UNIT V

NP Hard and NP complete Problems:

In this unit we are concerned with the distinction between problems that can be solved by a polynomial time algorithm and problems for which no polynomial time algorithm is known.

For many of the problems we know and study the best algorithms for their solutions have computing times that cluster into two groups:

a) Problems whose solution times are bounded by polynomials of small degree.

Examples: Ordered Searching : $O(\log n)$
Polynomial evaluation : $O(n)$
Sorting : $O(n \log n)$
String editing : $O(mn)$

b) Problems whose best-known algorithms are nonpolynomial.

Examples: Traveling Salesperson : $O(n^2 2^n)$
Knapsack problem : $O(2^{n/2})$

Nondeterministic Algorithms:

Deterministic algorithms are those with the property that the result of every operation is uniquely defined. Such algorithms agree with the way programs are executed on a computer.

In a theoretical framework we can remove this restriction on the outcome of every operation.

We can allow algorithms to contain operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities. The machine executing such operations is allowed to choose any no. of these outcomes subject to a termination condition to be defined later. This leads to a concept of a "non-deterministic" algorithm. To specify such algorithms, we use three new functions:

1) choice(s): arbitrarily chooses one of the elements of set S.
2) Failure(): Signals an unsuccessful completion.
3) Success(): Signals a successful completion.

x := choice(1, n) could result in x being assigned any one of the integers in the range [1, n]. No rule to specify how the choice is made.

The Failure() and Success() signals are used to define a computation of the algorithm. These statements cannot be used to effect a return. Whenever there is a set of choices that leads to a successful completion, then one such set of choices is always made and the algorithm terminates successfully

"A non-deterministic algorithm terminates unsuccessfully iff there exists no set of choices leading to a success() signal."

The computing times for choice, Success and failure are taken to be O(1).

Examples of non-deterministic algorithms:

i) Searching an element x in a given set of elements $A[1:n]$, $n \geq 1$, we are required to determine and index j such that $A[j] = x$ or $j = 0$ if $x \notin A$.

```
1. j := choice(1, n);
2. if A[j] = x then { write(j); Success(); }
3. write(0); Failure();
```

From the definition of a non-deterministic computation. The number 0 can be output if there is no j such that $A[j] = x$.

Note: A is not ordered.

ii) Knapsack decision problem:

A non-deterministic polynomial time algorithm for the knapsack decision problem.

```
Algorithm DKP(P, w, n, m, r, x)
{
    w := 0; p := 0;
    for i := 1 to n do
    {
        x[i] := choice(0, 1);
        w := w + x[i] * w[i];
        p := p + x[i] * p[i];
    }
    if ((w > m) or (p < r)) then Failure();
    else Success();
}
```

A successful termination is possible iff the answer to the decision problem is yes. Time complexity = $O(n)$.

The time required by a non-deterministic algorithm performing on any given input is the minimum no. of steps needed to reach a succesful completion if there exists a sequence of choices leading to such a completion. In case succesful completion is not possible, then the time required is O(1).

A nondeterministic algorithm is of complexity O(f(n)) if for all inputs of size n, $n \geq n_0$, that result in a succesful completion, the time required is at most $cf(n)$ for some constants c and n_0.

Decision problems: Any problem for which the answer is either zero or one is called a decision problem. An algorithm for a decision problem is termed a decision algorithm.

Optimization problems: Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as optimization problem. An optimization algorithm is used to solve an optimization problem.

ii) The non-deterministic sort algorithm:

```
Algorithm NSort(A,n)
{
  for i:=1 to n do B[i]:=0; // initialize B[]
  for i:=1 to n do
  {
    j:= choice(1,n);
    if B[j]≠0 then Failure();
    B[j]:=A[i];
  }
  for i:=1 to n-1 do // verify order
    if B[i] > B[i+1] then Failure();
  write(B[1:n]); Success();
}
```

iv) Maximum Clique:

A maximal complete subgraph of a graph $G = (V, E)$ is a clique. The size of the clique is the no. of vertices in it.

The max. clique problem is an optimization problem that has to determine the size of a largest clique in G.

The corresponding decision problem is to determine whether G has a clique of size at least k for some given k.

Note: Many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem can.

Also, if the decision problem cannot be solved in polynomial time, then the optimization problem cannot either.

The algorithm DCK is a non-deterministic algorithm for the clique decision problem. The algorithm begins by trying to form a set of k distinct vertices. Then it tests to see whether these vertices form a complete subgraph. If G is given by its adjacency matrix and $|V| = n$, the input length m is $n^2 + \log_2 k + \log_2 n + 2$

Note: Because the input to the max. clique decision problem can be provided as a sequence of edges and an integer k. Each edge in E(G) is a pair of numbers (i, j). The size of the input for each edge (i, j) is

$\log_2 i + \log_2 j + 2$ if a binary representation is assumed.

The input size of any instance is

$$n = \sum_{\substack{(i,j) \in G(h) \\ i<j}} (\log_2 i + \log_2 j + 2) + \log_2 k + 1$$

There is no known polynomial time deterministic alg. for this problem.

The non-deterministic algorithm is as follows:

```
Algorithm Dck(G, n, k)
{
  S := 0 ; // S is an initially empty set
  for i := 1 to k do
  {
     t := choice(1, n);
     if t ∈ S then failure();
     S := S ∪ {t} // Add t to set S.
  }
// At this point S contains k distinct vertex indices
  for all pairs (i, j) such that i ∈ S, j ∈ S, i ≠ j do
  { if (i, j) is not an edge of G then
     failure();
  }
  success();
}
```

Satisfiability Problem

Let $x_1, x_2 \ldots$ denote boolean variables (their value is either true or false).

Let $\bar{x}_i$ denote the negation of x_i. A literal is either a variable or its negation.

A formula in the propositional calculus is an expression that can be constructed using literals and the operations '$\wedge$' (and) & '$\vee$' (or).

Examples of such formulas are:

1) $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ — DNF

2) $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ — CNF

A formula is in conjunction normal formal (CNF) iff it is represented as: $\bigwedge_{i=1}^{k} C_i$, where C_i are clauses, each represented as $\bigvee_{j} l_{ij}$, l_{ij} are literals.

It is in disjunction normal form (DNF) iff it is represented as $\bigvee_{i=1}^{k} C_i$ and each ~~clause~~ clause is represented as $\bigwedge_j l_{ij}$.

The satisfiability problem is to determine whether a formula is true for some assignment of truth values of to the variables.

CNF - satisfiability is the satisfiability problem for CNF formula.

It is easy to obtain a polynomial time non-deterministic algorithm that terminates successfully iff a given propositional formula $E(x_1, x_2 \ldots x_n)$ is satisfiable.

Such an algorithm could proceed by simply choosing (non-deterministically) one of the 2^n possible assignments of truth values to $(x_1 \ldots x_n)$ and verifying that $E(x_1, x_2 \ldots x_n)$ is true for that assignment.

```
Algorithm: Eval (E, n)
// Determine if formula E is satisfiable, for
// variables x1, x2 ... xn
{
   for i:= 1 to n do
      xi = choice (false, true);
   if E(x1, ... xn) then Success ();
   else Failure ();
}
```

The classes P, NP, NP-Hard and NP complete :-

In measuring the complexity of an algorithm, we use the input length as the parameter. An algorithm A is of polynomial complexity if there exists a polynomial $p()$ such that the computing time of $A = O(p(n))$ for every input of size n.

Class P is the set of all decision problems solvable by deterministic algorithms in polynomial time.

Class NP is the set all decision problem solvable by non-deterministic algorithms in polynomial time. i.e. the problems can be solved in polynomial time but have a polynomial-time checking algorithm i.e. given a solution we can check in a polynomial time if that solution is what we are looking for.

Since deterministic algorithms are just a special case of non-deterministic ones, we conclude that P is a subset

of NP. $P \subseteq NP$.

What we do not know, and what is an unsolved problem in computer science, is whether $P = NP$ or $P \neq NP$.

It is possible that for all the problems in NP, there exist polynomial time deterministic algorithms that have remained undiscovered?

S. Cook formulated the following question:

Is there any single problem in NP such that if we showed it to be in P, then that would imply that $P = NP$?

Cook answered his own question in the ~~affirmative~~ affirmative with Cook's theorem: Satisfiability is in P iff $P = NP$.

⇒ Let L_1 and L_2 be problems

Problem L_1 reduces to L_2 (also written as $L_1 \propto L_2$) iff 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.

This definition implies that if we have a polynomial time algorithm for L_2, then we can solve L_1 in polynomial time. One can verify the $\propto$ (reducibility) is a transitive relation i.e.

if $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$.

~~the~~

Class NP-Hard :- A problem is NP-Hard if all other problems in NP can be polynomially reduced to it. (OR) A problem L is NP-Hard iff satisfiability reduces to L (satisfiability $\propto$ L).

Class NP-complete :- A problem that is NP-Hard as well as it is in NP is a NP-complete problem. (OR). A problem L is NP-complete iff L is NP-Hard and $L \in NP$.

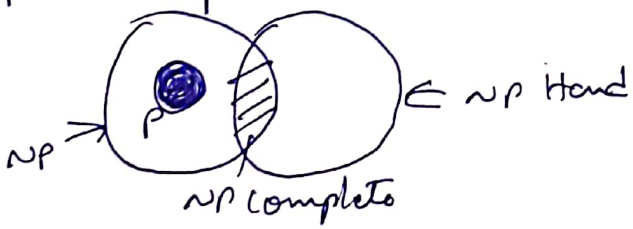
There are NP-Hard problems that are not NP-complete. Only a decision problem can be NP-complete. However, an optimization problem may be NP-Hard. Furthermore if L_1 is a decision problem and L_2 is an optimization problem, it is quite possible that $L_1 \propto L_2$.

Ex:- Knapsack decision problem reduces to the knapsack optimization problem. Clique decision problem can be reduced to clique optimization problem.

Optimization problems can also be reduced to their corresponding decision problems.

Yet, optimization problems cannot be NP-complete they can only be NP-Hard, whereas Decision problems can be NP-complete.

Commonly believed relationship among P, NP, NP complete and NP-Hard problems:

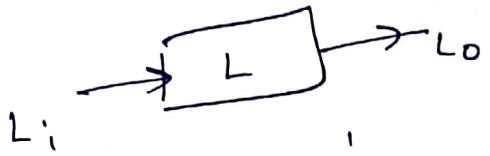


Two problems L_1 and L_2 are said to be polynomially equivalent iff $L_1 \propto L_2$ and $L_2 \propto L_1$.

Reduction :-

A problem L' is said to be polynomial time reducible to problem L, $L' \propto L$, if :

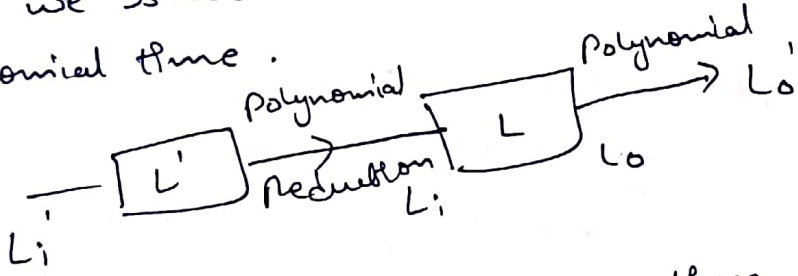
If we have a black box solution for the problem L the input to the black box be L_i and output L_o



Input of L' is L_i'

There should be some way of transforming L_i' to L_i in polynomial time.

And we should be able to transform L_o to L_o' in polynomial time.



Example (1) L = Quadratic equation

L' = Linear solution

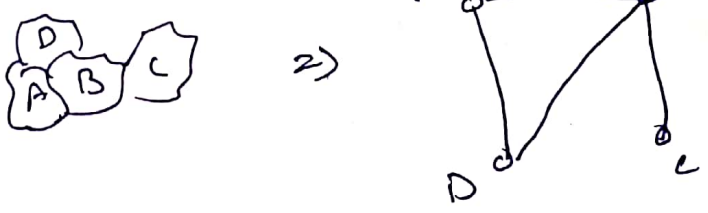
$L = ax^2 + bx + c = 0$; Input = a, b, c ; output = r_1, r_2

$L' = 3x + 5 = 0$

Try to solve L' using L, reduction, should take polynomial time, and less time than actually finding a solution for L' directly.

$\Rightarrow a = 0, b = 3, c = 5.$

(2) Reduce map coloring to graph coloring.



Boolean Formula Satisfiability (SAT) is NP-complete:

[Decision problem].

Given a boolean expression written using only ~~AND~~ $\wedge$ (AND) $\vee$ (OR) $\neg$ (NOT), $\rightarrow$ implication, $\leftrightarrow$ IFF of variables, determine the true or false variable assignment that will make the entire expression true variable $x = 0, 1$.

① $f = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3) \wedge (x_3 \vee x_2)$

The output is yes, because for $x_1 = 1, x_2 = 1, x_3 = 1$, f is satisfied i.e. $f = (1) \wedge (1) \wedge (1) \wedge (1) = 1$.

② $f = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_1)$

the output = NO, because f is not satisfied for any variable assignments.

For n variables, there are 2^n possible assignments.

∴ It is an exponential algorithm and not polynomial.

∴ SAT belongs to NP.

Now prove that SAT is in NP-Hard.

To prove SAT is in NP-Hard we use Cook's theorem which ~~sat~~ states that satisfiability is in P iff $P = NP$.

With the help of ~~this~~ this SAT is in NP-Hard.

NP Hard Graph Problems :-

To prove that a problem is NP-complete, we have to

1) Prove that it belongs to NP-class.

2) Prove that it is a NP-Hard problem.

The strategy we adopt to show that a problem L_2 is NP-hard is:

1) Pick a problem L_1 already known to be NP-Hard
2) Show how to obtain (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I of L_1
3) Conclude from step 2 that $L_1 \propto L_2$
4) Conclude from step 1 and 3 and the transitivity of $\propto$ that L_2 is NP-hard.

1) Clique Decision Problem: (CDP)

To prove clique problem is NP-complete

i) show that clique is in NP

ii) we show CNF-satisfiability $\propto$ CDP

As CNF satisfiability problem is already proved to be NP-Hard

i) Is proved because we have a non-deterministic algorithm that is solvable in polynomial time (i.e verify the solution in polynomial time)

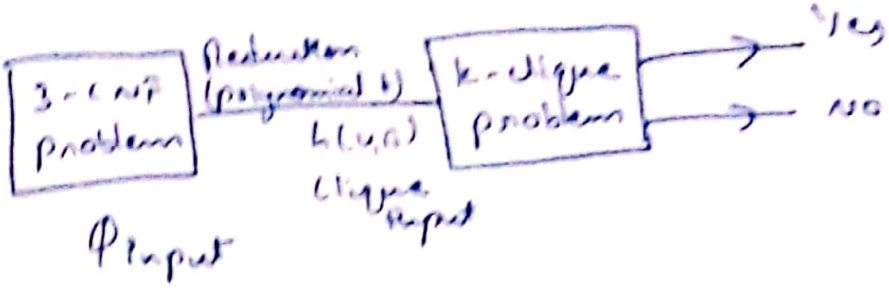
ii) 3-CNF: Circuit satisfiability problem

Let $\phi = C_1 \wedge C_2 \wedge C_3 \ldots \wedge C_k$

ϕ is a propositional formula of k clauses. Each clause C_n has three distinct literals:

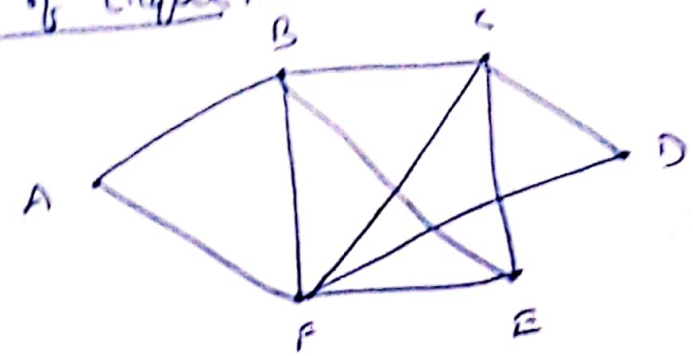
$$l_1^n \vee l_2^n \vee l_3^n$$

we show how to construct from ϕ a graph $G = (V, E)$ such that G has a size at least k iff ϕ is satisfiable.



ϕ Input

Example of clique :-



B, C, E, F is a clique

The maximum clique in the above graph is of size 4

(BFE, CFE, ... are also cliques)

Example of 3-CNF : $\phi = C_1 \wedge C_2 \wedge C_3$

i.e. $\phi = (v_1 \cup \bar{v}_2 \cup \bar{v}_3) \wedge (\bar{v}_1 \cup v_2 \cup v_3) \wedge (v_1 \cup v_2 \cup v_3)$

k-clauses $k = 3$ $\therefore$ 3 clauses

Need to transform ϕ to instance of k-clique ($k=3$)

⇒ For each clause $C_r = l_1^r \cup l_2^r \cup l_3^r$ in ϕ we place triple v_1^r, v_2^r, v_3^r into V of $h(V,E)$

⇒ we put edges in E (between v_i^r and v_j^s)

r, s = clauses, v_i, v_j = literals if :

a) v_i^r and v_j^s are in different triple, i.e. $r \neq s$

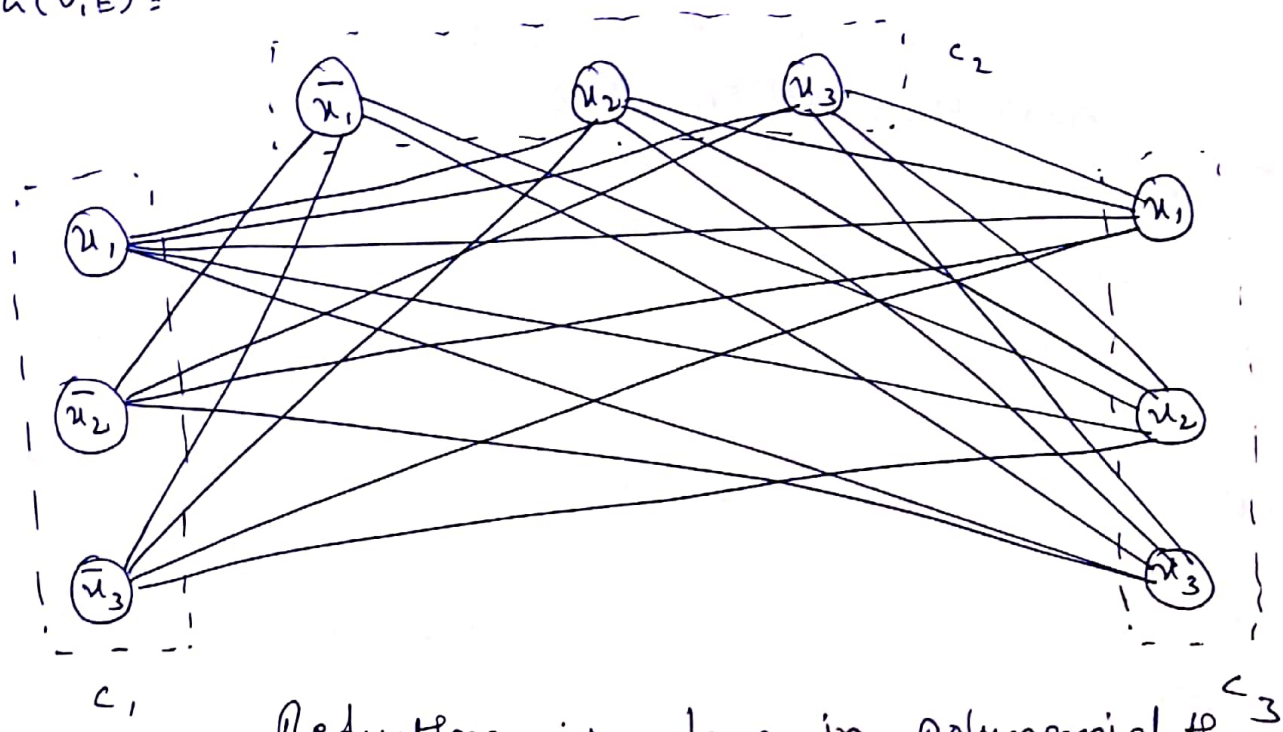
b) literals are consistent i.e. $l_i^r \; != \bar{l}_j^s$

$1 \leq i, j \leq 3$

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

c_1 c_2 c_3 $k=3$

$G(V, E)$ =



Reduction is done in polynomial time

ϕ has a satisfying argument $\Leftrightarrow$ G has a clique of size k.

Now consider: $\phi = 1$, if all c_k has at least one literal = 1

i.e. among all 3 triples in graph G, one of them will be 1 and no negations are connected. Assume $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$ then $\phi = 1$ and in the graph G.

The nodes x_1 from c_1 and x_2 from c_2 and x_3 from c_3 ~~beco~~ become 1 and also $x_1 - x_2 - x_3$ form a 3-clique.

∴ Any combination of input literals (nodes) whose truth values satisfy ϕ, will also form a clique in the graph G. In Reverse, pick any clique of size-k (size 3) from the graph G and assign the node values as 1, they will satisfy the ϕ.

c_1 c_2 c_3

Eg: consider the clique $\bar{x}_2 \rightarrow \bar{x}_1 \rightarrow x_3$

assigning $x_2 = 1$, $\bar{x}_1 = 1$, $x_3 = 1$

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

Hence proved, that CLIQUE is NP Hard

2) Vertex cover (node cover) problem is NP-Hard

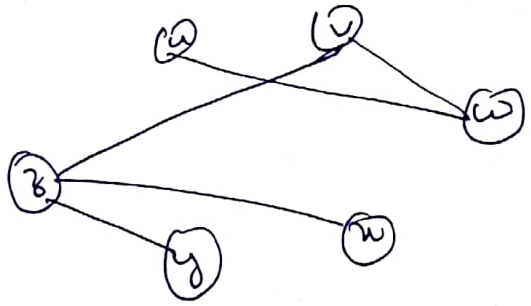
Vertex cover problem

- vertex cover of undirected graph $G(V, E)$ is subset $V' \in V$ such that if $(u, v) \in E \Rightarrow u \in V'$ or $v \in V'$ or both u and $v \in V'$
- Each vertex covers its incident edge
- Find vertex cover of minimum size? (Optimization problem)

Decision Problem :- Does the graph have a vertex cover of size k?

Vertex cover is set of vertices which cover all the edges.

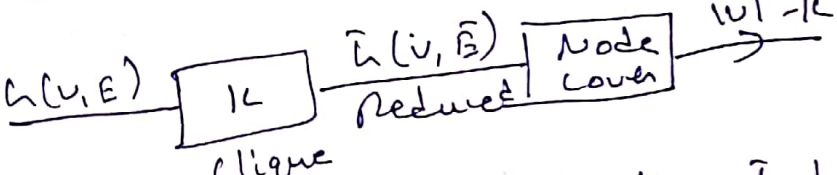
Eg:-



For the above graph the vertex cover is $\{w, z\}$ where size = 2.

because between w and z, they cover all the edges of the graph.

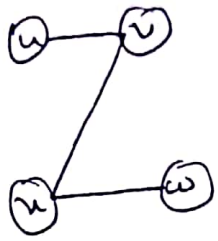
Now that we have proved clique problem is NP hard, we will reduce it to node (vertex) cover problem in polynomial time to prove that the Node cover is NP Hard. Clique $\leq_p$ Vertex cover



→ If G has clique of size k then $\bar{G}$ has a vertex cover of size $|V| - k$.

→ If h doesnot have clique of size k, then $\bar{h}$ doesnot have a vertex cover of size $|V| - k$.

Given a graph $\bar{h}$



and set of nodes $\{v, x\}$ we can verify in polynomial time if $\{v, x\}$ is a vertex cover.

	u	v	w	x
u	1	1	0	0
v	1	1	0	1
w	0	0	1	1
x	0	1	1	1

$|E|$ = edges (u,v), (v,x) and (x,w). we can check in $O[|E|]$ i.e. polynomial time if v,x cover ~~is~~ all the above 3 edges using the adjacency matrix.

∴ the node cover decision problem is NP.

Reducing clique problem of node cover:

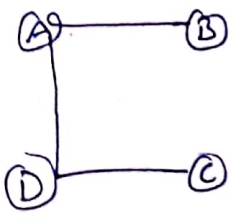
$$h(V, E) \longrightarrow \bar{h}(V, \bar{E})$$

$$\bar{E} = \{(u,v) : u, v \in V, u \neq v \text{ and } u,v \notin E\}$$

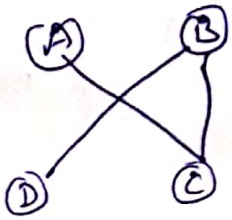
$\bar{E}$ has exactly those edges that are not in h.

we can do this in polynomial time by inverting the values in the adjacency matrix 0 to 1 and 1 to 0.

Eg:- h =



$\bar{h}$ =



Graph G has a clique of size k iff $\bar{G}$ has a vertex cover of size $|V| - k$.

$(u,v) \in E \Rightarrow (u,v) \notin \bar{E}$

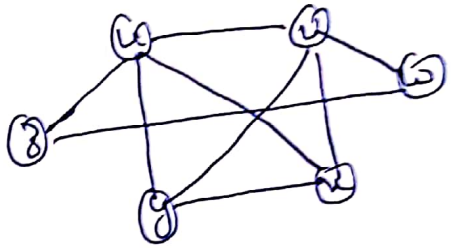
i.e. if $(u,v) \in E \Rightarrow$ atleast u or v does not belong to V', because every pair in V' is connected by an edge in E.

$\Rightarrow$ Atleast one of u or v is in $V - V'$.

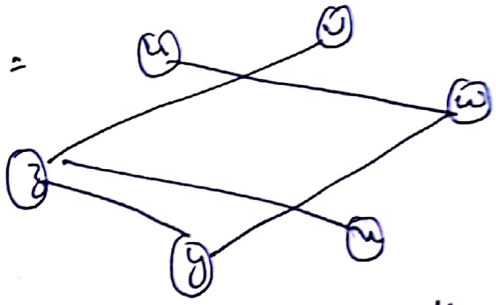
Edge (u,v) is covered by $V - V'$ size.

Consider the following example:

$G(V,E)$ =



$\bar{G}(V,\bar{E})$ =



The graph G has a 4 clique $\{u,v,x,y\}, k=4$

the graph G' has a vertex cover $= \{w,z\}$, size $=2$

$|V| = 6$ (total no. of nodes in G)

size of vertex cover $= |V| - k = 6 - 4 = 2$

Inverse if $\bar{G}$ has a vertex cover of size k, V' is the size of the maximum clique in G

$|V| - V' = k$ i.e. $V' = |V| - k$

$\Rightarrow \bar{G}$ has vertex cover $V' \subseteq V$ $|V'| = |V| - k$ for all $u,v \in V$ if $(u,v) \in \bar{E}$ $u \in V'$ or $v \in V'$ or both.

for all $u, v \in V$, if $u \notin v'$ and $v \notin v'$

$\Rightarrow (u, v) \in E$, $V - V'$ is clique

size $= |V| - |V'| = k$.

∴ Node (vertex) covering problem is NP Hard.

Other graph problems that can be proved NP Hard are: Directed Hamiltonian cycle
Traveling Salesperson problem etc

NP Hard code generation problems

- A function of a compiler is to translate programs written in some source language into an equivalent assembly language or machine language program.
- we look at the problem of translating arithmetic expressions in a language such as C++ into assembly language code.
- we assume a simple machine A, which has only one register called the accumulator. All arithmetic has to be performed in this register.
- If $\odot$ represents a binary operator such as +, −, ×, / then the left operand of $\odot$ must be in the accumulator. For simplicity, we restrict ourselves to these four operators.
- The relevant assembly language instructions are:

```
LOAD X   // Load accumulator with contents of memory
            location X.
STORE X  // Store contents of accumulator into memory
            location X.
OP X     // OP may be ADD, SUB, MPY or DIV
```

The instruction OP X, computes the operator OP using the contents of the accumulator as the left operand and that of memory location X as the right operand.

Consider the arithmetic expression : (a+b)/(c+d)

Let T_1, T_2 be temporary storage areas in memory. Two possible assembly language versions of this expression are :

a)
```
LOAD  a
ADD   b
STORE T1
LOAD  c
ADD   d
STORE T2
LOAD  T1
DIV   T2
```

(b)
```
LOAD  c
ADD   d
STORE T1
LOAD  a
ADD   b
DIV   T1
```

If each instructions takes the same amount of time, code (b) will take 25% less time than code (a)

∴ For the given machine A, and expressions (a+b)/(c+d), code (b) is optimal.

A translation of sub expression E into the machine or assembly language of a given machine is optimal if and only if it has a minimum number of instructions.

A binary operator ⊙ is commutative in domain D if $a \odot b = b \odot a$ for all a and b in D.

Code generation with common subexpressions :

When arithematic expressions have common subexpressions, they can be represented by a directed acyclic graph (dag).

- Every internal node (node with non-zero outdegree) in the dag represents an operator.

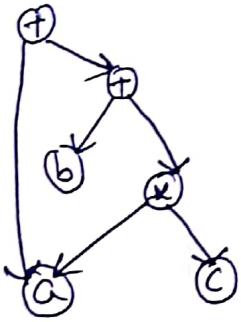
- Assuming the expression contains only binary operators, each internal node P has out degree two
- The two nodes adjacent from P are called the left and right children of P respectively. The children of P are the roots of the dags for the left and right operands of P.

Definitions :-

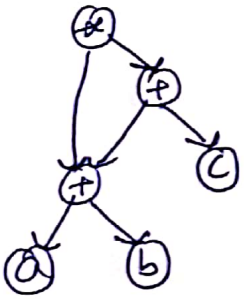
- A leaf is a node with out-degree zero.
- A level-one node is a node both of whose children are leaves.
- A shared node is a node with more than one parent.
- A leaf dag is a dag in which all shared nodes are leaves.
- A level-one dag is a dag in which all shared nodes are level-one nodes.

Examples of some expressions and their dag representations :

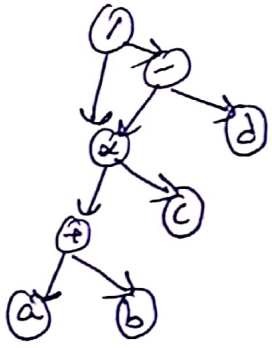
① a + (b + a * c)



② (a+b) * (a+b+c)



③ (a+b) * c / ((a+b) * c-d)



- The dag of ① is a leaf dag.
- The dag of ② is a level-one dag.
- The dag of ③ is neither a leaf dag nor a level-one dag.

→ A leaf dag results from an arithmetic expression in which the only common subexpressions are simple variables or constants.

→ A level-one dag results from an expression in which the only common subexpressions are of the form $a \odot b$, where a and b are simple variables or constants and $\odot$ is an operator.

- The problem of generating optimal code for level-one dags is NP-Hard even when the machine for which code is being generated has only one register.

Determining the minimum number of registers needed to evaluate a dag with no STORES is also NP-Hard. To prove the above statements, we use the feedback node set (FNS) problem that is already proved to be NP-Hard.

Feedback Node Set Problem (FNS):

Given a directed graph $h = (V, E)$ and an integer k, determine whether there exists a subset V' of vertices $V' \subseteq V$ and $|V'| \leq k$ such that the graph $H = (V - V', E - \{(u, v) \mid u \in V' \text{ or } v \in V'\})$ obtained from h by deleting all vertices in V' contains no directed cycles.

FNS & optimal code generation for level-one dags on one register machine.