Indian Institute of Information Technology
Design and Manufacturing, Kancheepuram
Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 501 Advanced Data Structures and Algorithms

Instructor
N.Sadagopan
Scribe:
P.Renjith

# Theory of NP-completeness

Computational problems can be broadly classified into *solvable* and *unsolvable* problems. Solvable problems are problems that have an algorithm running in finite time whereas for unsolvable problems, every algorithm takes infinite time to complete. For example, printing all natural numbers, printing all real numbers, and halting problem are unsolvable. Alternately, solvable problems are those for which there exists an algorithm and for unsolvable problems no algorithm exists. Recall that, by definition, any algorithm must terminate after a finite amount of time.

Problems such as sorting and searching are solvable. Restricting our attention to solvable problems, we can think of many deterministic approaches to solve a given problem and it is natural to pick the best (efficient) among different approaches possible. As far as efficiency is concerned, polynomial-time solutions are of practical interest. That is, algorithms running in time $O(n^k), k \in Z^+$, where $n$ is the input size. A natural question is whether all solvable problems have polynomial-time solutions? Interestingly, enumeration problems such as enumerating all possible spanning trees of an arbitrary graph of size $n$, enumeration of all subsets of a set of size $n$ etc., have exponential-time complexity. The former incurs $\Omega(n^{n-2})$ time-effort and the latter incurs $\Omega(2^n)$ time-effort. This shows that not all solvable problems have polynomial-time solutions.

This brings a classification among solvable computational problems, namely *tractable* and *intractable* problems. A problem is tractable if there exists a polynomial-time algorithm. On the other hand, intractable problems on input size $n$ have $\Omega(c^n), c > 1$ algorithms. I.e., every algorithm for such problems incurs $\Omega(c^n)$. An immediate and a natural question at this context is that can every solvable problem be classified into tractable and intractable classes. Interestingly, there are computational problems like travelling salesperson problem, maximum independent set problem, maximum clique problem, minimum vertex cover problem, minimum steiner tree problem, etc., for which, as of this date, there is no known polynomial-time algorithm, nor do they have exponential-time lower bound. That is, such problems cannot be included either in tractable or intractable class. We call such problems as HARD problems.

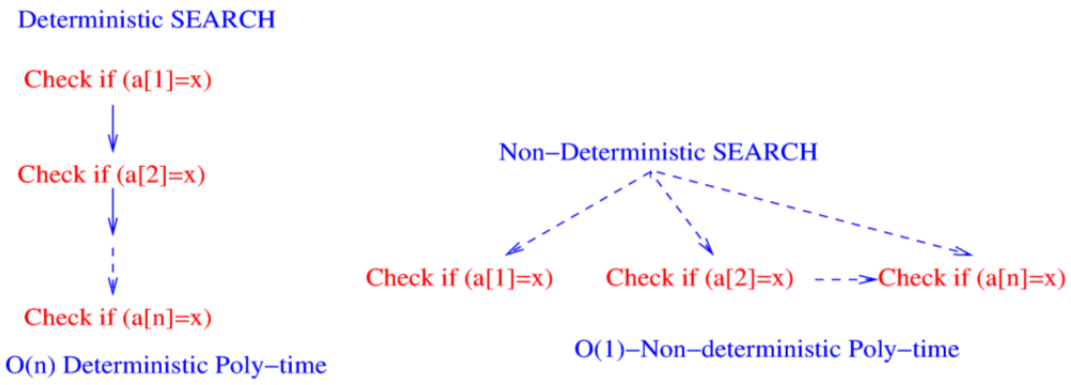
Classical algorithms in practice follow deterministic computational model and the underlying machine is a deterministic machine or sequential computer. That is, these algorithm make exactly one move at each step of the computation. For example, linear search, compares $x$ with $A[1]$ in Step 1 and compares $x$ with $A[2]$ and so on. Further, the underlying computational graph appears like a path. This is the case with all deterministic algorithms irrespective of whether it runs in polynomial time or exponential time complexity.

To cope up with HARD problem and to understand the inherent difficulty of the problem, we ask; can the computational model be altered so that it is different from the traditional deterministic computations? Motivated by this line of research, we introduce the non-deterministic model in which the computational graph is a tree instead of a path. The non-deterministic computation takes a non-deterministic move (guess) at each step of the computation. Unlike deterministic model, non-deterministic model guesses the solution in non-deterministic fashion. Towards this attempt, the guessing machine branches to multiple partial solutions at each move. That is, it guesses the (partial) solution at each step, the guess is always correct and finally a correct solution is obtained at one of the leaves of the computational tree. The sequence of correct guesses constitute a solution which is a path in the associated tree. We can see the non-deterministic model as a powerful model that is capable of computing more tasks at once and produces the correct solution if the input has one such solution. It is important to note that non-deterministic machine is a hypothetical

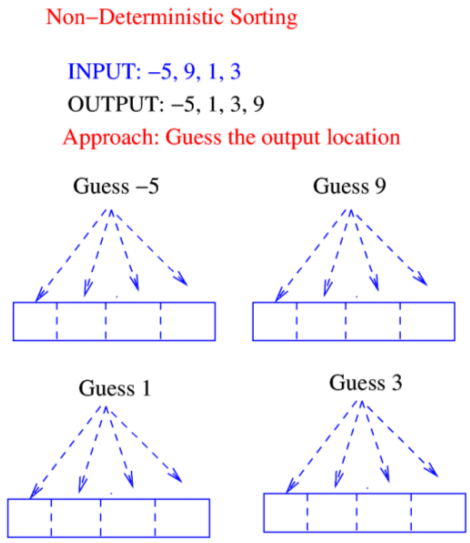
machine (cannot be realized in practice) and the objective of introducing this machine is to understand the computational complexity of the problem in hand. Questions such as how difficult the problem is over other problems, the inherent complexity of the problem, etc., can be answered better under this model.

Let us revisit tractable (intractable) problems. In particular, tractable problems are problems that have polynomial-time algorithms under a deterministic computational model and intractable problems have exponential-time algorithms under a deterministic computational model. To cope-up with HARD problems, we shall introduce non-deterministic model and ask, do HARD problems have polynomial-time algorithms under this variant. This line of study was introduced by researchers Stephen Cook and Leonid Levin. Subsequently, they introduced complexity classes P and NP. The complexity class P contains the set of problems that are solvable in deterministic polynomial time and the complexity class NP contains the set of problems that are solvable in non-deterministic polynomial time. Classical problems such as minimum spanning tree and shortest path belong to class P. Strictly speaking, both P and NP are defined only for decision problems which we shall introduce in the next section. We shall see next some examples of non-deterministic computations.

**Example 1:** Non-deterministic Linear search. Input: Array $A$, key $x$. Check whether $x \in A$.
In a deterministic linear search, the elements in the array are searched one after the other, and a successful match is acknowledged. In the non-deterministic computation model, the computational graph resembles a star like tree. The non-deterministic machine simultaneously checks whether $x$ is present in $A[1]$ or $A[2]$ and so on. In some sense, it guesses the location $i$ for which $x == A[i]$. The location of the key $x$, if present is returned by the non-deterministic search (guessing the solution) in constant time.




**Example 2:** Non-deterministic Sorting
Note: The input array is in $A[1..n]$ and output array is in $B[1..n]$.
The non-deterministic sorting algorithm guesses for each $A[i]$ the position in $B[\,]$ array. It correctly guesses for each element in $A[\,]$, and in total, the model makes $O(n)$ guesses. Therefore, the non-deterministic sorting is performed in linear time. Note that the non-deterministic machine, in the process of guessing, explores all $n!$ permutations of the input.




**Example 3:**
Hamiltonian cycle problem
Input:-Graph $G$
Question: Does $G$ have a Hamiltonian cycle?
A trivial deterministic algorithm for this problem generates all the permutations of the vertices of $G$ and checks whether a permutation forms a spanning cycle. Thus, this approach incurs $O(n!)$. In the non-

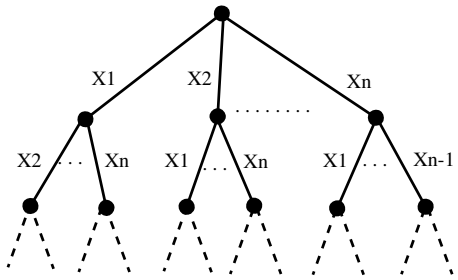



Figure 1: Non-Deterministic computation graph: Hamiltonian cycle problem

deterministic model, it guesses the first vertex, followed by the second, and so on. A computational graph is shown in Figure.1. If the input graph is an yes instance of Hamiltonian cycle problem then the guessing part guesses the right path in the computational tree. Further, the path gives one of the permutations which is available at the leaf node. Interestingly, this guessing can be done in linear time as there are $n$ guesses, one per iteration. Checking whether a permutation is a spanning cycle or not can be done using a deterministic polynomial-time algorithm. Therefore, there exists a non-deterministic polynomial-time algorithm to solve this problem and hence it is in NP.

**Example 4:**
Boolean Satisfiability
Input: Boolean formula $F$ on variables $x_1, \ldots, x_n$
Question: Is $F$ satisfiable?
For example, for the variables $\{x_1, x_2, x_3, x_4\}$ and $F = (x_1 \vee \bar{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x_4}) \wedge (x_2 \vee x_3 \vee x_4)$, find a truth assignment for $\{x_1, x_2, x_3, x_4\}$ such that $F(\{x_1, x_2, x_3, x_4\})$ is evaluated to be true. A trivial deterministic algorithm try all $2^n$ possibilities and thus runs in $\Omega(2^n)$ time. For the non-deterministic approach,

it correctly guesses the truth assignment for each variable $x_i, 1 \leq i \leq n$, as shown in the Figure 2. Note that in the computational graph, each leaf corresponds to a truth assignment for the variables $x_1, \ldots, x_n$. Since the guess at each step made by the non-deterministic model is always correct, the path chosen results in an assignment which is in turn a satisfiable assignment if $F$ has one such assignment. Also there exists

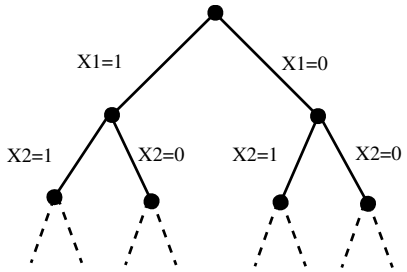



Figure 2: Non-Deterministic computation graph: Satisfiability problem

a deterministic polynomial-time algorithm to check the validity of the truth assignment in polynomial time. Therefore, 3-SAT is in NP.

**Example 5:**
Input: Graph $G$ with the vertex set $V(G) = \{v_1, \ldots, v_n\}$ and the edge set $E(G)$
Question: Find a Minimum Vertex Cover. A set $S \subset V(G)$ is called a vertex cover of $G$ if for every edge $uv \in E(G)$ either $u$ or $v$ or both are in $S$.
In a deterministic approach, one has to find all subsets which are possible candidates for a minimum vertex cover, and check whether the subset is a valid vertex cover or not. Note that this method incurs exponential effort. In a non-deterministic approach, it guesses the vertices one after the other which are part of a minimum vertex cover. To find a minimum vertex cover one has to check all the leaves and finally output the minimum one. In this process, even though the model is a non-deterministic one, the polynomial time is not achieved due to the exponential number of leaves. One way to overcome this situation is to increase the power of non-deterministic machine so that it guesses a minimum vertex cover in polynomial time. One may wonder, how much power can be given to non-deterministic machine given the fact that these machines are hypothetical.
**How powerful non-deterministic machines are ?**

- For linear search, non-deterministic machine guesses the location of $x$ in $O(1)$ time. Similarly, can the machine guess Hamiltonian cycle, truth assignment to a boolean formula in $O(1)$ time ? Can a non-deterministic machine guess the minimum in minimum vertex cover in $O(1)$ or $O(n)$ time ?
- The objective of introducing non-determinism is to understand the inherent complexity of the problem given that classical problems such as Hamiltonian, vertex cover and 3-SAT have only trivial deterministic algorithms as of this date.
- To understand the inherent complexity of the problem, one must give minimal power to the machine and perform maximum computational subtasks associated with the problem using polynomial time deterministic algorithms. Due to this, except guessing a vertex or a value to the literal (in 3-SAT), all other checks done at the leaves such as verifying the solution, whether the solution is minimum/maximum must be done in deterministic polynomial time.
- In some sense, guessing part implicitly brings enumeration of all subsets, all permutations, etc. This also gives an indication that if at all we convert a non-deterministic machine into an equivalent deterministic machine, then such enumeration task is inevitable. Most importantly, it is enumeration that makes these problems difficult.

We shall next given an example of a decision problem and show that how decision and optimization problems are related.

`Decision Vertex Cover:`

`Input:` Graph $G$, Integer $k$
`Question:` Find a vertex cover of size at most $k$.

Note that the optimization version (minimum vertex cover) can be solved by making a polynomial number (here linear) of invocations to the decision version of minimum vertex cover problem. Our objective is to determine the value of minimum using decision algorithm as a black box. To accomplish this task, we call decision algorithm with the input $(G, k = 1)$, i.e., ask for a vertex cover of size 1 in the first invocation. If the algorithm returns NO, then call the algorithm again with the input $(G, k = 2)$. We continue this process by incrementing the value $k$ by one at each iteration till we get YES. It can be observed that the there exists $k$ for which the decision algorithm gives `YES` and NO until $k - 1$. Further, the algorithm will say YES for $k+1, k+2, ....$ From this it can be easily inferred that the minimum vertex cover is of size $k$.

Moreover, using this decision version, note that the overhead involved in scanning the entire leaves for finding the minimum is not needed. Instead, once the correct guess returns a subset of vertices $S$ with at most $k$, we can easily check whether $S$ is a vertex cover or not in deterministic polynomial time. Therefore, minimum vertex cover is in NP.

**Some observations:**

1. A deterministic computing model is analogous to modern sequential computer. Similarly, an analogy to the non-deterministic computational model is a parallel computer that has an array of processors operating in parallel. Note that the run time of non-deterministic algorithms cannot be compared with that of the deterministic algorithms as they differ in the underlying computational model.
2. Since optimization version of vertex cover can be solved using decision version of vertex cover as a subroutine (black box), it is sufficient to work with decision version.
3. Problems such as 3-SAT, Hamiltonicity are by definition decision problems and every optimization problem has an equivalent decision version, this shows that the set of decision problems is larger than the set of optimization problems.
4. While attempting 'black-box technique' to solve optimization problem using decision algorithm as a black box; one must find (i) the value of minimum/maximum using decision box (ii) the solution set using decision box.

# Class P and NP

Since optimization problems have equivalent decision problems, we shall focus on decision problems and let us define the complexity classes restricted to decision problems.

P={ X : X is a decision problem and X is solvable in deterministic polynomial time }

NP={ X : X is a decision problem and X is solvable in non-deterministic polynomial time }

Recall that the computational model NP guesses the solution and the solution is further checked using a deterministic polynomial-time algorithm. For example, for Hamiltonian cycle problem, after making

$O(n)$ guesses, the algorithm checks whether the guess is indeed a Hamiltonian cycle using a deterministic polynomial-time verification algorithm. We have no clue on how the guessing is done, even though the model ensures a correct solution if it has one. Therefore, we focus only on the verification algorithm and present an alternative definition for NP. This brings a more easier method to check whether a given problem is in NP or not;

NP={ X : there exists a deterministic polynomial-time algorithm to verify an instance of X}

According to the above definition, the input to verification algorithm consists of parameters such as a graph (formula), a certificate, an integer. For example, for vertex cover
`Input:` Graph $G$, $S = (v_7, v_1, v_4)$, $k = 4$
`Question:` Is $S$ a vertex cover of size at most 4?

Similarly,
Hamiltonian problem:

`Input:` Graph $G$, $S = (v_7, v_1, v_4, \ldots, v_n, v_2, \ldots, v_7)$
`Question:` Is $S$ a Hamiltonian cycle?

The verification algorithm proceeds as follows; check whether there exists an edge between the adjacent vertices in the given permutation, i.e., $v_7v_1, v_1v_4 \in E(G)$, and so on. Clearly, such a check can be done in deterministic polynomial time. Similarly, is the case for problems like travelling salesman problem, independent set of size at most $k$, clique of size at most $k$, etc. Note that all decision problems which are solvable in deterministic polynomial time can also be verified in polynomial time. By the above definition, all problems in class P is also in NP. As a result, P $\subseteq$ NP. It is an open question whether P is a strict subset of NP.

**Remark:** Both definitions of NP are equivalent. In the first definition, after guessing, the verification component runs in deterministic polynomial time. For the verification routine, the guessed subset (permutation) is given as part of the input.

**Some more interesting problems in class NP**
SAT, 3-SAT, 3-colorability, Travelling Salesman problem, Hamiltonian Cycle, Hamiltonian Path, Minimum Steiner tree, Minimum Dominating set, Minimum connected dominating set, Minimum Vertex Cover, Maximum Independent Set, Maximum Clique, Puzzles like Sudoku, Minesweeper.

Class NP consists of thousands of combinatorial problems, and the set NP is still growing in size. Surprisingly, no concrete progress has been made to show P=NP or P $\neq$ NP. While this being the status of NP one side, researchers Cook and Levin focused on identifying 'Hardest' problems in NP. A problem which is at least as difficult as every other problem in NP. In the next session, we shall see how to categorize such problems.

## Reducibility between Combinatorial problems

In the classical world, 'sorting problem' can be solved using 'problem of finding minimum in an array' as a black box. That is, by making $n$ repeated calls to 'find min' black box, we get, first min, second min, and so on upto $n^{th}$ min, which is precisely the sorting sequence of an array of $n$ numbers. Similarly, 'sorting problem' can be solved using 'in-order traversal' as a black box. The input array is converted into a binary search tree which is in turn passed as an input to in-order traversal algorithm to get a sorting sequence. The technique of solving problem A using problem B as a black box is known as reducibility between combinatorial problems. Since problem B is polynomial-time solvable (EASY), problem A is also polynomial-time solvable (EASY). Moreover, we make polynomial number of calls to problem B (in general the black box)

to solve problem A. We denote this reduction as $A \leq^p B$. Since $B$ is easy, implies that $A$ is easy. The contrapositive of the above implication says that, if $A$ is HARD then $B$ is also HARD. We shall work with the contrapositive as our goal is to understand the hardness of NP problems.

For decision problems $X$ and $Y$, $X$ is reduced to $Y$ if the following holds. An instance of $X$ is polynomially reduced to an instance of $Y$ and it must be a solution preserving reduction. That is if $X$ is an YES instance ($X$ has a solution) then $Y$ must also be YES instance and vice versa. It is clear that if $X$ is a NO instance, then $Y$ must also be a NO instance. We denote the reduction as $X \leq^p Y$. It follows that to solve the problem $X$, we can use problem $Y$ as a black box. Moreover, if the problem $Y$ is polynomial-time solvable and since the reduction is in polynomial time, then $X$ is also polynomial-time solvable.

That is, for every instance of problem $X$, we reduce it to $Y$, solve $Y$, and the solution to $Y$ is used

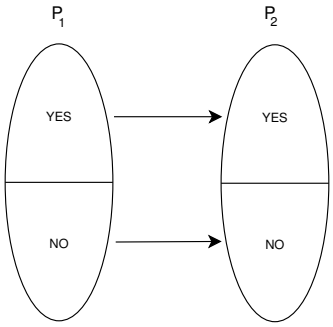



Figure 3: $X \leq^p Y$

to obtain a solution to $X$. Therefore, the problem $X$ can also be solved in polynomial time. Note that the reduction is a function that maps instances of $X$ to instances of $Y$. Each element in $X$ has an image in $Y$ by the reduction and the converse is not necessarily true. That is, there may exist instances in $Y$ which do not correspond to any instance of $X$. The image of $X$ may be a subset of instances of $Y$.

Reductions are useful in comparing the hardness between problems. That is, if $X$ is hard, then $X \leq^p Y$, implies that the problem $Y$ is at least as hard as problem $X$.

The class **NP-Hard** is a class of problems (not necessarily in NP) that are at least as hard as each problem in the class NP. Therefore to show that a problem $Y$ is in the class NP-Hard, we have to show a reduction from all the problems in the class NP to $Y$, $X \leq^p Y$, for every $X$ in NP.

`The first NP-Hard problem:` Stephen Cook and Leonid Levin showed that circuit satisfiability, boolean satisfiability, and 3-SAT are NP-Hard. Using fundamental principle, they proved that there exists a polynomial-time reduction from every problem in NP to 3-SAT. Therefore, 3-SAT is NP-Hard. Note that, if we show a polynomial-time reduction from 3-SAT to a problem $Y$, then due to transitivity property, there exists a polynomial-time reduction from all the problems in NP to $Y$ through 3-SAT as a bridge. That is, $X \leq^p$ 3-SAT $\leq^p Y$, for every problem $X$ in NP. It follows from the definition that $Y$ is NP-Hard.

**Note:** To show a problem $Y$ is NP-Hard, it is sufficient to choose a known NP-Hard problem and show a polynomial-time reduction to $Y$. The reduction does not say anything about the NP status of problem $Y$.

**MAX-Clique is NP-Hard**

`Input:` Graph $G$ with the vertex set $V(G) = \{v_1, \ldots, v_n\}$ and the edge set $E(G)$, Integer $k$
`Question:` Does there exist a clique of size at least $k$. A clique is a set of mutually adjacent vertices in $G$.

*Claim:* Clique problem is NP-Hard
*Proof:* We shall present a polynomial-time reduction from 3-SAT to clique. An instance of a 3-SAT consists of clauses $C = \{C_1, \ldots, C_m\}$ and variables $x_1, \ldots, x_n$, is reduced to the clique problem as follows: corresponding to each literal (variables or their negations) $x_i \in C_j$, $1 \leq i \leq n, 1 \leq j \leq m$, there exists a vertex in graph $G$. $E(G) = \{x_i x_j : x_j \neq \overline{x_i}$ and $x_i, x_j$ are in different clauses $\}$. We shall see that there exists a satisfying truth assignment for 3-SAT if and only if there exists a clique of size $m$ in $G$.

For *if* part, since there are $m$ clauses, each clause should be evaluated to 1 in any truth assignment. For this to happen, there exists a literal assigned 1 in each clause. Let $U = u_1, \ldots, u_m$ be the literals such that $u_i$ is the literal assigned 1 in the clause $C_i$. Clearly, there cannot be $u_i, u_j \in U$ such that $u_j = \overline{u_i}$. This implies that the vertices corresponding to the literals in $U$ are mutually adjacent as no two literals in $U$ belong to the same clause. Therefore, there exists a clique of size $m$ in $G$.

For *only if* part, consider a clique of size $m$ in the reduced graph $G$. Let the clique $K$ be induced on vertices $v_1, \ldots, v_m$. Since there are no edges between the literals of a clause, there are no two vertices in $K$ which corresponds to two literals of same clause. Also observe that there are no two vertices $v_i v_j$ in $K$ with corresponding literals $x_i, x_j$ such that $x_j = \overline{x_i}$. This implies that the literals $u_i, \ldots, u_m$ corresponding to the vertices in clique $K$ can be assigned 1 to get a satisfiable assignment in the 3-SAT formula. Since 3-SAT is NP-Hard and from the above polynomial-time reduction (from 3-SAT to clique problem), it follows that clique problem is NP-Hard. □

**MAX-Independent set is NP-Hard**
`Input:` Graph, $G$, Integer $k$
`Question:` Does there exist an independent set of size at least $k$. An independent set is a set of mutually disjoint vertices in $G$.

*Claim:* Independent set problem is NP-Hard.
We reduce MAX-clique to MAX-Independent set. Finding a clique of size $k$ in $G$ is reduced in polynomial time to finding an independent set problem in the complement graph of $G$. That is,

$$(G, k) \leq^p (\overline{G}, k)$$

The complement graph $\overline{G}$ of $G$ is such that $V(\overline{G}) = V(G)$ and $uv \in E(\overline{G})$ if and only if $uv \notin E(G)$. A clique of size $k$ in $G$ is mapped to an independent set of size $k$ in $\overline{G}$. It is clear that $(G, k)$ if and only if $(\overline{G}, k)$. That is, any clique of size $k$ in $G$ will be an independent set of size $k$ in $\overline{G}$ by definition. Similarly, any independent set of size $k$ in $\overline{G}$ is a clique of size $k$ in $G$. Therefore, the independent set problem is NP-Hard.

**MIN-Vertex cover is NP-Hard**
We reduce an instance of MAX-Independent set to an instance of MIN-Vertex cover as follows;
Maximum independent set of size $k$ in $G$ is reduced to minimum vertex cover of size $n - k$ in $G$.

$$(G, k) \leq^p (G, n - k)$$

That is, $S$ is a minimum vertex cover in $G$ if and only if $G \backslash S$ is a maximum independent set in $G$. This implies that vertex cover problem is NP-Hard.

**Note:** Circuit-SAT $\leq^p$ 3-SAT $\leq^p$ Clique $\leq^p$ IS $\leq^p$ VC.

## NP-Complete Problems

NP-Complete problems are the hardest problems in NP.
Problem $X$ is NP Complete if
$X \in$ NP and
$X$ is NP-Hard
Since there exists a polynomial-time verification algorithm for the independent set, clique, vertex cover problems, they are all in NP, and from the reductions presented above, it follows from the definition that independent set, clique, and vertex cover problems are NP-complete.

**Some interesting Observations:**

1. In $X \leq^p Y$, if $Y \in NP$, then $X \in NP$.
2. If an NP-Hard problem $X$ belongs to class P then, every problem in NP has a deterministic polynomial-time algorithm; therefore, P=NP.
3. If an NP-Hard problem $X$ has a lower bound argument that says, $X$ is solvable in $\Omega(c^n), c > 1$, then $P \neq NP$.
4. To show a problem $Z$ is NP-Hard, it is sufficient to present just one reduction from a known NP-Hard problem $X$ to $Z$. That is, $X \leq^p Z$ would suffice.
5. Reductions such as 3-SAT $\leq^p$ (minimum spanning tree) MST or 3-SAT $\leq^p$ (shortest path) SPATH imply that P=NP.
6. Reductions such as SPATH $\leq^p$ 3-SAT has no significance. An easy problem is reduced to a difficult problem. Independent of the complexity, if the reduction exists, then it is a valid reduction respecting the definition.
7. Since NP-Hardness focuses on the hardness, all unsolvable problems are also NP-Hard. Any unsolvable problem is at least as hard as any solvable problem. Problems such as Halting problem and post-correspondence problem have polynomial-time reductions from 3-SAT and hence, they are NP-Hard. However, these problems are not in NP. This indicates that, NP-Hard result does not imply the NP status.
8. Problems such as listing all subsets, listing all spanning trees are at least as hard as any problem in NP, therefore, they are NP-Hard. Since, their solution cannot be verified in deterministic polynomial-time, such enumeration problems are not in NP.

**Acknowledgements:** Lecture contents presented in this module and subsequent modules are based on the text books mentioned at the reference and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thank all of them.

**References:**

1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.