# Module IV

# Complexity Theory

## NP Hard and NP Complete Problems

### Decision problems vs. optimization problems

The problems we are trying to solve are basically of two kinds. In **decision problems** we are trying to decide whether a statement is true or false. In optimization problems we are trying to find the solution with the best possible score according to some scoring scheme. **Optimization problems** can be either maximization problems, where we are trying to maximize a certain score, or minimization problems, where we are trying to minimize a cost function.

Example 1: Hamiltonian cycles

Given a directed graph, we want to decide whether or not there is a Hamiltonian cycle in this graph. This is a decision problem.

Example 2: TSP - The Traveling Salesman Problem

Given a complete graph and an assignment of weights to the edges, find a Hamiltonian cycle of minimum weight. This is the optimization version of the problem. In the decision version, we are given a weighted complete graph and a real number c, and we want to know whether or not there exists a Hamiltonian cycle whose combined weight of edges does not exceed **c.**

Each optimization problem has a corresponding decision problem.

## Deterministic and non-deterministic algorithms

In the context of programming, an Algorithm is a set of well-defined instructions in sequence to perform a particular task and achieve the desired output. Here we say set of defined instructions which means that somewhere user knows the outcome of those instructions if they get executed in the expected manner.

On the basis of the knowledge about outcome of the instructions, there are two types of algorithms namely − Deterministic and Non-deterministic Algorithms.

A deterministic algorithm is an algorithm that is purely determined by its inputs, where no randomness is involved in the model. **Deterministic**

**algorithms will always come up with the same result given the same inputs**. **Algorithms such that the result of every operation is uniquely defined are called *deterministic algorithms***.

Up to now, we have focused on problems that were deterministic.

A non-deterministic algorithm can provide different outputs for the same input on different executions. Unlike a deterministic algorithm which produces only a single output for the same input even on different runs, a non-deterministic algorithm travels in various routes to arrive at the different outcomes.

Non-deterministic algorithms are useful for finding approximate solutions, when an exact solution is difficult or expensive to derive using a deterministic algorithm.

Discussing **nondeterministic algorithms** requires 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

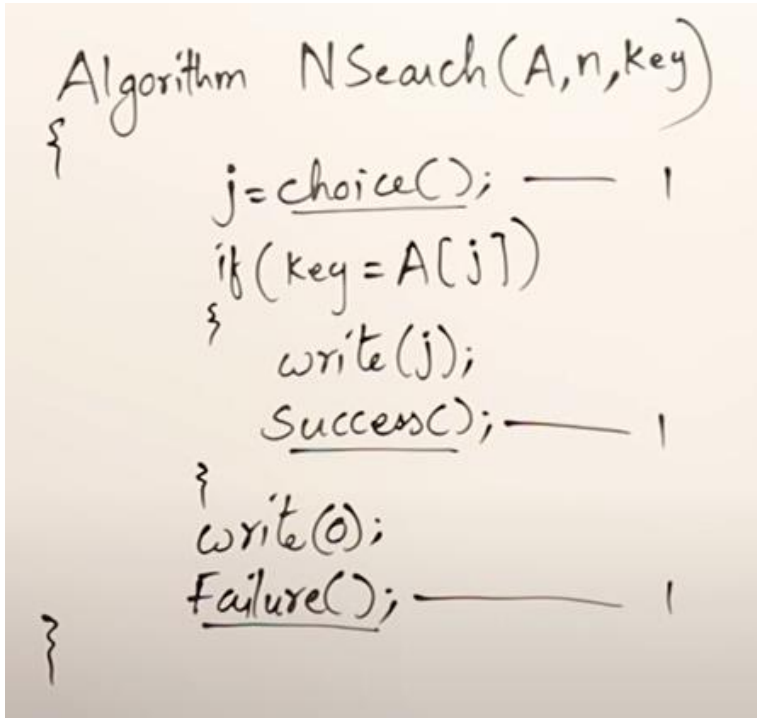
*A nondeterministc algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a successful signal*.

***Nondeterministic search***

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

The computing times for **Choice, Success**, and **Failure** are taken to be O(1).

Eg.

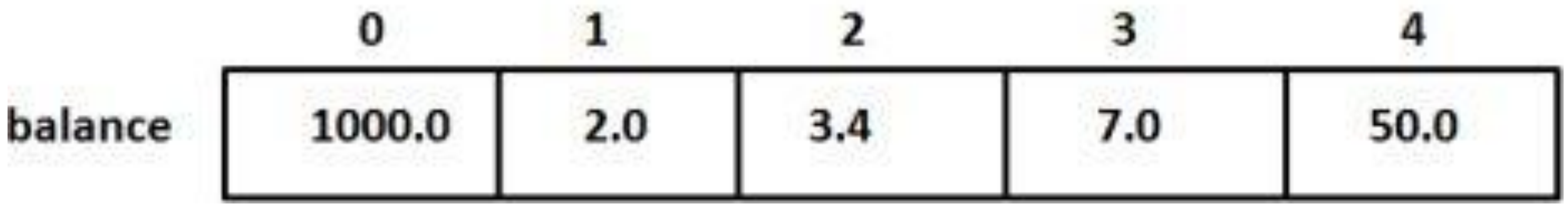


```
Algorithm NSearch(A,n,key)
{
    j=choice(); —— 1
    if(key=A[j])
    {
        write(j);
        Success(); —— 1
    }
    write(0);
    Failure(); —— 1
}
```

J=Choice() gives the index of the key element. If element in jth index is key element then search is successful. Otherwise failure.

How this choice came to know that key element is present in that jth position. That’s why it is non deterministic. If we are able to find the key position in constant time then we can fill up this choice() function.

For example,



If we know the key is at position 3 (7.0) in a constant time it will become deterministic. That is algorithm directly gets the answer. (method is not known now, may be developed in future)

A machine capable of executing a nondeterminsistic algorithm in this way is called a *nondeterministic machine*.

| **Sr. No.** | **Key** | **Deterministic Algorithm** | **Non-deterministic Algorithm** |
|---|---|---|---|
| 1 | Definition | The algorithms in which the result of every algorithm is uniquely defined are known as the Deterministic Algorithm. | On other hand, the algorithms in which the result of every algorithm is not uniquely defined and result could be random are |

| Sr. No. | Key | Deterministic Algorithm | Non-deterministic Algorithm |
|---|---|---|---|
| | | In other words, we can say that the deterministic algorithm is the algorithm that performs fixed number of steps and always get finished with an accept or reject state with the same result. | known as the Non-Deterministic Algorithm. |
| 2 | Execution | In Deterministic Algorithms execution, the target machine executes the same instruction and results same outcome which is not dependent on the way or | On other hand in case of Non-Deterministic Algorithms, the machine executing each operation is allowed to choose any one of these outcomes subjects |

| Sr. No. | Key | Deterministic Algorithm | Non-deterministic Algorithm |
|---|---|---|---|
| | | process in which instruction get executed. | to a determination condition to be defined later. |
| 3 | Type | On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for a particular input instructions the machine will give always the same output. | On other hand Non deterministic algorithm are classified as non-reliable algorithms for a particular input the machine will give different output on different executions. |
| 4 | Execution Time | As outcome is known and is consistent on different | On other hand as outcome is not known and is non- |

| Sr. No. | Key | Deterministic Algorithm | Non-deterministic Algorithm |
|---|---|---|---|
| | | executions so Deterministic algorithm takes **polynomial time** for their execution. | consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time. |
| 5 | Execution path | In deterministic algorithm the path of execution for algorithm is same in every execution. | On other hand in case of Non-Deterministic algorithm the path of execution is not same for algorithm in every execution and could take any random path for its execution. |

## Polynomial time and exponential time algorithms

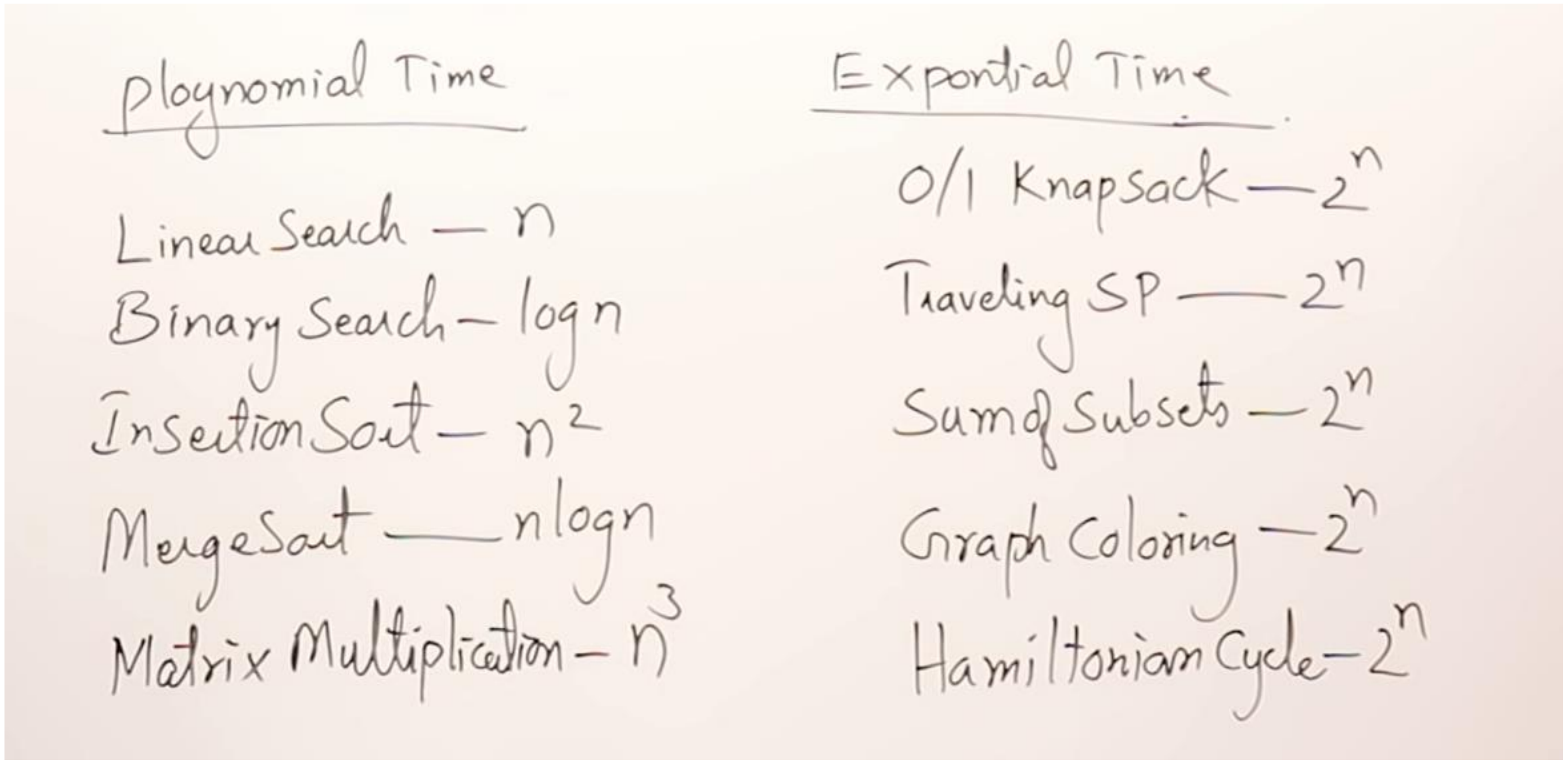
Basically, we have problems that can be solved in a **short time (polynomial)** and ones that **require vast amounts of time** because of the numerous possibilities inherent in the problems (**non polynomial**). The first group consists of problems whose solution times are bounded by polynomials of small degree.

Eg. O(log n), O(n)

The second group is made up of problems whose best-known algorithms are non polynomial.

Eg. $O(n^2 2^n)$, $2^{n/2}$

Examples are given here.



| Ploynomial Time | | Expontial Time | |
|---|---|---|---|
| Linear Search | $n$ | 0/1 Knapsack | $2^n$ |
| Binary Search | $\log n$ | Traveling SP | $2^n$ |
| Insertion Sort | $n^2$ | Sum of Subsets | $2^n$ |
| MergeSort | $n\log n$ | Graph Coloring | $2^n$ |
| Matrix Multiplication | $n^3$ | Hamiltonian Cycle | $2^n$ |

Exponential time taking algorithms are much bigger than Polynomial time taking algorithms

So we want polynomial time algorithms for solving these exponential time algorithms.

**Classes of Algorithms**

**Class P**

**P** is a set of all **decision problems** solvable by **deterministic algorithms** in **polynomial time**.

The class P EXAMPLE: The Minimum Spanning Tree Problem is in the class P.

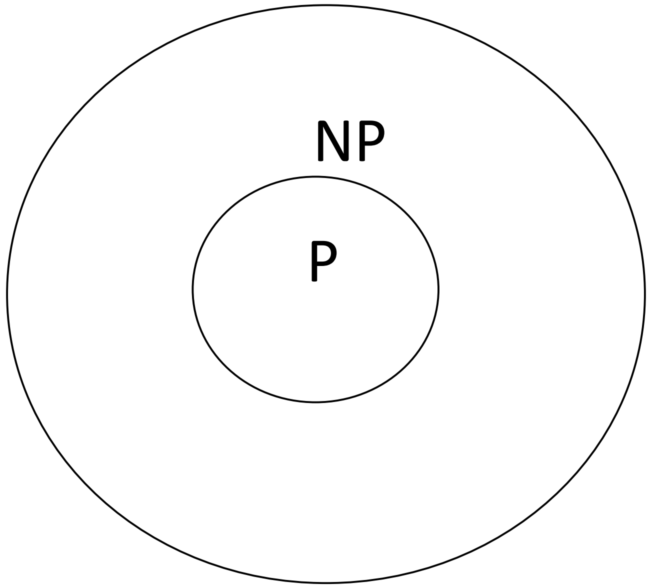
**The class NP**

A problem that is NP-Complete has the property that it can be solved in polynomial time iff all other NP-Complete problems can also be solved in polynomial time. **If an NP-Hard problem can be solved in polynomial time, then all NP-Complete problems can be solved in polynomial time. All NP-Complete problems are NP-Hard, but not all NP-Hard problems are NP-Complete.**

**NP** stands for **Nondeterministic Polynomial**

NP is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Deterministic algorithms are the special case of non deterministic one. The



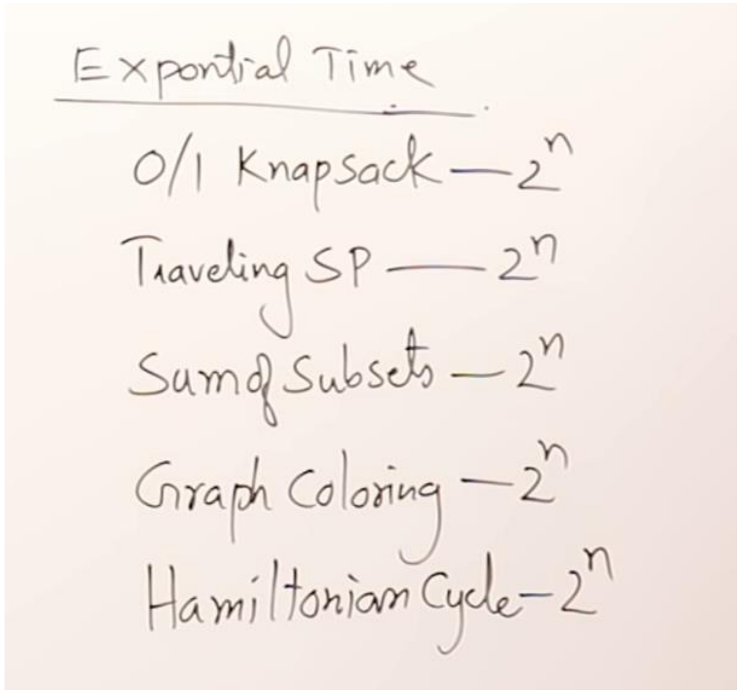
given figure shows the relationship between P and NP.

**The P=NP Problem**

It is not hard to show that every problem in P is also in NP, but it is unclear whether every problem in NP is also in P. The P=NP Problem. The best we can say is that thousands of computer scientists have been unsuccessful for decades to design polynomial-time algorithms for some problems in the class NP. This constitutes overwhelming empirical evidence that the classes P and NP are indeed distinct, but no formal mathematical proof of this fact is known.

**The Satisfiability problem (CNF-Satisfiability)**

If one problem is solved in polynomial time in the given example it is easy to solve other problems also in polynomial time.



Expontial Time

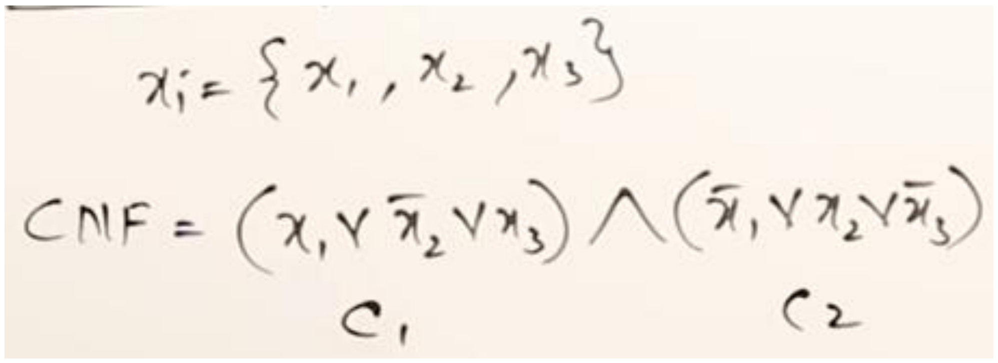
| | |
|---|---|
| 0/1 Knapsack | $2^n$ |
| Traveling SP | $2^n$ |
| Sum of Subsets | $2^n$ |
| Graph Coloring | $2^n$ |
| Hamiltonian Cycle | $2^n$ |

To do this, we take satisfiability problem as base problem.

- An expression *E* is *satisfiable* if there exists a truth assignment to the variables in *E* that makes *E* true.

- The *satisfiability problem* (SAT) is to determine whether a given boolean expression is satisfiable.
- SAT can be used to prove that other problems are NP complete by showing that the other problem is in NP and that SAT can be reduced to the other problem in polynomial time.

For example



$$x_i = \{x_1, x_2, x_3\}$$

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

The **satisfiability problem is to find out in what values of xi this formula is true.**

This is CNF propositional calculus formula.

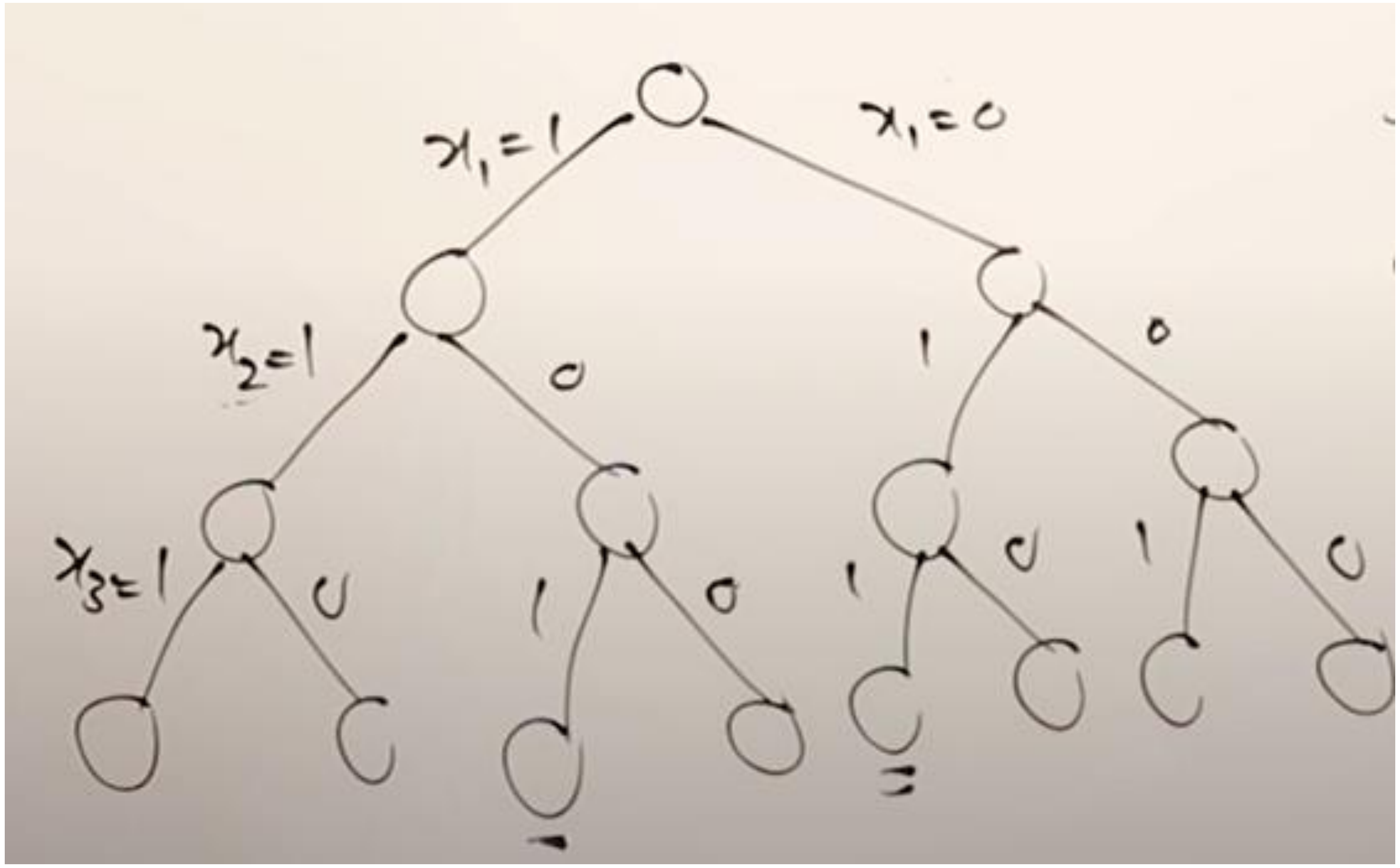
The possible values of xi are:

| x1 | x2 | x3 |
|---|---|---|
| 0 | 0 | 0 |

|   |   |   |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Example : check the value 8 -> $2^3$ -> $2^n$ – **exponential time taking problem**

State space tree of the above values are:



If **satisfiability solved in polynomial time** all the exponential time algorithms can be solved in polynomial time. Satisfiability problem can be solved in polynomial time.

For example 0/1 Knapsack problem, p={10,2,3} and w={5,4,3}, m=8 and n=3. This can be represented by Boolean variables such as

xi = {0/1, 0/1,0/1}

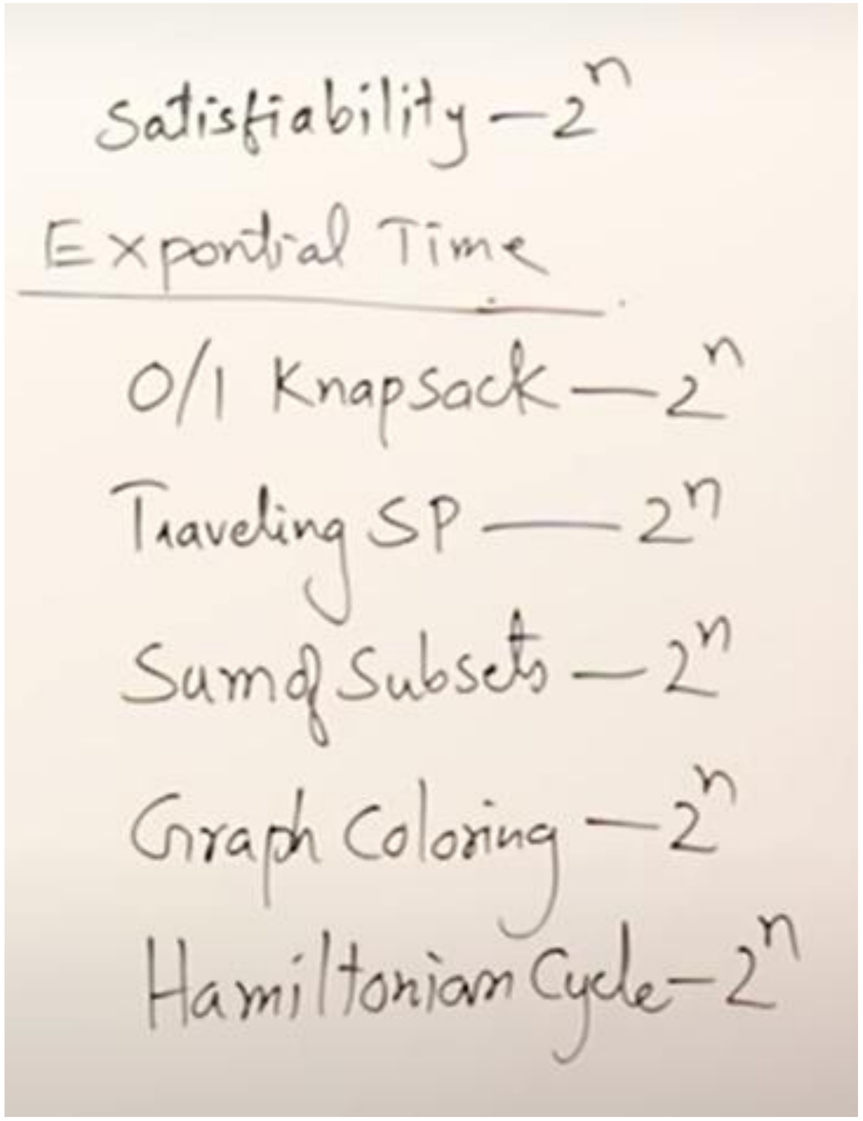
| | | |
|---|---|---|
| x1 | x2 | x3 |
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |

……….

……….

n=3, so $2^3$ -> $2^n$

0/1 Kanpsack problem can also be solved using the above state space tree.



Satisfiability — $2^n$

Expontial Time

0/1 Knapsack — $2^n$

Traveling SP — $2^n$

Sum of Subsets — $2^n$

Graph Coloring — $2^n$

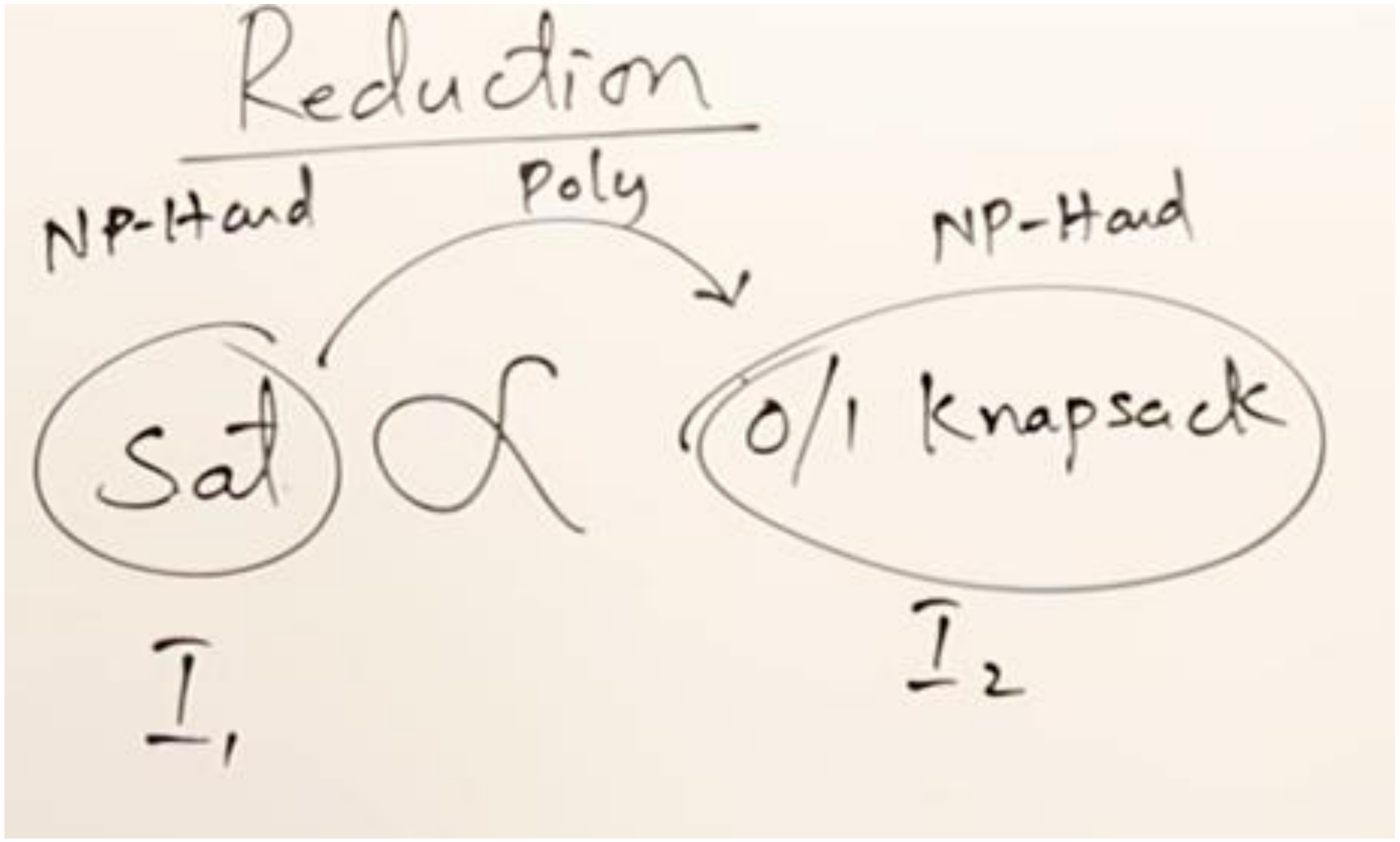
Hamiltonian Cycle — $2^n$

All the problems solved in exponential time complexity are hard problems. **Satisfiability problem is the base problem of all these problems**. We can call **satisfiability problem** as hard problem, NP- hard problem. If satisfiability can solve in polynomial time all the other hard problems can also solve in polynomial time.  In order to show the relationship between **satisfiability** and all the above problems we can use **the procedure reduction**

**Polynomial-time reducibility**

Let L1 and L2 be problems.  Problem L1 reduces to L2 (also written as L1 α L2) if and only if there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time.

For example: We take the example of satisfiability problem and we convert that problem into 0/1 knapsack problem and we say that if this 0/1 knapsack problem solved in polynomial time then the same algorithm can be used for solving satisfiability problem also in polynomial time and vice versa. They are related each other.  Conversion is done in polynomial time. Here satisfiability is NP-hard so 0/1 Knapsack problem is also NP-hard.



If satisfiability problem can be reduced to L then L is also NP-Hard. Reduction has transitive property.

SAT α L1, L1 α L1 -> **SAT α L2**

Non deterministic polynomial time algorithm for Satisfiability problem is existing so it is NP-Complete also. When we write a non-deterministic polynomial time algorithm for NP-hard problem then it is in NP-Complete class.

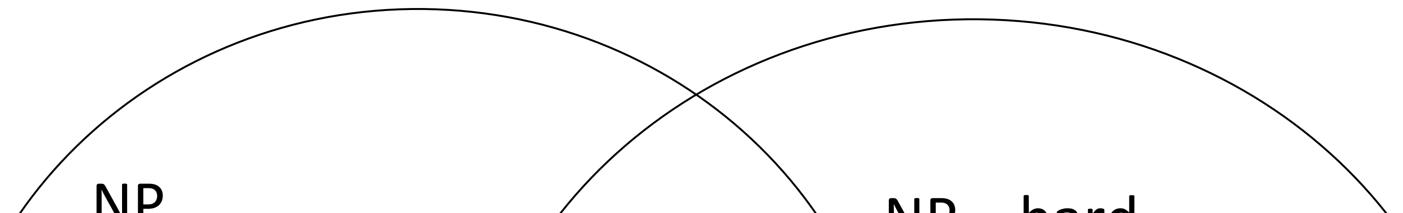
## NP-complete problems

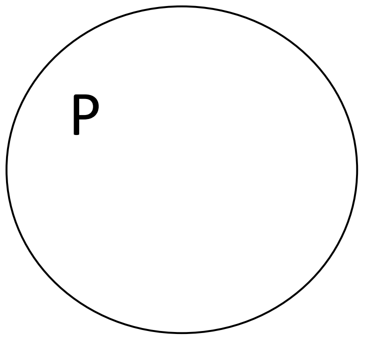
A decision problem E is NP-complete if every problem in the class NP is polynomial-time reducible to E. The Hamiltonian cycle problem, the decision versions of the TSP and the graph coloring problem, as well as literally hundreds of other problems are known to be NP-complete.

## NP-hard problems

Optimization problems whose decision versions are NP complete are called NP-hard.

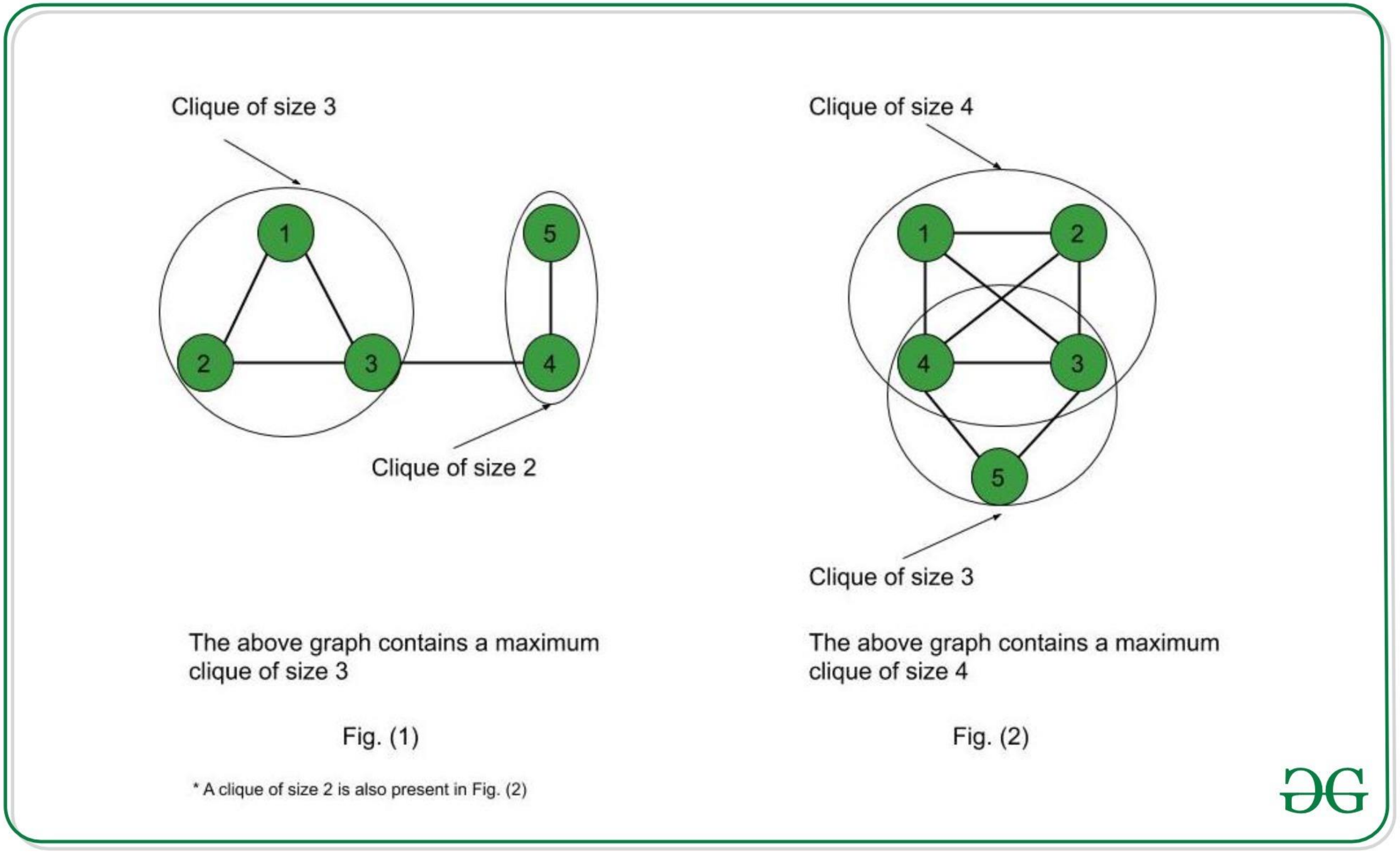
Following figure shows the relationship among P. NP, NP-complete, and NP-hard.





## Clique Decision Problem (CDP)

A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other that is the subgraph is a complete graph. The Maximal Clique Problem is to find the maximum sized clique of a given graph G, that is a complete graph which is a subgraph of G and contains the maximum number of vertices. This is an optimization problem. Correspondingly, the Clique Decision Problem is to find if a clique of size k exists in the given graph or not.


Clique of size 3
1
2
3
5
4
Clique of size 2
The above graph contains a maximum clique of size 3
Fig. (1)
* A clique of size 2 is also present in Fig. (2)
Clique of size 4
1
2
4
3
5
Clique of size 3
The above graph contains a maximum clique of size 4
Fig. (2)

To prove that a problem is NP-Complete, we have to show that it belongs to both NP and NP-Hard Classes. (Since NP-Complete problems are NP-Hard problems which also belong to NP)

**The Clique Decision Problem belongs to NP**

If a problem belongs to the NP class, then it should have polynomial-time verifiability, that is given a certificate, we should be able to verify in polynomial time if it is a solution to the problem.

**Proof:**

1. Certificate – Let the certificate be a set **S** consisting of nodes in the clique and S is a subgraph of G.{1,2,3} ->S=k
2. Verification – We have to check if there exists a clique of size k in the graph. Hence, verifying if number of nodes in S equals k, takes O(1) time. Verifying whether each vertex has an out-degree of (k-1) takes O($k^2$) time. (Since in a complete graph, each vertex is connected to every

other vertex through an edge. Hence the total number of edges in a complete graph = ${}^{k}C_{2}$ = **k*(k-1)/2).** Therefore, to check if the graph formed by the k nodes in S is complete or not, it takes $O(k^2) = O(n^2)$ time (since k<=n, where n is number of vertices in G).

Therefore, the Clique Decision Problem has polynomial time verifiability and hence belongs to the NP Class.

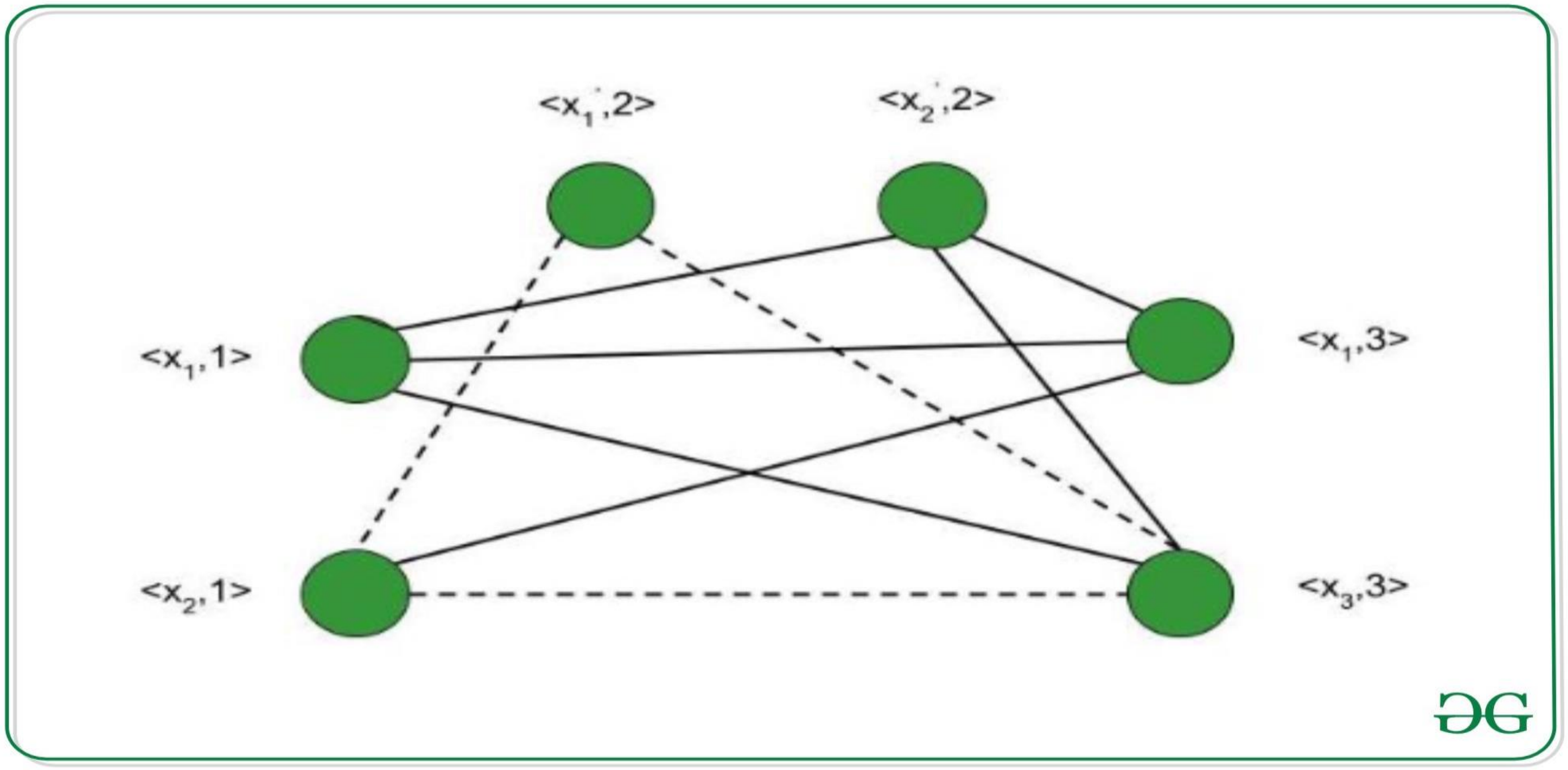
**The Clique Decision Problem belongs to NP-Hard**

A problem L belongs to NP-Hard if every NP problem is reducible to L in polynomial time. Now, let the Clique Decision Problem by C. To prove that C is NP-Hard, we take an already known NP-Hard problem, say S, and reduce it to C for a particular instance. If this reduction can be done in polynomial time, then C is also an NP-Hard problem. The Boolean Satisfiability Problem (S) is an NP-Complete problem as proved by the Cook's theorem. Therefore, every problem in NP can be reduced to S in polynomial time. Thus, if S is reducible to C in polynomial time, every NP problem can be reduced to C in polynomial time, thereby proving C to be NP-Hard.

**Proof that the Boolean Satisfiability problem reduces to the Clique Decision Problem**

Let the Boolean expression be – **F = ($x_1$ v $x_2$) ^ ($x_1$' v $x_2$') ^ ($x_1$ v $x_3$)** where $x_1$, $x_2$, $x_3$ are the variables, '^' denotes logical 'and', 'v' denotes logical 'or' and **x'** denotes the **complement of x**. Let the expression within each parentheses be a clause. Hence we have three clauses – $C_1$, $C_2$ and $C_3$. Consider the vertices as – <$x_1$, 1>; <$x_2$, 1>; <$x_1$', 2>; <$x_2$', 2>; <$x_1$, 3>; <$x_3$, 3> where the second term in each vertex denotes the clause number they belong to. We connect these vertices such that –

1. No two vertices belonging to the same clause are connected.
2. No variable is connected to its complement.



Thus, the graph G (V, E) is constructed such that – V = { <a, i> | a belongs to $C_i$ } and E = { ( <a, i>, <b, j> ) | i is not equal to j ; b is not equal to a' .

Consider the subgraph of G with the vertices <$x_2$, 1>; <$x_1$', 2>; <$x_3$, 3>. It forms a clique of size 3 (Depicted by dotted line in above figure) . Corresponding to this, for the assignment – <$x_1$, $x_2$, $x_3$> = <0, 1, 1> F evaluates to true. Therefore, if we have k clauses in our satisfiability expression, we get a max clique of size k and for the corresponding assignment of values, the satisfiability expression evaluates to true. Hence, for a particular instance, the satisfiability problem is reduced to the clique decision problem.

Therefore, the **Clique Decision Problem is NP-Hard.**

## Conclusion

The Clique Decision Problem is NP and NP-Hard. Therefore, the Clique decision problem is **NP-Complete**.

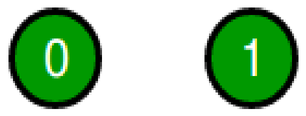
## Vertex Cover Problem (Node Cover Decision Problem (NCDP))

The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph.

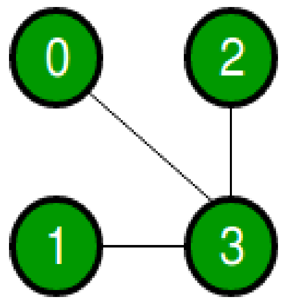
Vertex cover of a graph is a subset of vertices which cover every edge.

An edge is covered if one of its endpoint is chosen.

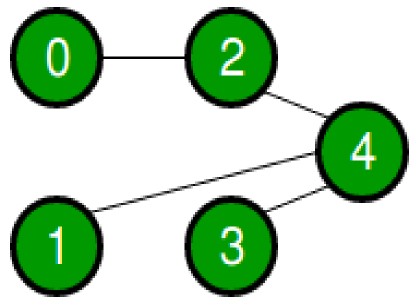
Eg.



Minimum vertex cover is empty{}



Minimum vertex cover is {3}



Minimum vertex cover is {4, 2} or {4, 0}

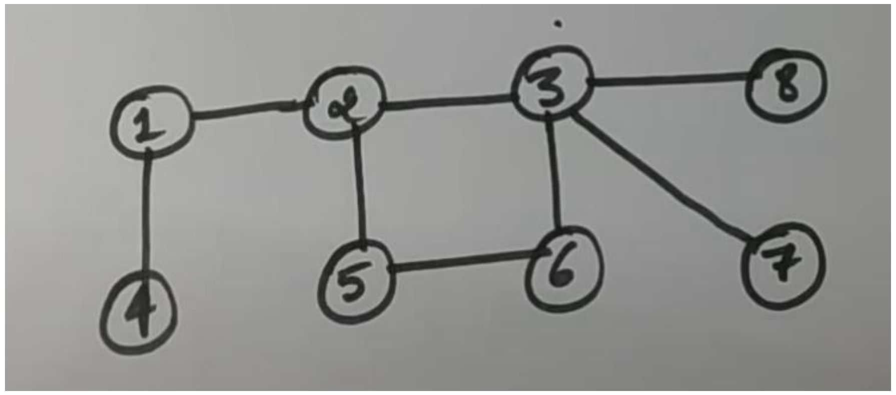
It is solved by

- Greedy Approach –Get best solution
- Approximation Method – Get nearby solution

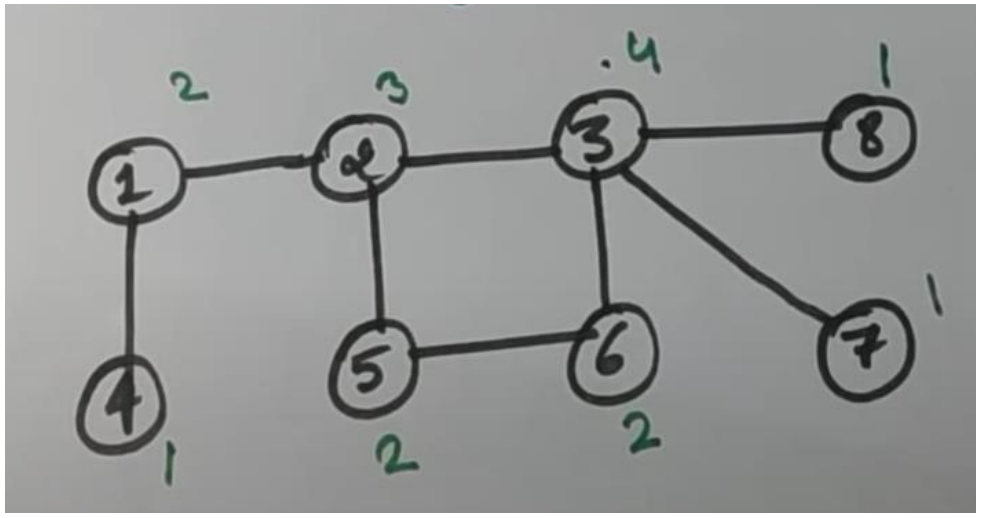
Greedy method steps

- Find a vertex v with maximum degree (Maximum number of edges associated with a vertex)
- Add v to the solution set and remove v and all its incident edges from the graph
- Repeat until all edges are covered.

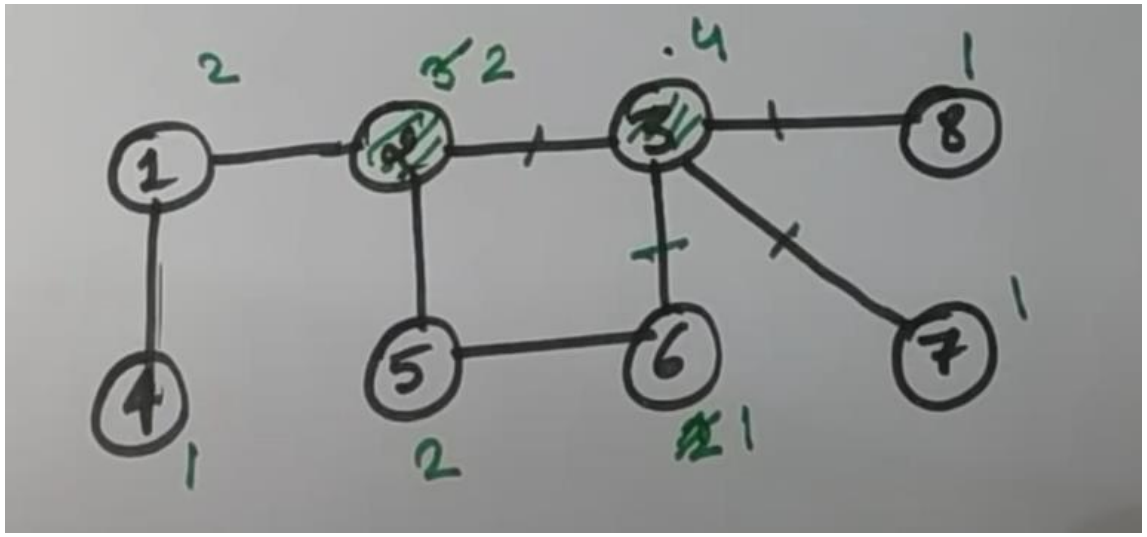
As an example, consider the graph



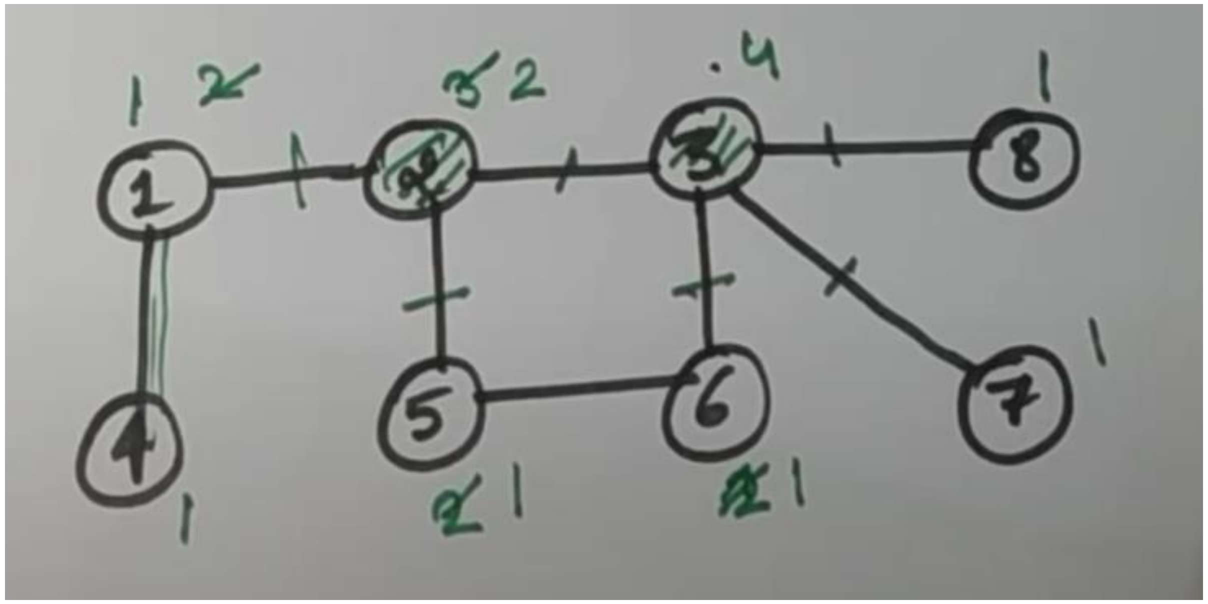
- Find the degree of all graph



- Choose the vertex which has maximum degree, vertex 3 and remove all the edges associated with vertex 3. Recalculate the degree of remaining vertices.
- Removed vertex set - {3}



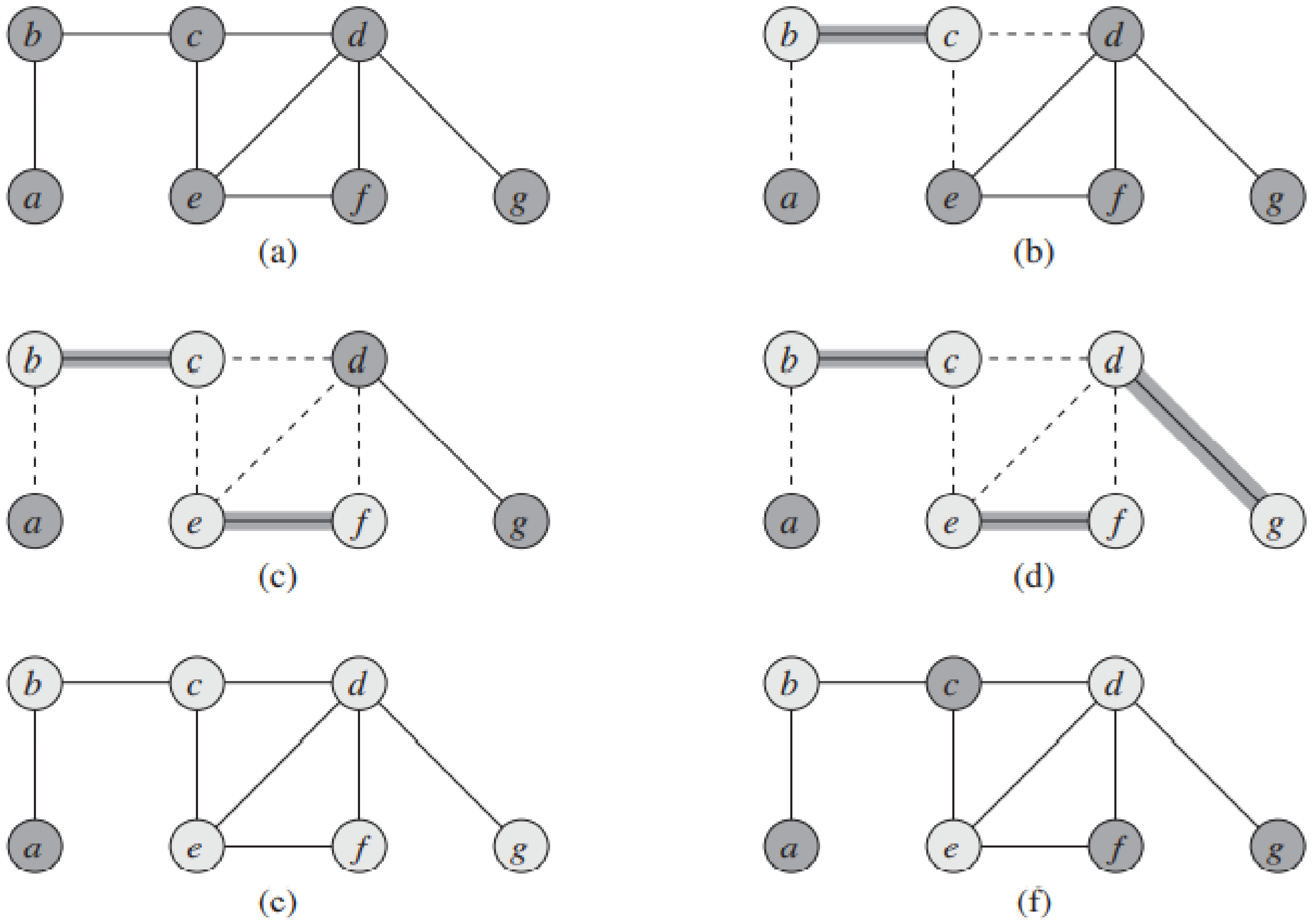
- Remove next vertex with minimum degree , vertex 2 and associated edges
- Removed vertex set - {3, 2}
- Recalculate the degree and remove the vertex with minimum degree



- Removed vertex set - {3, 2, 1}
- Next removed vertex is 5 so {3,2,1,5} is new set
- {3,5,1} and {3,1, 5} are other solutions from this set. Time complexity is O(V+E)
- It is not an optimal solution

## APPROX Vertex Cover

It is an optimal vertex cover. This problem is the optimization version of an NP-complete decision problem. Even though we don’t know how to find an optimal vertex cover in a graph G in polynomial time, we can efficiently find a vertex cover that is near-optimal. The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.



The operation of APPROX-VERTEX-COVER.

(a) The input graph G, which has 7 vertices and 8 edges.

(b) The edge (b, c) shown heavy, is the first edge chosen by APPROX-VERTEX COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b),(c,e), and (c,d) shown dashed, are removed since they are now covered by some vertex in C.

(c) Edge (e, f) is chosen; vertices e and f are added to C.

(d) Edge (d, g) is chosen; vertices d and g are added to C.

(e) The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b,c, d, e, f, g.

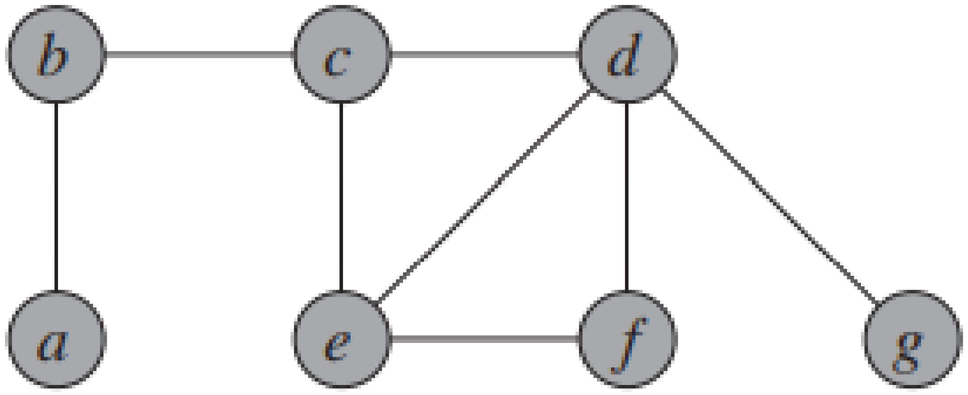
(f) The optimal vertex cover for this problem contains only three vertices: **b, d, and e.**

## Algorithm

```
APPROX-VERTEX-COVER($G$)
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
    let $(u, v)$ be an arbitrary edge of $E'$
    $C = C \cup \{u, v\}$
    remove from $E'$ every edge incident on either $u$ or $v$
return $C$
```

The running time of this algorithm is O (V + E), using adjacency lists to represent E’

## Optimal solution

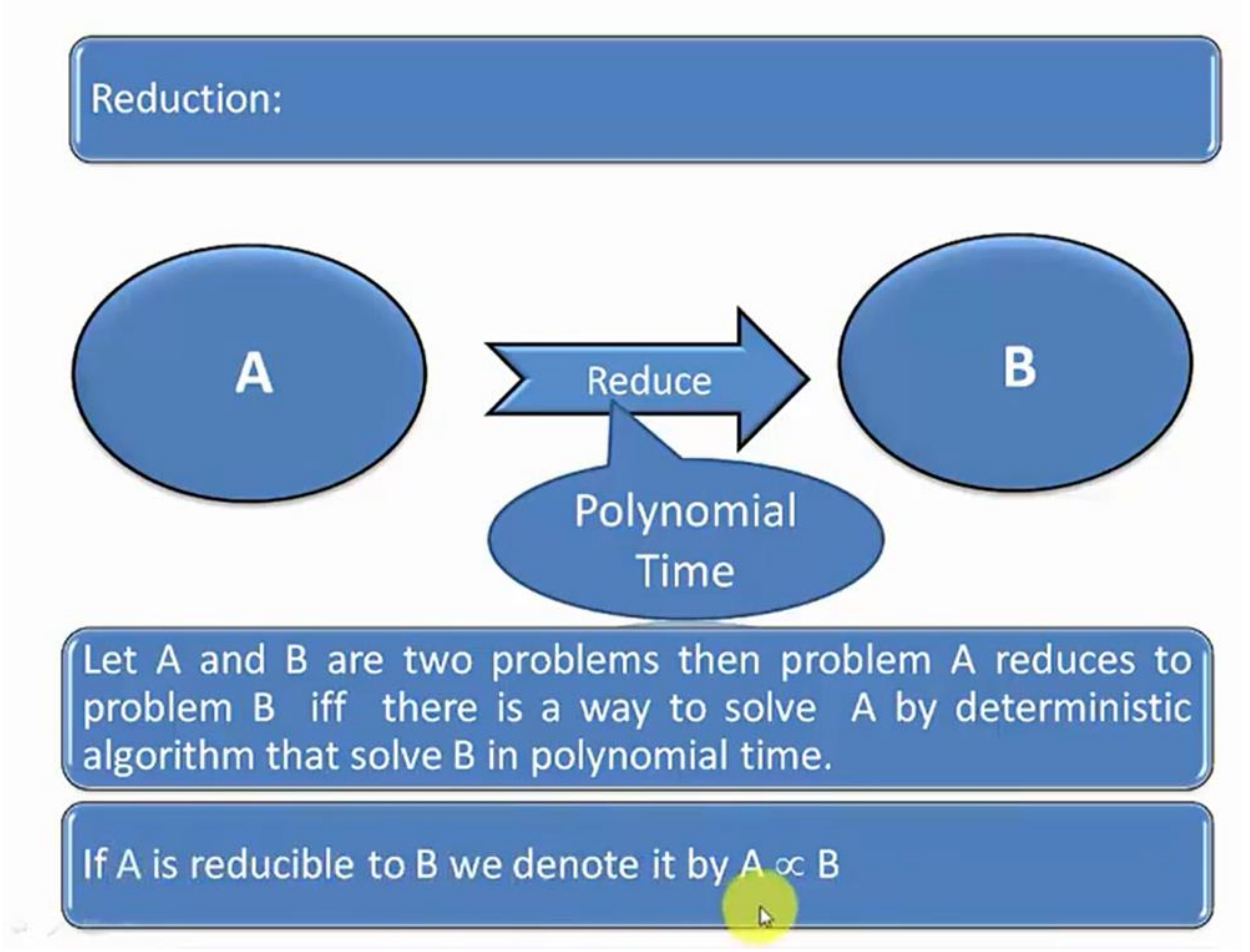


Select the vertex which has highest degree. That is node d.

Remove all the edges connected into it.

Select the highest degree vertex in the rest of the vertices (a, b, c, e and f), that is b or c or e. Select b, remove the edges, after that choose e and remove edge. Final list of C={b, d, e}

## NP-Completeness of Vertex Cover



Reduction:

Let A and B are two problems then problem A reduces to problem B iff there is a way to solve A by deterministic algorithm that solve B in polynomial time.

If A is reducible to B we denote it by $A \propto B$

Properties:

1. if A is reducible to B and B in P then A in P.
2. A is not in P implies B is not in P

If A and B are two languages. If A is reducible to B in polynomial time then B is NP-Hard.

If B is in NP, then B is NP-Complete.

## Steps for proving NP-Complete:

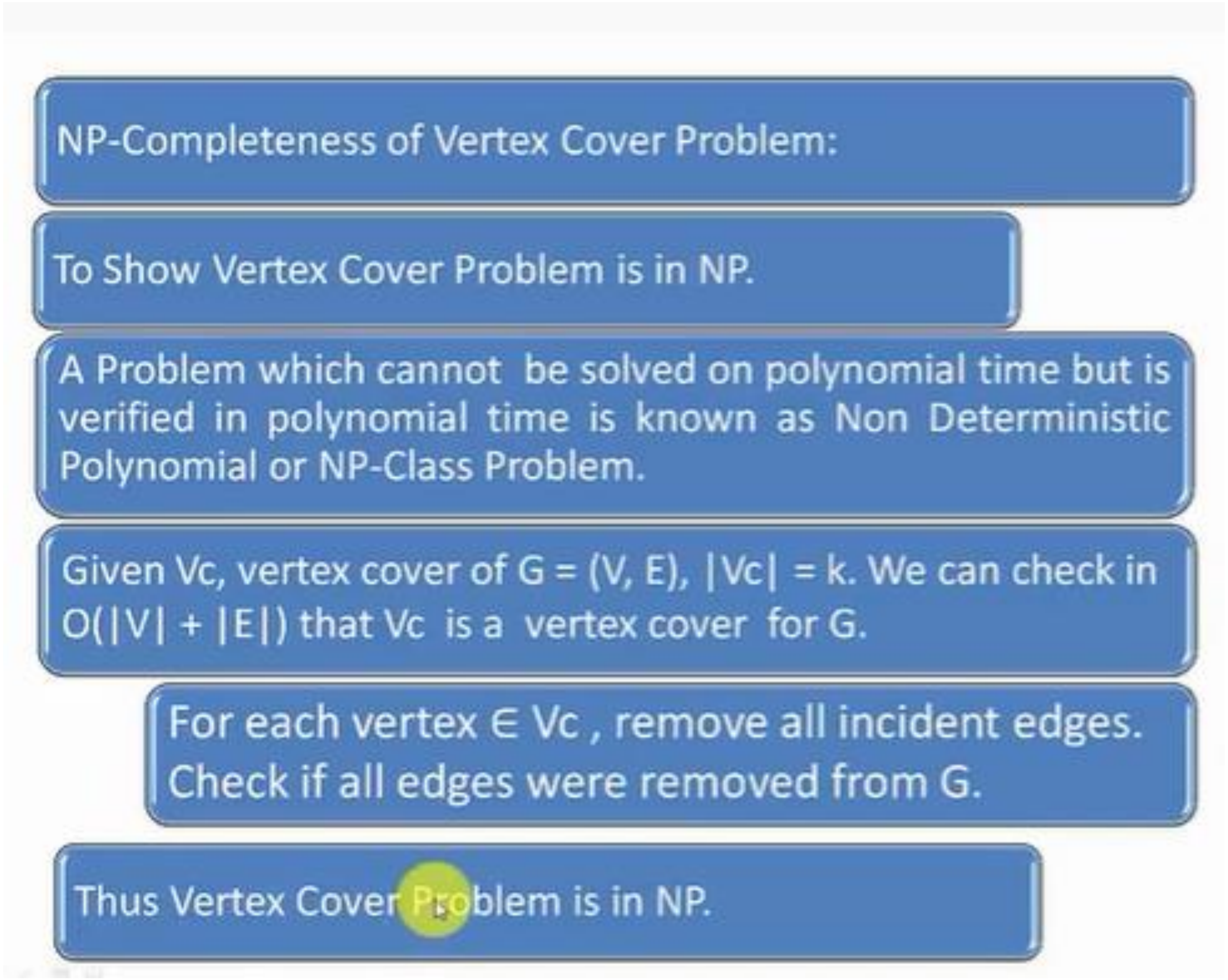
Step 1: Prove that B is in NP

Step 2: Select an NP-Complete Language A.

Step 3: Construct a function f that maps members of A to members of B.

Step 4: Show that x is in A iff f(x) is in B.

Step 5: Show that f can be computed in polynomial time.



## NP-Completeness of Vertex Cover Problem:

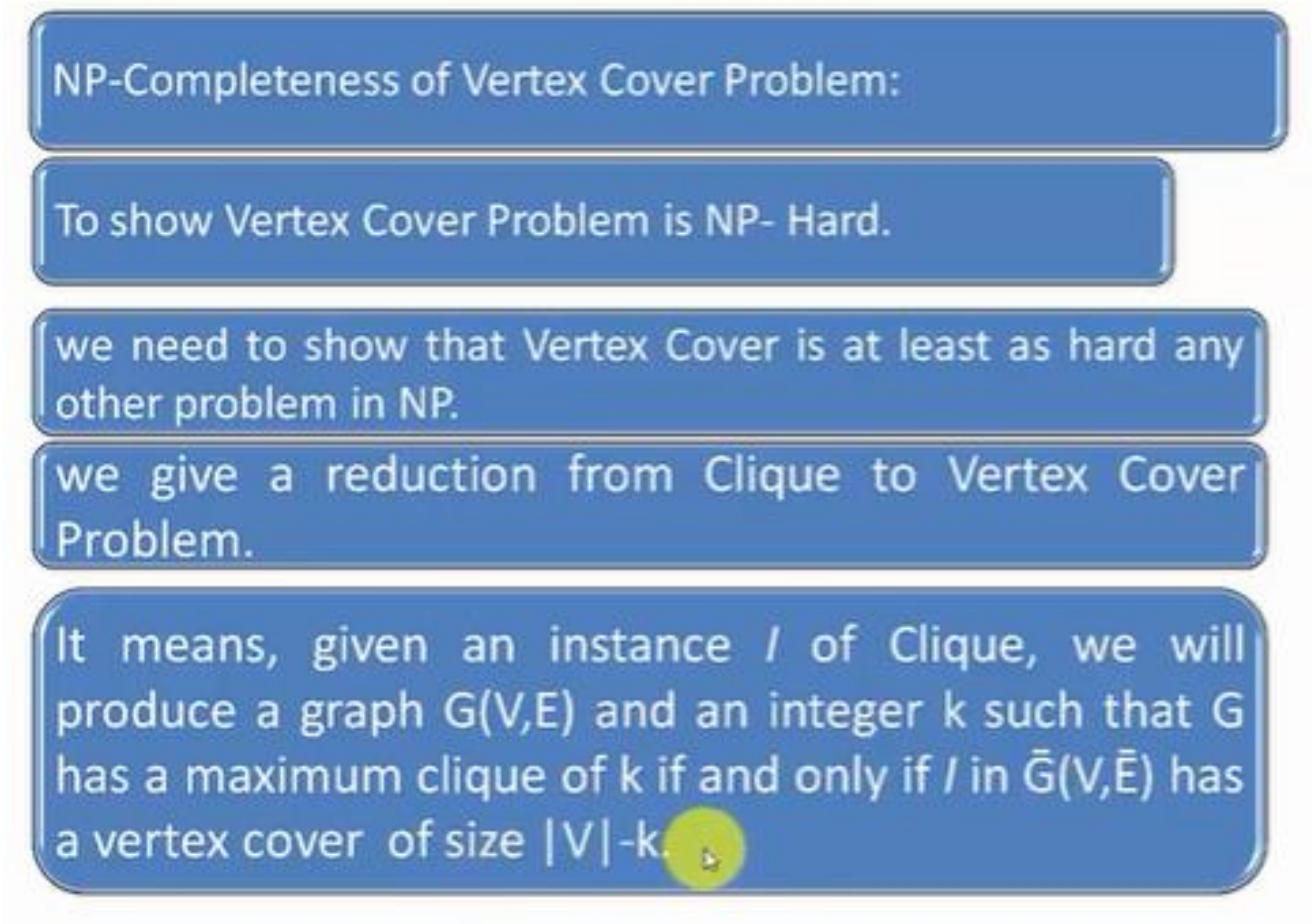
To Show Vertex Cover Problem is in NP.

A Problem which cannot be solved on polynomial time but is verified in polynomial time is known as Non Deterministic Polynomial or NP-Class Problem.

Given Vc, vertex cover of G = (V, E), |Vc| = k. We can check in O(|V| + |E|) that Vc is a vertex cover for G.

For each vertex ∈ Vc , remove all incident edges.
Check if all edges were removed from G.

Thus Vertex Cover Problem is in NP.


NP-Completeness of Vertex Cover Problem:

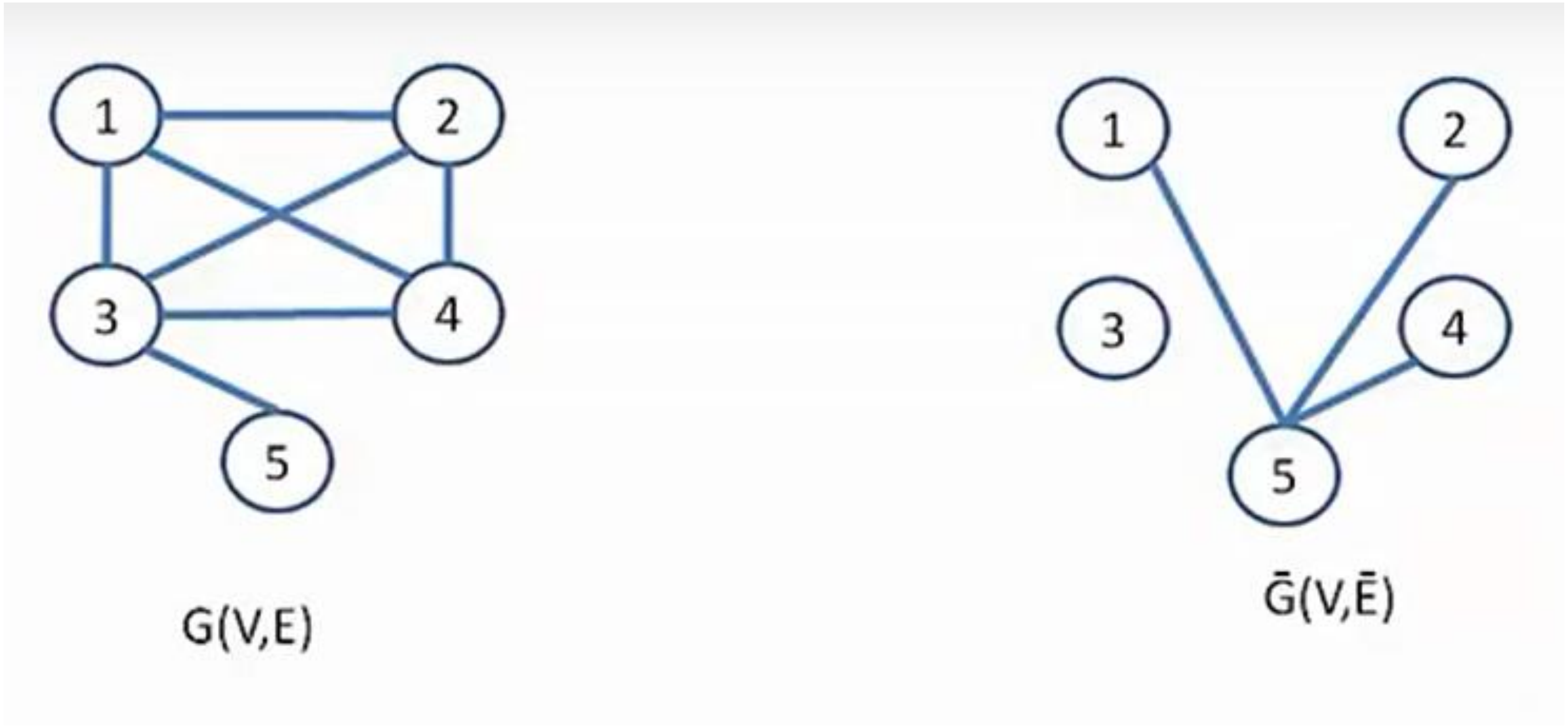
To show Vertex Cover Problem is NP- Hard.

we need to show that Vertex Cover is at least as hard any other problem in NP.

we give a reduction from Clique to Vertex Cover Problem.

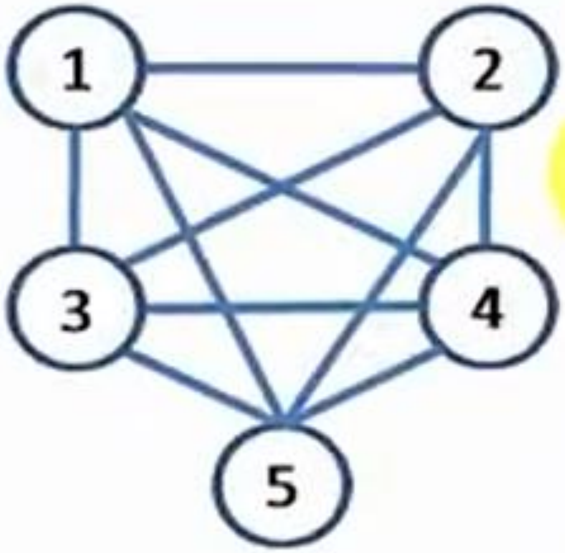
It means, given an instance *I* of Clique, we will produce a graph G(V,E) and an integer k such that G has a maximum clique of k if and only if *I* in $\bar{G}(V,\bar{E})$ has a vertex cover of size |V|-k.

G’ is the complement of G



When the above graphs are merged it will become a complete graph.

$G(V,E)$



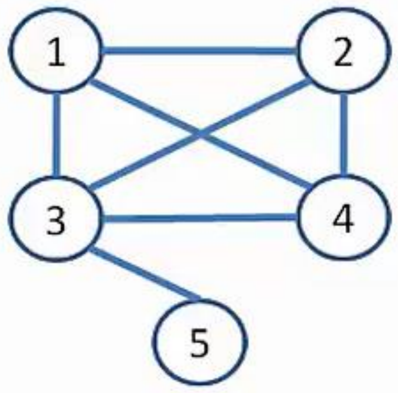
$\bar{G}(V,\bar{E})$

$G(V,E)$,
$|V|=n$,
Clique=k

Reduce

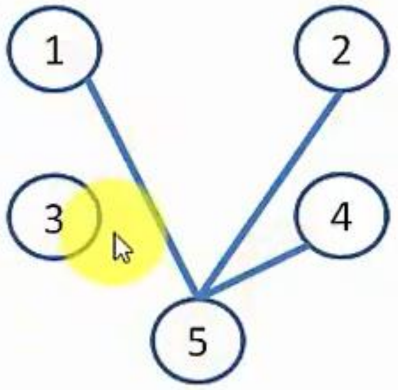
Polynomial Time

$\bar{G}(V,\bar{E})$
Vertex Cover
$= |V|-k$



$|V|=5$
Max. Clique(k)=4

Vertex Cover is
$|V|-k = 1$



Let G has clique V' of size k.

=> G has vertex cover of size |V|-k

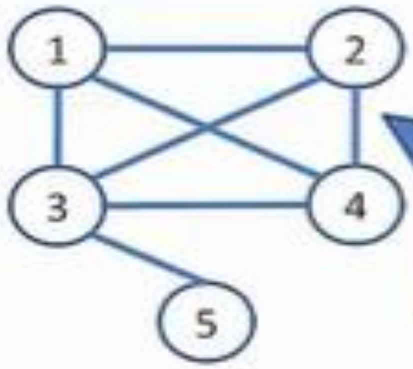
(a, b) ∈ E => (a, b) ∉ Ē

If (a, b) ∈ Ē, then at least a or b ∉ V' .

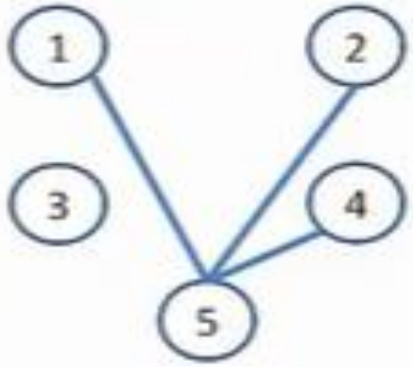
Every pair in V' is connected by an edge in E.

=> At least one of a or b is in V-V'

=> Edge (a,b) is covered by V-V'



V={1,2,3,4,5}
V'={1,2,3,4}
V-V'={5}



In the given example vertex (a,b)=(1,3) in E but vertex (1,3) not belongs to E’.

If (a,b)=(1,5) belongs to E’, but (1,5) not belongs to E.

But at least one a or b is in V-V’.

**Flow Networks and Flows**

Flow Network is a directed graph that is used for modelling material Flow. There are two different vertices; one is a **source** which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed. The flow of the material at any mark in the system is the rate at which the element moves.

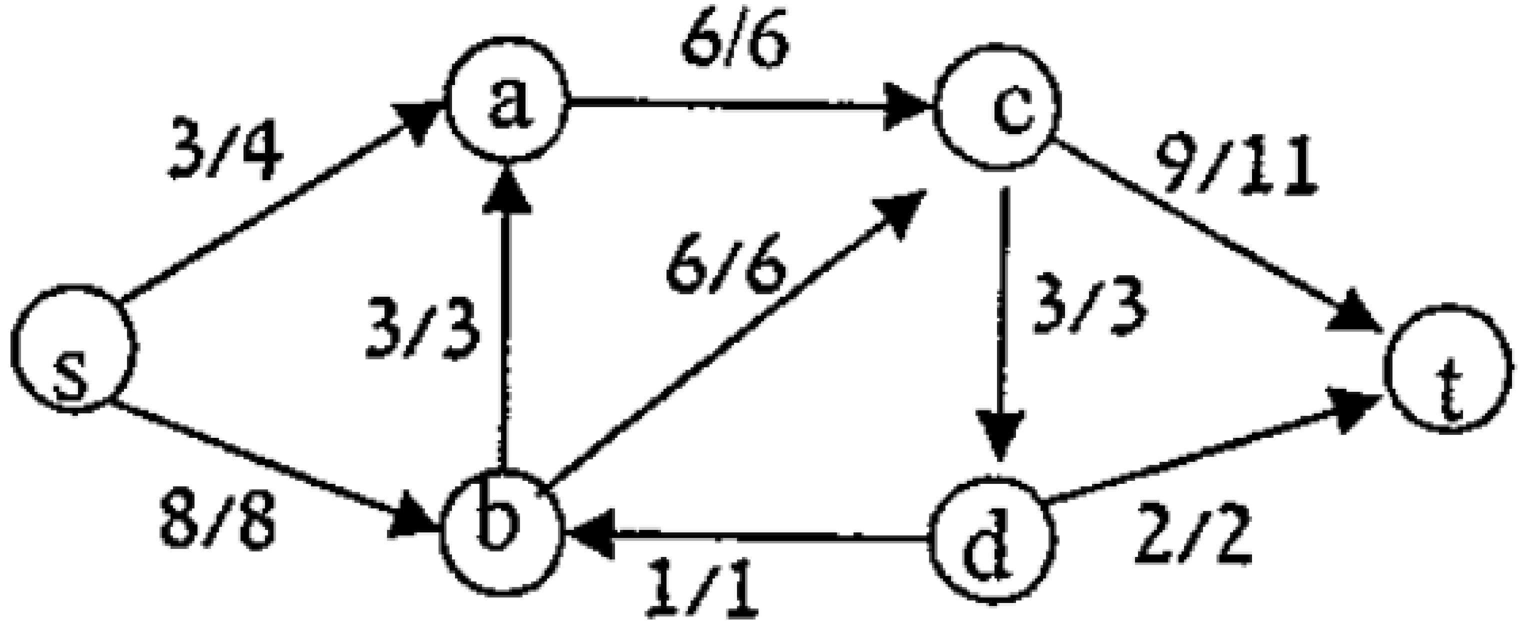
Some real-life problems like the flow of liquids through pipes, the current through wires and delivery of goods can be modelled using flow networks.

**Definition:** A Flow Network is a directed graph G = (V, E) such that

1. For each edge (u, v) ∈ E, we associate a nonnegative weight capacity **c(u, v) ≥ 0**. If (u, v) ∉ E, we assume that c (u, v) = 0.

2.There are two distinguishing points, the source s, and the sink t;

3.For every vertex v ∈ V, there is a path from s to t containing v.



Let G = (V, E) be a flow network. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function f: V x V→R such that the following properties hold:

- **Capacity Constraint:** For all u, v ∈ V, we need f (u, v) ≤ c (u, v). (Each edge (u,v) have a capacity given as c(u,v).

- **Skew Symmetry:** For all u, v ∈ V, we need f (u, v) = **-** f (u, v). (+ value shows the incoming-in and – value shows the outgoing-out)
- **Flow Conservation:** For all u ∈ V-{s, t}, (nodes except s and t) we need

$$\boxed{\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0}$$

(+ value shows the incoming-in and – value shows the outgoing-out – from the figure s to a =3 + s to b = 8, **3+8=11**, c to t =9 + c to d=2, **9+2=11)**

**( +3+8 and – 9+2 =0**)

The quantity f (u, v), which can be positive or negative, is known as the net flow from vertex u to vertex v. In the **maximum-flow problem**, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

The three properties can be described as follows:

1. **Capacity Constraint** makes sure that the flow through each edge is not greater than the capacity.
2. **Skew Symmetry** means that the flow from u to v is the negative of the flow from v to u.
3. **The flow-conservation** property says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of flow into a v is the same as the amount of flow out of v for every vertex v ∈ V - {s, t}

The value of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s, v)$$

The **positive net flow entering** a vertex v is described by

$$\sum_{\{u \in V : f(u,v) > 0\}} f(u, v)$$

The **positive net flow** leaving a vertex is described symmetrically. One interpretation of the Flow-Conservation Property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow f is said to be **integer-valued** if f (u, v) is an integer for all (u, v) ∈ E. Clearly, the value of the flow is an integer is an integer-valued flow.

## Ford Fulkerson method

The Ford-Fulkerson algorithm is used to **detect maximum flow from start vertex to sink vertex** in a given graph. In this graph, every edge has the capacity. Two vertices are provided named Source and Sink. The source vertex has all outward edge, no inward edge, and the sink will have all inward edge no outward edge. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems:

- Residual networks
- Augmenting paths

- Cuts

**Residual networks**

It is a graph which indicates additional possible follow. If there is such path from source to sink then there is a possibility to add flow denoted as $G_{f.}$

**Residual capacity** is original capacity of the edge minus flow that is denoted as $C_f$.

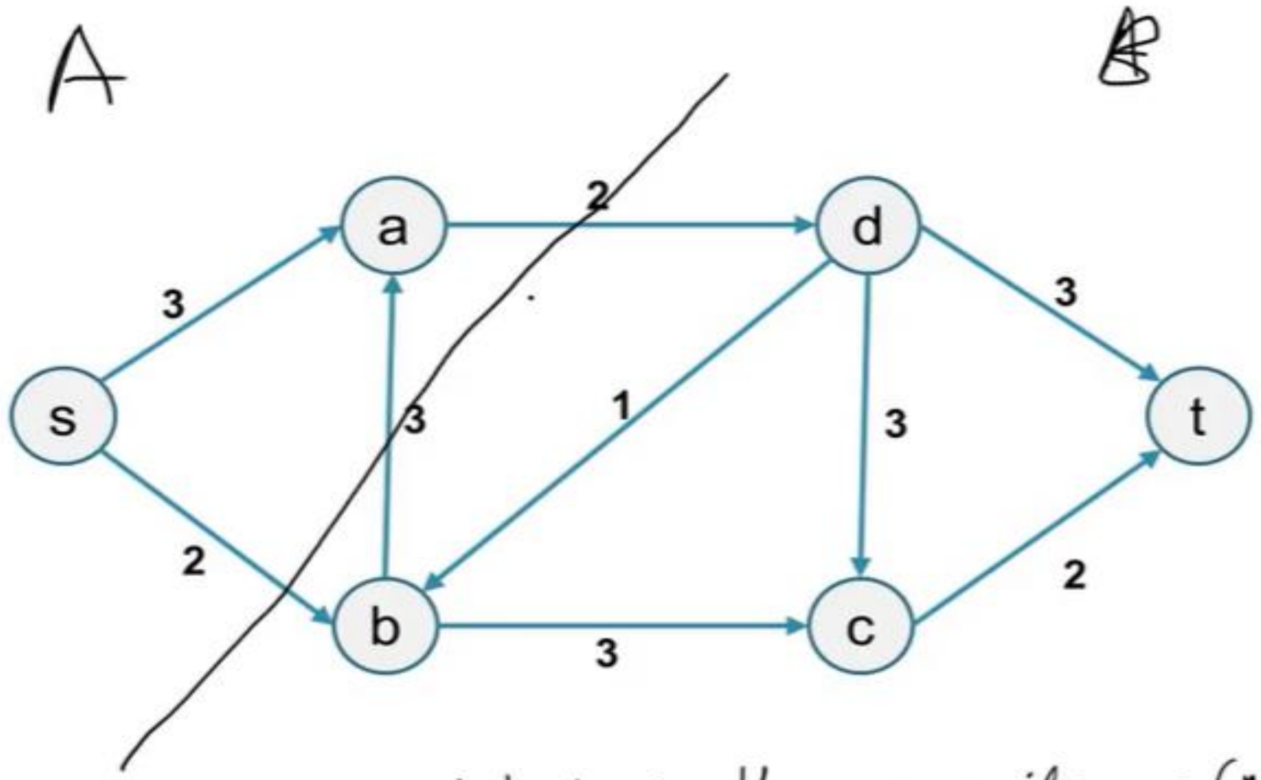
**Definition of a Cut**

A cut (also called as st-cut) is the division of graph into two partitions A and B such that some nodes are in partition A and some nodes in partition B and should satisfy the constraint that s must belongs to A while t must belongs to B.

The capacity of an s-t cut is defined by the sum of the capacity of each edge in the cut-set.

The max-flow min-cut theorem states that in a flow network, the amount of maximum flow is equal to capacity of the minimum cut.

## Example of Cut - 1



$A = \{s, a\}$

$B = \{b, d, c, t\}$

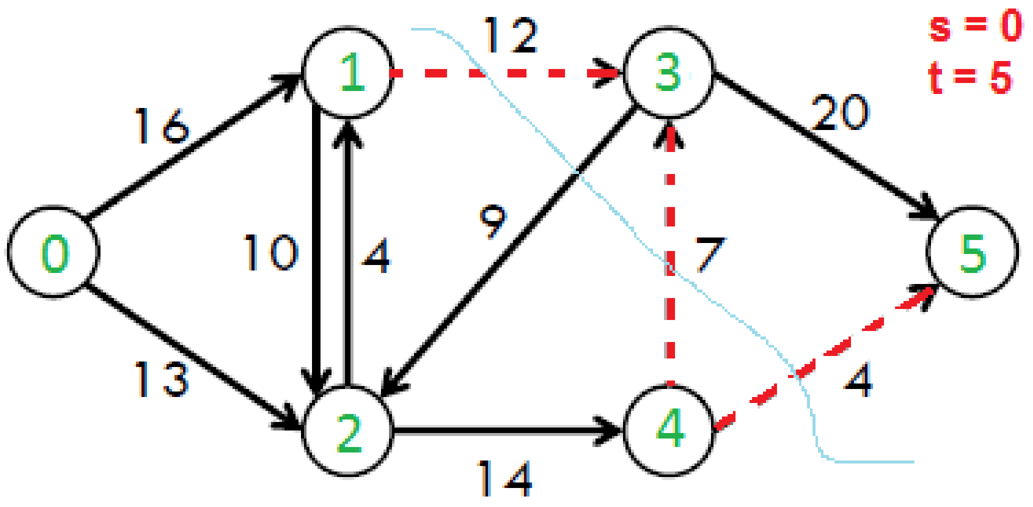
What is the capacity $c(A, B) = \sum_{e \text{ out of } A} c(e)$

$2 + 2 = 4$

Summation of all the capacities of the edges going out of the cut. ( s to b and a to d)

## Another example

In the following flow network, example s-t cuts are {{0 ,1}, {0, 2}}, {{0, 2}, {1, 2}, {1, 3}}, etc. The minimum s-t cut is {{1, 3}, {4, 3}, {4 5}} which has capacity as 12+7+4 = 23.



These ideas are essential to the important max-flow min-cut theorem

Simple Algorithm

1. Start with a initial flow as 0
2. While there is a augmenting path from source to sink

   Add two path flow to flow

3. Return flow

Bipartite matching