Unit-5

NP-Hard and Np-complete problems

Basic concepts:

Deterministic algorithms:

- The algorithms in which every operation is uniquely defined is called deterministic algorithms.
- The working of algorithms is known.

Non Deterministic algorithms:

- The algorithms in which the operations are not uniquely defined but are limited to specific set of possibilities for every operation, such an algorithms is called non-deterministic algorithms.
- The non-deterministic algorithms use the following functions:
  1. choice: Arbitrarily choose one element from given set.
  2. failure: Indicates an unsuccessful completion.
  3. success: Indicates a successful completion.

A non deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.

whenever, there is a set of choices that leads to a successful completion then one such set of choices is selected and the algorithm terminates successfully.

Polynomial time & Exponential time:

1. polynomial time:

- An algorithms 'A' is of polynomial complexity if there exists a polynomial p() such that computing time is $O(p(n))$ for every input size 'n'.

$$P(n) = (n^k) \text{ for some non negative integer.}$$

- The computing time of polynomial time Algorithms are less.

Ex: Linear search $O(n)$

Binary search $O(\log n)$

Insertion sort $O(n^2)$

Merge sort $O(n \log n)$

2. Exponential time / Non-polynomial time algorithms:

- An algorithm 'A' is of exponential complexity if the complexity is of the form $O(2^n)$.
- computing times of non-polynomial algorithms are greater than polynomial algorithms.

Ex: 0/1 Knapsack $-O(2^n)$

Travelling salesman $-O(2^n)$

Graph coloring $-O(2^n)$

sudoko $-O(2^n)$

P class problems / class P:

- It consists of all set of decision problems solvable by using deterministic algorithms in polynomial time.

Ex: Algorithms for searching, sorting, addition, multiplication etc.

NP class problems / class NP:

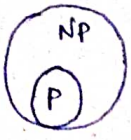
- It consists of set of decision problems solvable
- The problems that are solvable in polynomial time is called tractable problems.
- All deterministic polynomial time algorithms are tractable.

NP class problems / class NP:

- It consists of set of decision problems solvable by using Non deterministic algorithms in polynomial time (exponential time).
- The problems that require non-polynomial time are called non-tractable problems.
- All non-deterministic polynomial time algorithms are intractable.

Ex: Travelling salesmans problem, sudoko etc.

- class P is a subset of class NP.



- Any problems that can be solved by using deterministic algorithms in polynomial time can also be solved by using Non deterministic algorithms in polynomial time.

Satisfiability problems:

- The satisfiability is a boolean formula that can be constructed using the following literals and operations.

1. A Literal is either a variable or its Negation of the variable.
2. The Literals are connected with operators $\vee, \wedge, \Rightarrow, \Leftrightarrow$
3. parenthisis

- The satisfiability problems (SAT) is to determine whether a Boolean formula is true for some assignment of truth values to the variables.

In general, formulas are expressed in conjunctive Normal form (CNF)

- A Boolean formula is in CNF iff it is represented by

$$(x_1 \vee x_j \vee x_k') \wedge (x_i \vee x_j' \vee x_k)$$

- A Boolean formula is in 3CNF if each clause has exactly 3 distinct Literals.

Ex: The non-deterministic algorithm that terminates successfully iff a given formula $E(x_1, x_2, x_3)$ is satisfiable.

Reducability:

Definition: Let L_1 & L_2 be two problems.

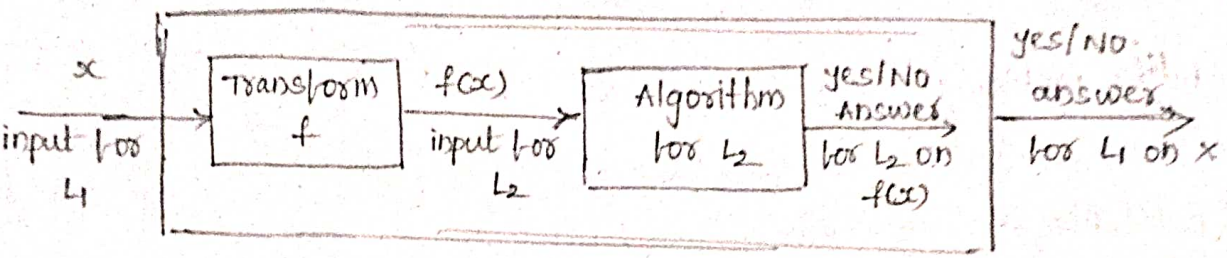
problem L_1 is reducable to L_2 if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.

Reducability between L_1 & L_2 is represented as

$$L_1 \propto L_2$$

Ex: Let L_1 & L_2 be two decision problems. suppose algorithm A_2 solves L_2. ie, if y is an output for L_2, then algorithm A_2 will answer yes or no depending upon whether 'y' belongs to L_2 or not.

The idea is to find a transformation from L_1 to L_2 so that algorithm A_2 can be part of an algorithm A_1 to solve L_1.

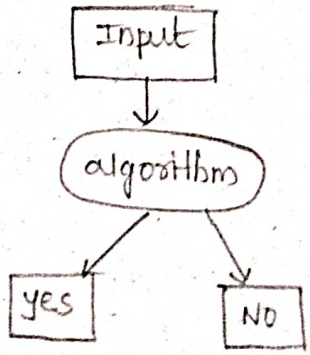


Here we dont design algorithm for L_1 we just transform the input of L_1 to a algorithm L_2 which gives an output yes/no.

Decision problems:

- Any problem for which the answer is either yes or NO is called decision problem.

Ex: Deciding whether a given natural number is prime;



optimization problem:

- optimization problem is finding an optimal value (minimum or maximum) among set of all possible values.

Ex: 0/1 Knapsack, Travelling salesman problem.

NP- Hard and NP- complete:

the Non deterministic polynomial time (NP) problems can be classified into 2 classes. They are

1. NP Hard
2. NP complete

1. Np Hard :

A problem x is Np-Hard if **any** problem $y \in Np$ reduces to x.

Ex: Halting problems, flow shop scheduling problems.

Halting problem – Deciding if a program will stop or run forever.

Flow shop scheduling problems – figuring out the best way to schedule tasks on machines to save time.

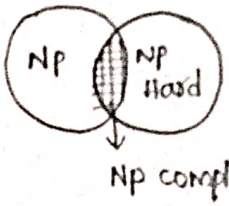
2. Npcomplete :

steps to show that a problem is Np-hard :

1. pick a problem L_1 already known to be Np-hard.
2. show that L_1 is reducible to L_2.
3. Thus L_2 is Np-hard.

2. NP complete :

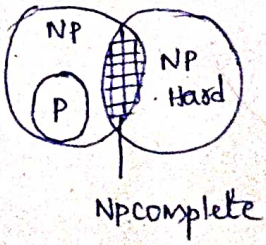
A problem 'L' is Np-complete if L is Np-Hard and L-belongs to Np.



Ex: Boolean satisfiability problem, knapsack problem.

Relationship between P, NP, Np-hard, Np- complete :

Let P, NP, Np-hard, Np-complete are the sets of all possible decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic algorithms, Np-Hard and Np-complete respectively. Then the relationship between P, NP, Np-Hard, Np-complete can be expressed using venn diagrams as



Cook's Theorem / Cook-Levin Theorem:

The scientist stephen cook in 1971 stated that Boolean satisfiability problem is NP-complete.

This means if we find a way of solving SAT in polynomial time, we will be there is a position to solve any NP problem in polynomial time.

- Instance: Given a set of variables 'V', and a collection of clauses 'C' over 'V'
- Question: Is there a truth assignment for V that satisfies all clauses in C?
- CNF-satisfibility (CONJUNCTIVE NORMAL FORM)

  * Because the expression is in product of sums.

Example:

"$\neg x_i$" = "Not x_i", "or" = "Logical or", "AND" = "Logical and", $V = \{x_1, x_2\}$

$C_1 = \{(x_1, \neg x_2), (\neg x_1, x_2)\}$

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

= if $x_1 = x_2 =$ True $\Rightarrow C_1 =$ true (satisfiable).

$C_2 = \{(x_1, x_2)\ (x_1, \neg x_2)\ (\neg x_1)\}$

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

= if $x_1 = x_2 =$ True $\Rightarrow C_2 =$ Not satisfiable.

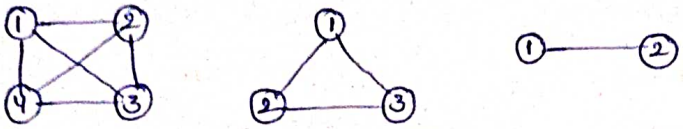
NP Hard Graph problems

clique Decision problem (CDP):

A complete graph is a graph in which every vertex is connected with every other vertices in the graph.

- the property of complete graph is if number of vertices are 'n' then the number of edges in a complete graph is $\frac{n(n-1)}{2}$

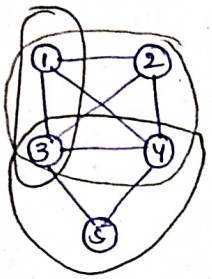
Ex:



clique decision problem (CDP):

- clique is a subgraph of any graph which is complete.
- The size of clique is the number of vertices in it.

Ex:



→ 1-2-3-4 is a clique of size 4

→ 3→4-5 is a clique of size 3

→ 1-3 is a clique of size 2.

problem: TO prove clique Decision problems is Np-hard.

sol: steps to prove any problem is Np-hard:

1. pick a problem L_1, already known as Np-hard.
2. show that $L_1 \propto L_2$
3. thus L_2 is Np-hard.

- TO prove that a clique decision problem is Np-hard, pick CNF satisfibility problem that is known as Np-hard problem.
- show that CNF satisfiability $\propto$ CDP

consider a propositional formula in CNF

$$f = \bigwedge_{1 \le i \le k} C_i$$ ← CNF multiple clauses are connected through ∧ operation

$$f = \underbrace{(x_1 \vee x_2 \vee x_3)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{C_2}$$

Here x_1, x_2, x_3 are literals connected through 'or' operator in a clause.

we have to create a graph where vertices of graph are Literals

$V = \{ \langle \sigma, i \rangle \mid \sigma \text{ is a literal in clause } i \}$

$E = \{ \langle \sigma, i \rangle, \langle \delta, j \rangle \mid i \neq j \text{ and } \sigma \neq \bar{\delta} \}$

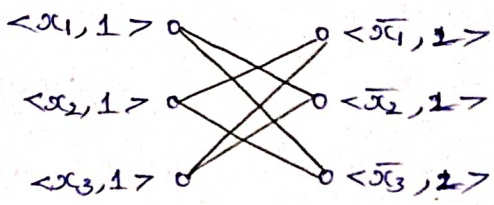
↓ Literal of 1st clause ↑ Literal of 2nd clause

↓ we don't make an edge from same clause & if the literal is in 1st clause it shouldn't be equal to the Negation of literal of 2nd clause

Ex: we can't combine $x_2, \bar{x}_2$ bcz $x_2, \bar{x}_2 = x_2, x_2$

vertices are

clause 1	clause 2
$\langle x_1, 1 \rangle$	$\langle \bar{x}_1, 2 \rangle$
$\langle x_2, 1 \rangle$	$\langle \bar{x}_2, 2 \rangle$
$\langle x_3, 1 \rangle$	$\langle \bar{x}_3, 2 \rangle$



consider a clique with vertices $\{ \langle x_1, 1 \rangle \langle \bar{x}_2, 2 \rangle \}$

By setting $x_1 = \text{true}$ $\bar{x}_2 = \text{true}$ $x_3 = \text{True/false}$

$x_2 = \text{false}$

$F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$= (T \vee F \vee T) \wedge (F \vee T \vee F)$

$= T \wedge T$

$= T$

∴ clique decision problem is Np Hard.

chromatic Number Decision problem (CNDP):

Graph coloring:

Graph coloring is the process of assigning colors to the vertices of a graph such that no two adjacent vertices have same color.

chromatic Number:

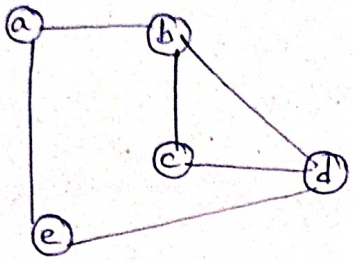
The chromatic number of a graph is the minimum number of colors needed to color the graph properly.

Chromatic Number Decision problem:

Given a graph and a number (k), can you color the graph with (k) or less colors?

In otherwords, is it possible to color the graph using (k) colors so that no two connected vertices have same color?

Ex: consider the following graph with 5 vertices



problem statement:

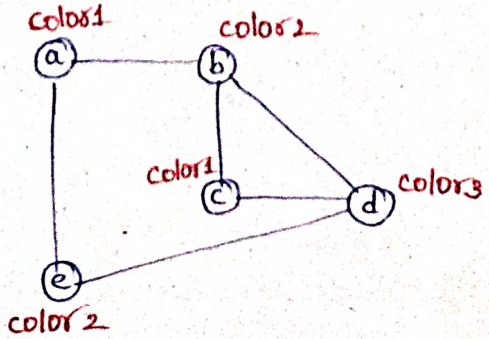
can we color this graph with 3 colors or fewer?

step1: Try to color the graph with 3 colors

we need to assign colors to the vertices such that no two connected vertices get the same color.

step2: color the graph

- Let's assign color1 to vertex 'a'.
- vertex 'b' is adjacent to 'a', so it can't have color1. so give color2 to vertex 'b'.
- vertex 'c' is adjacent to 'b' so it can't have color2. we give it color1.
- vertex 'd' is adjacent to 'b' and 'c' so we can't give color1 and color2 to 'd', we assign color3 to vertex 'd'.
- vertex 'e' is adjacent to 'a', 'd', so we can't give color1, color3 to vertex 'e'. we assign color2 to vertex 'e'.



Step3: check if the Graph is properly colored.

After coloring the graph, we have,

A = color1

B = color2

C = color 1

D = color3

E = color2

No two connected vertices have the same color, so this is a valid coloring with 3 colors.

step4: since we successfully colored the graph with 3 colors, the answer to the CDNP for (K=3) is yes.

Why CDNP is NP-Hard problem?

Verification: If someone gives you a colored graph, its easy to check if the coloring is just correct (just check if adjacent vertices have different colors). This can be done in polynomial time.

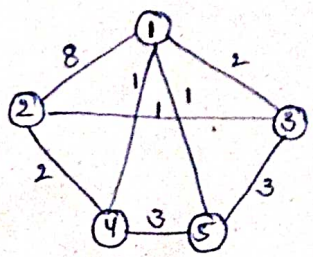
Hardness: However, finding the chromatic number (the minimum number of colors) is difficult because you have to try different combination of color assignments, especially for Large graphs.

Since we don't know of any efficient way to solve CNDP for Large graphs, it is classified as NP-Hard.

Travelling salesperson Decision problem (TSDP):

Given a set of cities and distance between every city are given, the problem is to find the shortest path that visit every city exactly once and return back.

Ex:



shortest path = 5 → 1 → 4 → 3 → 2 → 5 (Hamiltonian path with minimum weight)

Theorem: Travelling salesperson decision problem is NP-Hard.

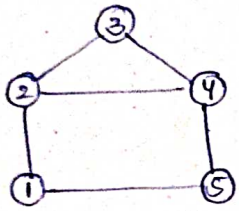
Proof:

Inorder to prove Travelling salesperson problem is in NP hard, we need to take a problem in NP-hard that is already known and reduce that known NP-hard problem to TSP.

We consider Hamiltonian cycle (NP-Hard) problem for which we know the solution and reduce it to TSP.

ie, Hamiltonian cycle $\propto$ TSP

step1: consider the graph G(V,E)



Hamiltonian path = $1 \to 2 \to 3 \to 4 \to 5 \to 1$

> Hamiltonian cycle or circuit in a graph G is a cycle that visits every vertex of G exactly once and returns to the starting vertex.

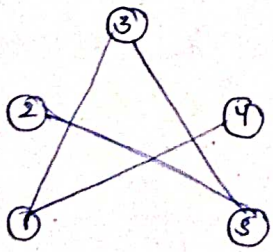
Here we try to take an i/p to Hamiltonian cycle problem and we try to prove that the input is similar to the TSP input.

step2: we have to construct TSP from above graph.

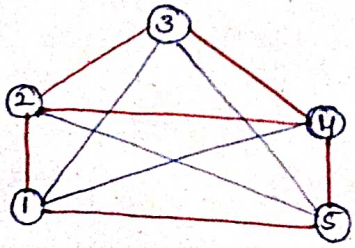
a) construct a complete graph for the considered graph.

construct G' =(V,E') (G' is Negation of G which has same vertices but the edges we have to consider which are not present in G)

G' =



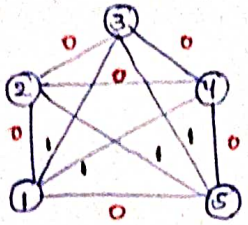
combine G' with G ie,



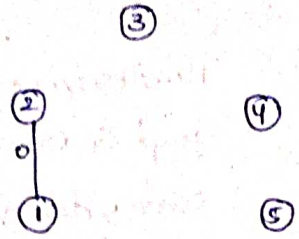
$\Rightarrow$ This is a complete graph.

b) Define the cost function

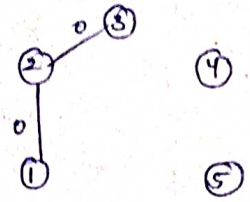
c(i,j) = 0 if i,j ∈ E (edge belongs to 'G' graph)
= 1 if i,j ∈ E' (edge belongs to 'G' graph)



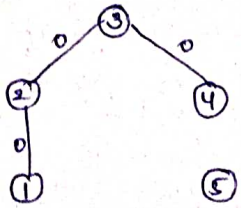
Now, choose Minimum cost edge adjacent to 1 (choose 2)



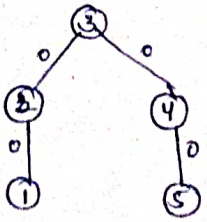
Now choose minimum cost edge adjacent to 2 (choose 3)



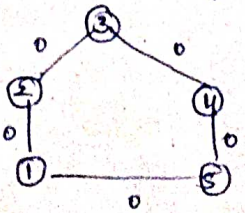
Now choose minimum cost edge adjacent to 3 (choose 4)



Now choose minimum cost edge adjacent to 4 (choose 5)



Now choose minimum cost edge adjacent to 5 (choose 1)



path

1 → 2 → 3 → 4 → 5 → 1

We got the same path as original graph. so TSP = Np Hard.

Scheduling Identical processors:

The scheduling problem for identical processors is about dividing tasks between processors (or machines) so that each one has a similar amount of work. We want to finish all tasks as quickly as possible, which means making balancing workload across processors so that no single processor has to run much longer than others.

problem description:

- We have a list of tasks, each with a certain amount of work (or time) needed to finish.
- We have identical processors that can each handle any task.
- The goal is to spread the tasks among the processors so that they all finish around the same time, minimizing the makespan (the time it takes for the processor with the most work to finish).

Example:

Given 4 tasks with times 5, 2, 3, and 7 units.

2 processors to Handle the tasks.

Different ways of assigning the tasks to processors gives different results.

1. Assign tasks 5, 2 to processor 1 & tasks 3, 7 to processor 2.

   processor 1 — total time = 7

   processor 2 → total time = 10

   Makespan = 10

2. Assign tasks 5, 7 to processor 1 & tasks 2, 3 to processor 3

   processor 1 → total time = 12

   processor 2 → total time = 5

   Makespan = 12 (processor 1 takes Longest)

3. Assign task 2, 7 to processor 1 and tasks 5, 3 to processor 2

   processor 1 → total time = 9

   processor 2 → total time = 8

   Makespan = 9 (shortest so far)

Here the 3rd option gives the best balance, with the shortest makespan of 9 units.

Why scheduling Identical processors task is Np Hard?

The problem is Np-Hard because,

Verification: If someone gives you a schedule, it's easy to check how long each processor takes to finish and verify the total processing time.

Hardness: finding the best (optimal) schedule where the makespan is minimized is hard because there are many possible ways to assign the tasks, especially when there are many tasks and processors.

There is no known efficient (polynomial time) algorithm to find the optimal solution, which makes this problem Np-Hard.

As the reason this scheduling problems is Np-Hard is that there are many ways to assign the tasks to the processors. As the number of tasks increases, the number of possible assignments grows exponentially, making it very difficult to find the optimal solution in a reasonable amount of time for Large instances.

Applications of scheduling problem for Identical processors:

This problem is important in many real-world scenarios such as,

1. Manufacturing – scheduling jobs on machines to minimize production time.

2. computer systems – Allocating tasks to processors in a multi-core systems to optimize performance.

3. Logistics – Distributing workloads to workers or vehicles in a balanced way to minimize total operation time.

Job shop scheduling:

In Job scheduling problem, you are given:

Jobs – Each job is made up of sequence of tasks.

Machines – Each task must be processed on a specific machine, and different jobs may need the same machine.

The aim is to schedule the tasks so that the jobs are completed as quickly as possible while minimizing the makespan (the total time to finish all jobs.

The challenge is to decide the order in which to process the tasks on each machine, ensuring no machine works on more than one task at a time, and each job follows its required sequence of tasks.

Example:

problem set up:

- 2 Jobs : Job 1, Job 2
- 3 Machines : Machine A, Machine B, Machine C
- Each job has tasks that must be done in a specific order on certain machines.

Task Details:

Job 1:

- Task 1.1 : process on Machine A for 2 units of time
  Task 1.2 : process on Machine B for 3 units of time
  Task 1.3 : process on Machine C for 2 units of time

Job 2:

- Task 2.1 : process on Machine B for 4 units of time
  Task 2.2 : process on Machine A for 1 unit of time.
  Task 2.3 : process on Machine C for 3 units of time.

problem statement:

We need to schedule the tasks on the machines in such a way that all tasks are completed as soon as possible, minimizing the makespan.

4 steps to solve the example.

step 1: We can represent the order of tasks like this

Job 1: Task 1.1 must be completed before Task 1.2, and task 1.2 must be completed before Task 1.3

Job 2: Task 2.1 must be completed before Task 2.2, and task 2.2 must be completed before Task 2.3

step2: Assign tasks to machines.

Let's start assigning tasks to the machines, keeping in mind that no two tasks can be processed on the same machine at the same time.

Machine A:

first, we can process Task 1.1 (Job 1) for 2 units of time.

After that, we can process Task 2.2 (Job 2) for 1 unit of time

Machine B:

we can process Task 2.1 (Job 2) for 4 units of time first.

Then, process Task 1.2 (Job 1) for 3 units of time.

Machine C:

once Task 1.2 is finished, we can process Task 1.3 (Job 1) for 2 units of time.

After Task 2.2 is completed, we can process Task 2.3 (Job 2) for 3 units of time.

step3: Calculate the Makespan

Let's look at the schedule on each machine:

Machine A:

Task 1.1 from time 0 to 2

Task 2.2 from time 4 to 3.

Machine B:

Task 2.1 from time 0 to 4

Task 1.2 from time 4 to 7

Machine C:

Task 1.3 from time 7 to 9

Task 2.3 from time 3 to 6

The makespan is determined by the longest time any machine is active. In this case, Machine 'C' finishes last at time 9 units, so the total makespan is 9 units of time.

<u>Why Job shop scheduling is NP-Hard?</u>

Verification: Given a schedule, it's easy to check the total makespan (How long all jobs take to finish) and verify if the schedule is valid.

Hardness: finding the optimal schedule is very difficult. As the number of jobs and machines increases, the number of possible schedule grows exponentially, making it computationally expensive to find the optimal solution.

Because of this complexity, job shop scheduling is classified as a NP-hard problem.