problems are those for which the objective is to maximize or minimize some values.

      For example, Finding the shortest path between two vertices in a graph.
  - **Decision Problems**

    – A decision problem is one with yes/no answer

    -- ie, A problem is called a decision problem if its output is a simple “no" or “yes"

    (or you may need this of this as true/false, 0/1, accept/reject.)

Complexity Classes :- P, NP, NP-Hard and NP-Complete

P-Class

- Note that P stands for “Polynomial-time”
- **The class P consists of set of 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.
- Formally, an algorithm is polynomial time algorithm, if there exists a polynomial p(n) such that the algorithm can solve any instance of size n in a time O(p(n)).
- These problems are called tractable, or efficiently solvable
- **P:** set of problems that have polynomial-time deterministic algorithms.(class of problems solved by a **deterministic polynomial algorithm**). And a deterministic algorithm is (essentially) one that always computes the correct answer
- P problems being easy to solve

NP-Class

- Note that NP stands for "Non deterministic Polynomial-time"
- **The class NP consists of those problems that are "verifiable" in polynomial time**. In computational complexity theory, NP is a set of decisions problems that can be solvable by a **nondeterministic polynomial algorithm** (or with a non deterministic machine)
- These problems are called intractable or super polynomial.
- NP class contains P class as a subset.
- NP problems being hard to solve.
- Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time
- Solutions can be checked effectively
- NP class problems are verifiable in polynomial time. NP is the class of decision problems for which 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.

P versus NP

P = set of problems that can be solved in polynomial time

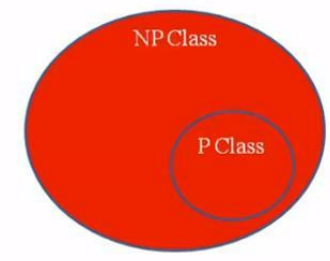
NP = set of problems for which a solution can be verified in polynomial time

P class problems are subset of NP class problems

(P is the set of all decision problems which can be solved in polynomial time by a deterministic turing machine. since it can be solved in polynomial time, and also can be verified in polynomial time. So p is a subset of np)

Open question.....

Does P = NP?



It is not known whether ***P = NP,P ≠ NP***

If P=NP, every computational problem can be solved in polynomial time.

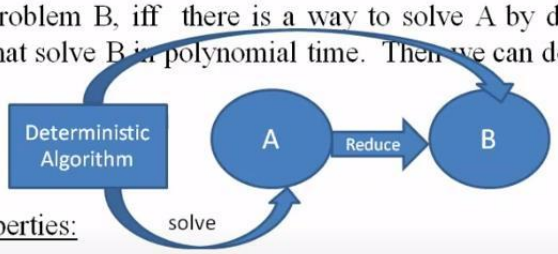
Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

If it turned out that P ≠ NP, which is widely believed, it would mean that there are problems in NP that are harder to compute than to verify: they could not be solved in polynomial time, but the answer could be verified in polynomial time

Polynomial time Reduction :

Reduction:

- Let A and B are two problems in NP. If problem A is reduce to problem B, iff there is a way to solve A by deterministic algorithm that solve B in polynomial time. Then we can denote AαB.



Properties:

- If A is reducible to B and B is in Polynomial time, then A also in Polynomial time.
- A is not in Polynomial time, it implies that B is not in Polynomial time.

A is polynomially reducible to B

$A \alpha B.$

$(A \leq_p B)$

Neethu Mathew , CSE Dept. EKCTC

❑ Let us consider a decision **problem A, which we would like to solve in polynomial time**. We call the input to a particular problem an instance of that problem. Now suppose that there is another decision **problem B that we already know how to solve in polynomial time.** Finally, **suppose that we have a procedure that transforms any instance α of A into some instance β of B** with the following characteristics:

(1) The transformation takes polynomial time.

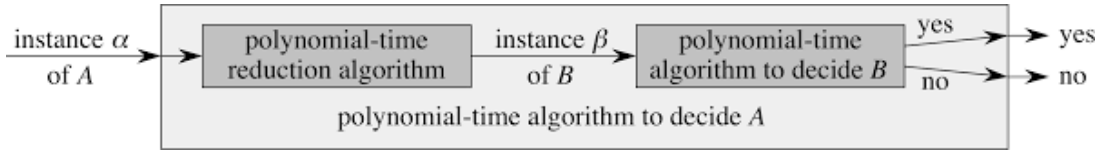
(2) The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."

We call such a procedure a **polynomial-time reduction** algorithm and, as the figure below shows, it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.

2. Run the polynomial-time decision algorithm for B on the instance β.

3. Use the answer for B as the answer for A.



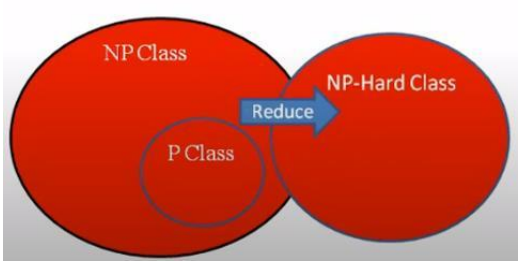
By "reducing" solving problem A to solving problem B, we use the "easiness" of B to prove the "easiness" of A.

Neethu Mathew , CSE Dept. EKCTC

NP-hard

What does NP-hard mean?

- A problem is **NP-hard** if all problems in NP are polynomial time reducible to it
- A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time.

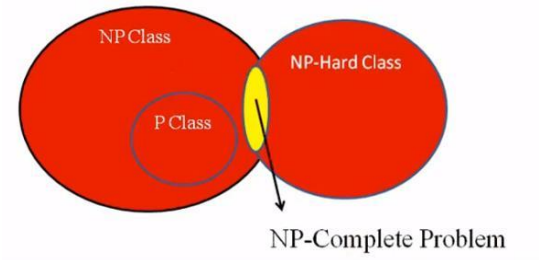


All NP problems can be transformed in polynomial time into it

NP-hard (Non-deterministic Polynomial-time hard), is a class of problems that are, informally, "at least as hard as the hardest problems in NP".

NP-complete (NPC)

- A problem is **NP-complete** if the problem is both
  - NP-hard, and
  - NP.



- "All NP-Complete problems are NP-Hard But not all NP-Hard problems are not NP-Complete."
- NP-Complete problems are subclass of NP-Hard

If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.

NP complete problems :

- 3 CNF Satisfiability
- Clique
- Vertex cover
- Subset sum
- Hamiltonian cycle
- Travelling salesman

etc

SAT problem (Boolean satisfiability problem)

- **Boolean satisfiability problem** abbreviated as SAT problem
- This problem determines if there exist a set of boolean variables which satisfy a boolean formula
- A Boolean variable has two possible values: True (1) or False (0).
- A literal is either a boolean variable x or its negation ¬x.
- A Boolean formula is composed of literals joined by operations AND (conjunction, ∧), OR (disjunction, ∨), and perhaps further negations.
- A boolean formula is in **conjunctive normal form (CNF)** if it consists of clauses joined by AND.
- A clause is a set of literals joined by OR.
- An assignment gives a truth value to every variable in a Boolean formula.
- An assignment is said to be satisfying if the formula evaluates to 1.
- A boolean formula is said to be satisfiable if a truth assignment that evaluate the formula to be 1 (true)

Consider a CNF(Conjunctive Normal Form)

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3)$$
$$C_1 \qquad C_2 \qquad C_3$$

3 Clauses – C_1,C_2,C_3

V denotes OR , Λ denotes AND

x	$\neg x$
0	1
1	0

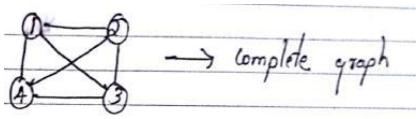
x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

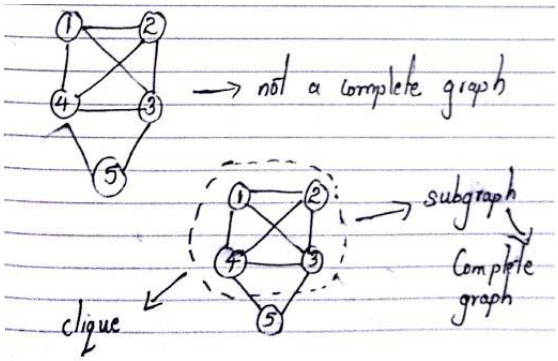
Clique

- A clique is a complete subgraph of a graph

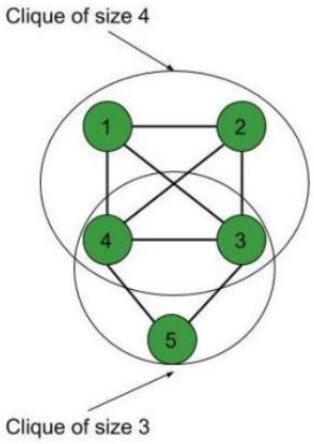
  (all the vertices in this subgraph are connected with each other . Subgraph is a complete graph.)



* One of the property of the complete graph is
number of vertices $|V| = n$ (4)
no. of edges $|E| = \frac{n(n-1)}{2}$ $\frac{4\times(4-1)}{2} = 6$



- **Clique size k**
- **k is number of vertices**



A graph of G =(V,E) is said to have a clique of size k , if there is $V' \subset V$ with $|V'|=k$ such that $\forall$ (u,v) $\in V'$ and edge (u,v) $\in E$

NP-completeness of Clique Problem

- Clique is NP complete

  To **prove that Clique is NP complete** , we have to show that

  -> Clique $\in$ NP

  -> Clique is NP-hard

1. Clique belongs to NP
   - ✓ Check if $V' \subset V$ with $|V'|=k$ vertices of G , check if each pair (u,v) in V' has an edge in G.
   - ✓ This verification is done in polynomial time

2. Clique is NP-hard

- In order to prove that this problem is NP-Hard then reduce a known problem (SAT Problem ie, Boolean satisfiability problem) to our problem
- Let our problem be L2. To prove L2 is NP Hard , select a problem L1 ,which is already known as NP Hard and show that

  $L1 \leq L2$ ie, $L1 \alpha L2$ => $SAT \leq_p$ CLIQUE

  ie, SAT problem(**Boolean satisfiability problem**) Reduces to Clique problem

Consider a CNF(Conjunctive Normal Form)

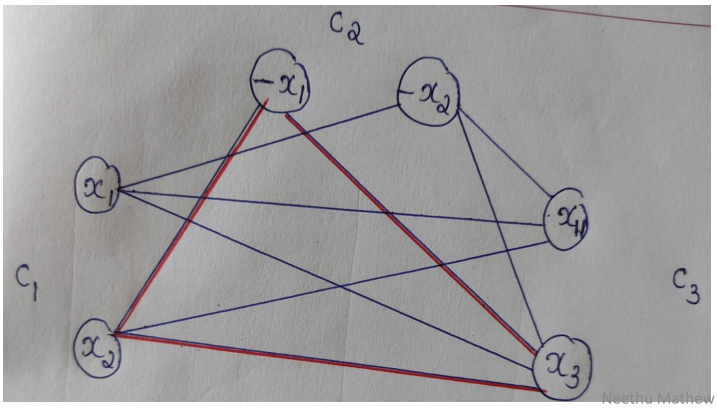
$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3)$$
$$C_1 \qquad C_2 \qquad C_3$$

3 Clauses – C_1, C_2, C_3

CNF: conjunction(AND) of one or more clauses, where a clause is a disjunction(OR) of literals

V denotes OR,
∧ denotes AND

- If the formula can be converted into a graph & if that graph present a clique , then we are able to reduce SAT problem to clique



✓ We connect the vertices such that –
- No 2 vertices belonging to the same clause are connected.
- No variable is connected to its complement(negation).

Clique $X_2, -X_1, X_3$
Clique size k is 3
No. of clauses and k is 3

satisfiability expression evaluates to true (1)

$x_1 \quad x_2 \quad x_3$

$0 \quad 1 \quad 1$

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

$(0 \vee 1) \wedge (1 \vee 0) \wedge (0 \vee 1)$

$1 \wedge 1 \wedge 1 = 1$

Clique $X_2, -X_1, X_3$
Is taken as true
Ie,
$X_1 = 0$
$X_2 = 1$
$X_3 = 1$

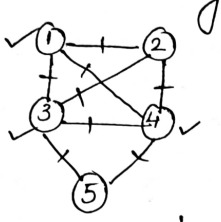
Boolean satisfiability problem ie , SAT $\leq_p$ CLIQUE
Clique is NP-Hard.

Clique belongs to NP and NP-Hard.

So Clique is NP complete

NP completeness of vertex cover problem

A vertex Cover of a graph is a set of vertices that touches every edges in the graph.



$G(V, E)$

$V = \{1, 2, 3, 4, 5\}$ set of vertices

Vertex cover of graph $G = \{1, 3, 4\}$

To Prove vertex cover is NP Complete;

1) vertex cover is in NP

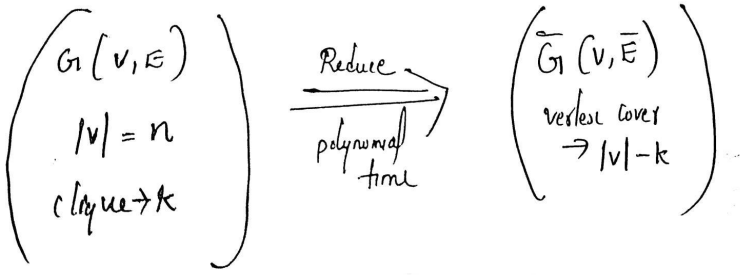
∵ Can be verifiable in polynomial time

2) vertex cover is NP Hard

- In order to prove that vertex cover problem is NP-Hard , reduce CLIQUE problem to our problem
- Let our vertex cover problem be L2. To prove L2 is NP Hard , select a problem L1 (clique) ,which is already known as NP Hard and show that

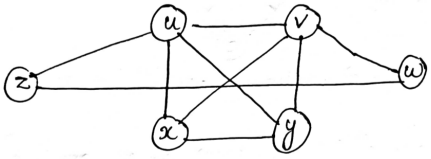
$L1 \leq L2$ ie, $L1 \alpha L2$ => CLIQUE $\leq_p$ Vertex Cover

ie, Clique problem Reduces to vertex cover



A graph $G(V, E)$ has a clique of size k iff the complement Graph $\bar{G}$ has a vertex cover of size $|V| - k$

Consider Graph $G_1 \Rightarrow$



$G(V, E)$

no. of vertices $|V| = 6$

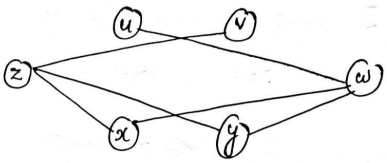
max clique of size $k = 4$

V : vertices in $G_1 \Rightarrow \{u, v, w, x, y, z\}$

V' : vertices in clique $\Rightarrow \{u, v, x, y\}$

Complement graph $\overline{G}$

(It consist of edges not in G_1)



Vertex cover of size $|V| - k$

$6 - 4 = 2$

Vertices in G_1 − vertices in clique

$\{u, v, x, y, z, w\} - \{u, v, x, y\}$

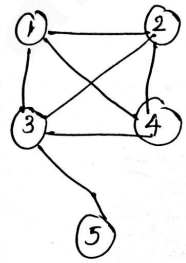
$\Rightarrow \{w, z\}$

w, z cover all the vertices in $\overline{G_1}$

Vertex cover is both NP and NP-Hard.

So Vertex cover problem is NP complete

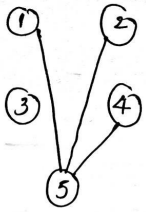
G_1



$|V| = 5$

$k \to 4$

$\overline{G_1}$



$|V| - k$

$5 - 4$

1

$\{1, 2, 3, 4, 5\} - \{1, 2, 3, 4\}$

$= \{5\}.$

(5) cover all edges in $\overline{G_1}$

Graph coloring

- Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color.
- This is also called the vertex coloring problem.
- If coloring is done using at most k colors, it is called k-coloring.
- The smallest number of colors required for coloring graph is called its chromatic number.
- Graph coloring problem is both, decision problem as well as an optimization problem.
    - A decision problem is stated as, “With given M colors and graph G, whether such color scheme is possible or not?”.
    - The optimization problem is stated as, “Given M colors and graph G, find the minimum number of colors required for graph coloring.”

The other **graph coloring problems** are:

- **Edge coloring** assigns a color to each edge so that no two adjacent edges share the same color
- **Face coloring** of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color

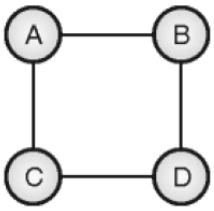
Applications of Graph Coloring Problem

- Design a timetable
- Sudoku
- Register allocation in the compiler
- Map coloring
- Mobile radio frequency assignment

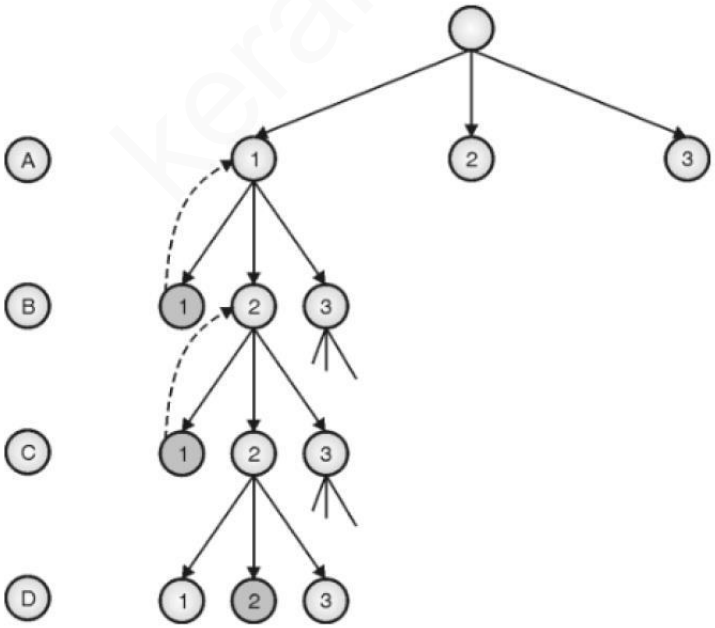
This problem can be solved using backtracking algorithms as follows:

- List down all the vertices and colors
- Assign color 1 to vertex 1
- If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.
- Repeat the process until all vertices are colored

Consider the graph



If we assign color 1 to vertex A, the same color cannot be assigned to vertex B or C. In the next step, B is assigned some different colors 2. Vertex A is already colored and vertex D is a neighbor of B, so D cannot be assigned color 2. The process goes on. State-space tree is shown in Figure below :-



Thus, vertices A and C will be colored with color 1, and vertices B and D will be colored with color 2.

Complexity Analysis

The number of anode increases exponentially at every level in state space tree. With M colors and n vertices, total number of nodes in state space tree would be $1 + M + M^2 + M^3 + \ldots. + M^n$

Hence, $T(n) = 1 + M + M^2 + M^3 + \ldots. + M^n$

So, $T(n) = O(M^n)$.

Thus, the graph coloring algorithm runs in exponential time.

Approximation algorithms

- An algorithm that returns near-optimal solutions is called an ***approximation algorithm***.
- The goal of the approximation algorithm is to come as close as possible to the optimal solution in polynomial time
- This technique does not guarantee the best solution
- Approximation algorithms are also called heuristic algorithm.

Approximation Ratio or performance ratio or ratio bound , *ρ(n)*

- If, for any input of size *n* for a problem, the cost *C* of the solution produced by an algorithm is within a factor of *ρ*(*n*) of the cost C^* of an optimal solution (where max $(C/C^*, C^*/C) \leq \rho(n)$), we say that,
  - the algorithm has an *approximation ratio* of *ρ*(*n*) or
  - the algorithm is a *ρ(n)-approximation algorithm.*

That is,

for any input of size *n* , *costs C and* C^* of approximate and optimal solutions : $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$

C^* - optimal solution

C - solution produced by an approximate algorithm

An algorithm that achieves an *approximation ratio* of *ρ*(*n*) is a *ρ(n)-approximation algorithm.*

Bin Packing Problem

- Given n items of different weights and bins each of capacity c
- Assign each item to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.
- Lower Bound

  The lower bound is the minimum number of bins required

  Min no. of bins >= Ceil ((Total Weight) / (Bin Capacity))

Example 1)

Weight of the items = {5,4,7,1,3}

bin capacity = 1

lower bound = minimum number of bins required = Ceil ((Total Weight) / (Bin Capacity))

= Ceil((5+4+7+1+3)/10) = Ceil (20/10) = 2

ie, 2 bins.

Algorithms for bin packing

- Next fit
- First fit
- Best fit
- Worst fit

online algorithms

- First fit decreasing
- Best fit decreasing

offline algorithms

1. Next fit :

- When processing next item, check if it fits in the same bin as the last item. Use a new bin only if it does not.

2. First Fit:

- When processing the next item, scan the previous bins in order and place the item in the first bin that fits. Start a new bin only if it does not fit in any of the existing bins

3. Best Fit:

- The idea is to places the next item in the tightest spot. That is, put it in the bin so that the smallest empty space is left.

4. First Fit Decreasing:

- We first sort the array of items in decreasing size by weight and apply first-fit algorithm

5. Best fit decreasing algorithm

- We first sort the array of items in decreasing size by weight and apply Best-fit algorithm

Example 2)

Assuming the sizes(weight) of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6} And bin capacity is 1

Ans)

lower bound =

The minimum number of bins required = Ceil ((Total Weight) / (Bin Capacity))

= Ceil(3.7/1) = Ceil (3.7) =4

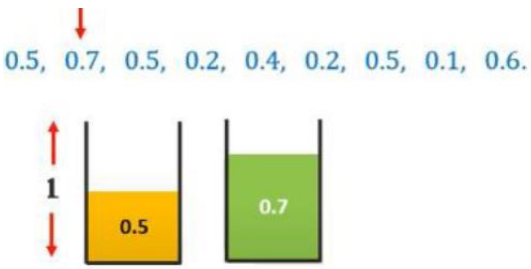
ie, 4 bins.

1. **Next fit** :
   - When processing next item, check if it fits in the same bin as the last item. Use a new bin only if it does not.

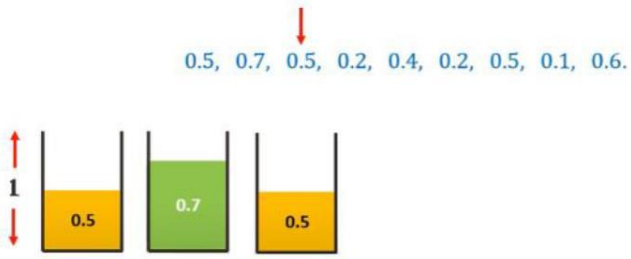
Consider bins of size 1

Considering 0.5 sized item first, we can place it in the first bin

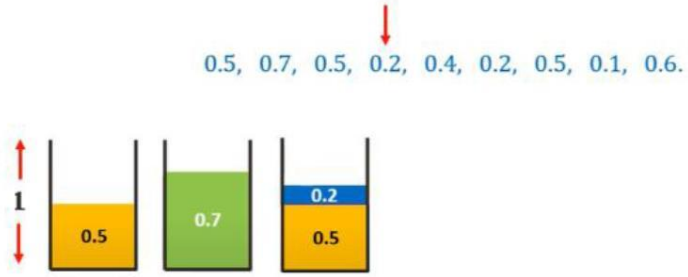
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin



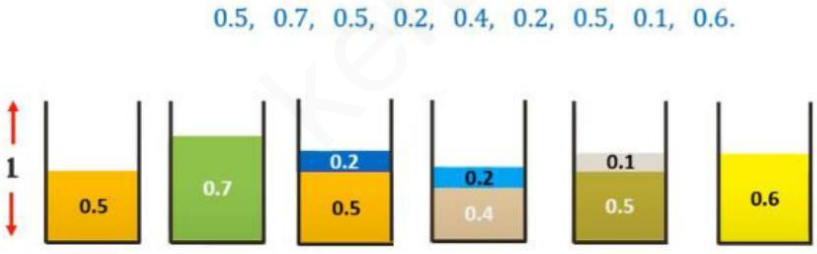
Moving on to the 0.5 sized item, we cannot place it in the current bin. Hence we place it in a new bin.



Moving on to the 0.2 sized item, we can place it in the current (third bin)



Similarly, placing all the other items following the Next-Fit algorithm we get-

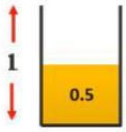


Thus we need 6 bins as opposed to the 4 bins of the optimal solution. Thus we can see that this algorithm is not very efficient.

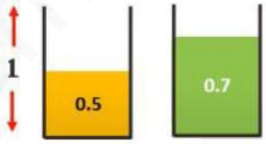
2. First Fit:

- When processing the next item, scan the previous bins in order and place the item in the first bin that fits. Start a new bin only if it does not fit in any of the existing bins

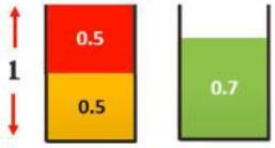
Considering 0.5 sized item first, we can place it in the first bin



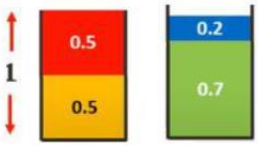
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



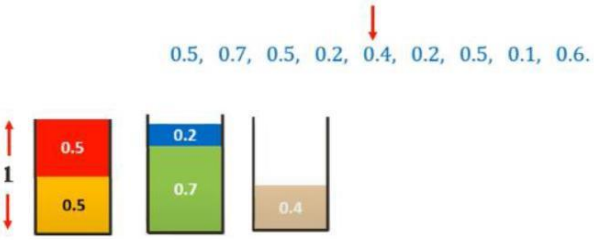
Moving on to the 0.5 sized item, we can place it in the first bin.



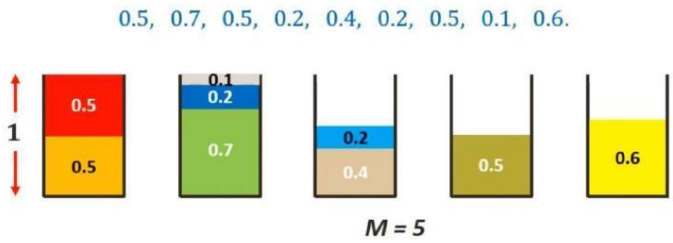
Moving on to the 0.2 sized item, we can place it in the first bin, we check with the second bin and we can place it there.



Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.



Similarly, placing all the other items following the First-Fit algorithm we get-



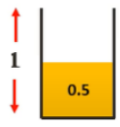
M = 5

Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

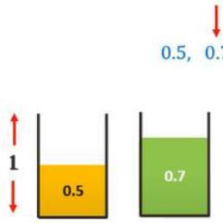
3. Best Fit:

- The idea is to places the next item in the tightest spot. That is, put it in the bin so that the smallest empty space is left.

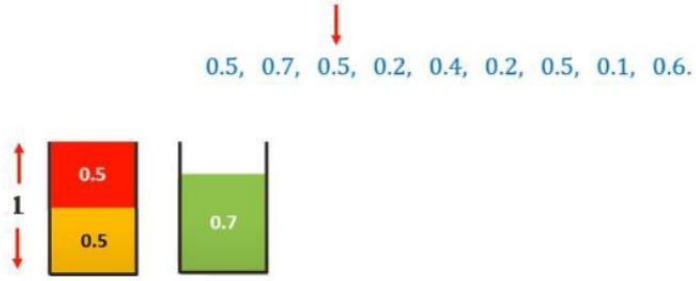
Considering 0.5 sized item first, we can place it in the first bin



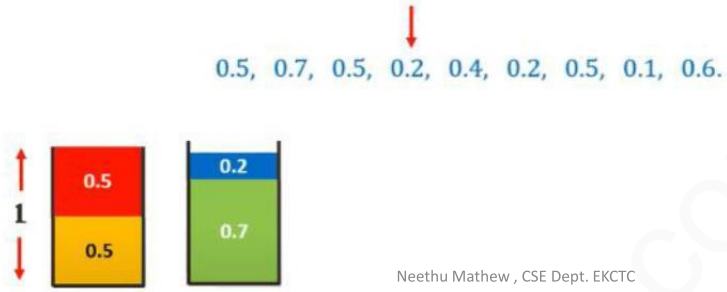
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



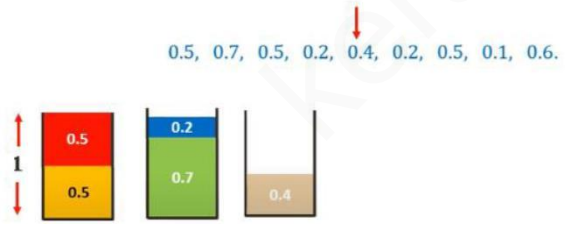
Moving on to the 0.5 sized item, we can place it in the first bin tightly.



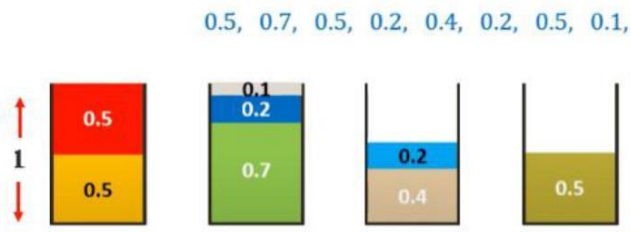
Moving on to the 0.2 sized item, we cannot place it in the first bin but we can place it in second bin tightly.



Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.



Similarly, placing all the other items following the Best-Fit algorithm we get-



Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

4. First Fit Decreasing:

- We first sort the array of items in decreasing size by weight and apply first-fit algorithm

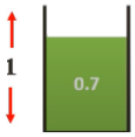
size of the items be **{0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}**

Sorting them we get **{0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1}**

The First fit Decreasing solution would be-

We will start with 0.7 and place it in the first bin

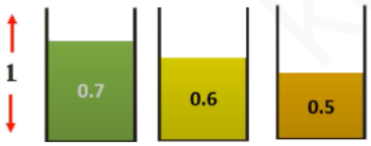
0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



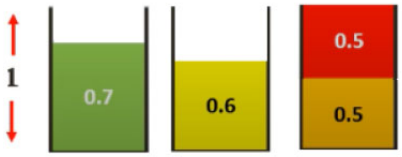
We then select 0.6 sized item. We cannot place it in bin 1. So, we place it in bin 2

We then select 0.5 sized item. We cannot place it in any existing. So, we place it in bin 3

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1

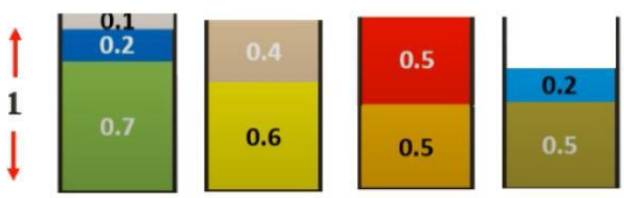


We then select 0.5 sized item. We can place it in bin 3



Doing the same for all items, we get

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Thus only 4 bins are required which is the same as the optimal solution.

5. Best fit decreasing algorithm

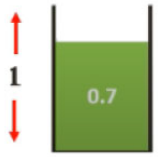
- We first sort the array of items in decreasing size by weight and apply Best-fit algorithm

Weight of the items be **{0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}**

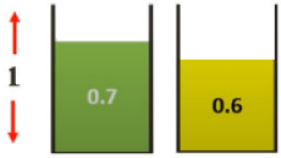
After Sorting **{0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1}**

We will start with 0.7 and place it in the first bin

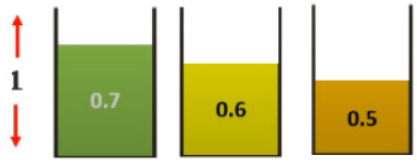
0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



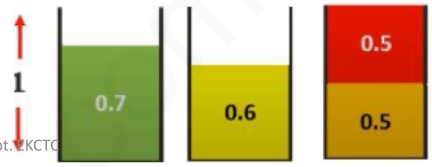
We then select 0.6 sized item.
We cannot place it in bin 1.
So, we place it in bin 2



We then select 0.5 sized item. We cannot place it in any existing. So, we place it in bin 3

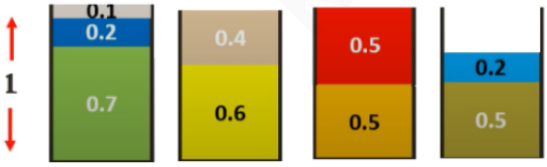


We then select 0.5 sized item.
We can place it in bin 3



Doing the same for all items, we get.

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Thus only 4 bins are required which is the same as the optimal solution.

Randomized Algorithms

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits, and it is then allowed to use these random bits to influence its computation. An algorithm is randomized if its output is determined by the input as well as the values produced by a random-number generator.

A randomized algorithm is one that makes use of a randomizer such as a random number generator. Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run for the same input. So, the execution time of a randomized algorithm could also vary from run to run for the same input.

Randomized algorithms can be categorized into *two* classes :

1. Las Vegas algorithms
2. Monte Carlo algorithms

Las Vegas algorithm always produces the same (correct) output for the same input. The execution time of a Las Vegas algorithm depends on the output of the randomizer. If we are lucky, the algorithm might terminate fast, and if not, it might run for a longer period of time. In general, its runtime for each input is a random variable whose expectation is bounded.

The second is algorithms whose output might differ from run to run for the same input. These are called **Monte Carlo algorithms.** Consider any problem for which there are only two possible answers, say yes and no. If a Monte Carlo algorithm is used to solve such a problem, then the algorithm might give incorrect answers depending on the output of the randomizer. We require that the probability of an incorrect answer from a Monte Carlo algorithm be low.

Typically, for a fixed input, a Monte Carlo algorithm does not display much variation in execution time between runs, whereas in the case of a Las Vegas algorithm this variation is significant.

40.2 Applications

40.2.1 Randomized Quick Sort Algorithm

- In randomized quick sort, we will pick randomly an element as the pivot for partitioning.
- The expected runtime of any input is $O(n \log n)$.

Analysis of Randomized Quick Sort

- Let $s(i)$ be the ith smallest in the input list S.
- X_{ij} is a random variable such that $X_{ij} = 1$ if $s(i)$ is compared with $s(j)$; $X_{ij} = 0$ otherwise.

- Expected runtime t of randomized QS is :

$$t = E\left[\sum_{i=1}^{n} \sum_{j \geq 1} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j \geq 1} E[X_{ij}]$$

- $E[X_{ij}]$ is the expected value of X_{ij} over the set of all random choices of the pivots, which is equal to the probability p_{ij} that $s(i)$ will be compared with $s(j)$.

The randomized quick sort algorithm uses $i \leftarrow$ Random (p, r) to be the new pivot element.

Randomized-Partition (A, p, r)

1. $i \leftarrow$ Random (p, r)
2. exchange $A[r] \leftrightarrow A[i]$
3. return Partition (A, p, r)

Randomized-Quicksort (A, p, r)

1. If $p < r$
2. then $q \leftarrow$ Randomized-Partition (A, p, r)
3. Randomized-Quicksort $(A, p, q-1)$
4. Randomized-Quicksort $(A, q+1, r)$