Indian Association for the Cultivation of Science

(Deemed to be University under *de novo* Category)

Master's/Integrated Master's-PhD Program/ Integrated Bachelor's-Master's Program/PhD Course

Theory of Computation II: COM 5108

Lecture II

Instructor: Goutam Biswas *Autumn Semester 2025*

1 P, NP, NP-completeness

The class **P** and **NP** and the question of equality or inequality of them is the *holy grail* of Complexity Theory. The historical definitions of these two classes are based on two models of Turing machine and their decision problems. But there are other interesting ways of looking at the **P** versus **NP** question. There are also similar complexity classes and similar question related to search problems.

Often it is more difficult to find the solution of a problem than to check the correctness of a given solution. The class **P** represents the collection of *efficiently solvable* (in polynomial time) decision problems. On the other hand the class **NP** is the class of decision problems for which the correctness of the given *proof* of the answer can be *checked efficiently* (in polynomial time).

In case of *decision problems* we try to find the *membership* of an element in a set. **P** is the collection of decision problems for which the membership can be tested *efficiently* (in polynomial time). Whereas **NP** is the collection of decision problems where the *proof* of membership, if presented, can be verified *efficiently* (in polynomial time).

1.1 Search Problems

Search problems are more common in daily life and computing. A few examples are solution of a set equations, factors of an integer, spanning tree of a graph, satisfying assignment of a Boolean expression etc. In case of search problems a question similar to **P** versus **NP** comes up. The class of problems for which the search can be performed efficiently and the class of problems for which a given solution can verified efficiently for its correctness. These are the counterparts of **P** and **NP**.

If the output of a search problem is very long compared to the size of the input, the solution cannot be generated efficiently. The computation time in this case is determined by the length of the output. We consider search problems where the output length is bounded by $poly(|x|)$, where x is the input.

1.1.1 Search Problems: PF versus PC

A search problem may be viewed as a binary relation over $\{0,1\}^*$. For efficient search problems we consider polynomial bounded relations.

The class **PF** stands for the collection of search problems whose solution can be found in polynomial time and **PC** stands for the collection of search problems whose solution can be verified in polynomial time.

Definition 1. A search problem S is a binary relation over $\{0,1\}^*$ i.e. $S \subseteq \{0,1\}^* \times \{0,1\}^*$. On input x, $S(x) = \{y : (x,y) \in S\}$ is the set of solutions of x. The set $S(x)$ may be empty. Solving a search problem may be viewed as computation of a function $f_S : \{0,1\}^* \to \{0.1\}^* \cup \{\perp\}$ so that

$$\begin{cases} f_S(x) \in S(x) & \text{if } S(x) \neq \emptyset \\ f_S(x) = \perp & \text{if } S(x) = \emptyset. \end{cases}$$

We also assume that S is polynomial bounded for efficient search problems. That is if $(x,y) \in S$, then $|y| \leq poly(|x|)$.

Definition 2. A search problem corresponding to a polynomial bounded relation $S \subseteq \{0,1\}^* \times \{0,1\}^*$ is efficiently solvable if there is a polynomial time algorithm M such that for each $x \in \{0,1\}^*$, $M(x) \in S(x)$ if and only if $S(x) \neq \emptyset$. Otherwise the algorithm halts indicating that there is no solution i.e. $M(x) = \perp$.

The collection of efficiently solvable search problems belong to the class **PF**[1]

Examples of such problems are finding spanning tree of a graph, finding the sorted permutation of a given list from an ordered data set, finding the two colouring of a graph etc. But there are also problems whose computation status are unknown[2] e.g. finding factors of an integer, finding satisfying assignment of a Boolean expression, finding a 3-colouring of a graph etc.

Another class of natural problems has the property that given an input x and a suggested solution y, it can be efficiently verified whether y is indeed a solution of the problem instance x. We call the class as **PC**[3]. A few examples are as follows: given an integer n and a set of integers y as its factors, it is easy to verify the truth of the claim. Given a Boolean expression and a set of assignments, it is easy to verify whether the expression is satisfied by the assignment. Given a graph and a possible 3-colouring of its vertices, the validity of the claim can be verified efficiently. All these verifications can be done in polynomial time.

Definition 3. A search problem S corresponding to a polynomial bounded relation $S \subseteq \{0,1\}^* \times \{0,1\}^*$ is efficiently checkable if there is a polynomial time algorithm/TM M such that for every $x, y \in \{0,1\}^*$,

$$M(x,y) = \begin{cases} 1 & \text{if } (x,y) \in S, \\ 0 & \text{if } (x,y) \notin S. \end{cases}$$

The relation is polynomial bounded. So there is polynomial p such that $|y| \leq p(|x|)$.

Is every every search problem belonging to **PC** is in **PF**[4]? If that is the case, then any search problem S for which the validity of the given solution can be checked efficiently, can also be *found* efficiently.

[1] Find a solution polynomial time, not a standard name.

[2] There is no known efficient algorithm. But there is also no proof that it is impossible to have an efficient algorithm.

[3] Efficiently checkable solution, not a standard name.

[4] Note that unlike $\mathbf{P} \subseteq \mathbf{NP}$, $\mathbf{PF} \not\subseteq \mathbf{PC}$.

1.2 Decision Problems: P versus NP

There is a close connection between the **P** versus **NP** question and the **PC** versus **PF** question. But first of all we formally define the class **P** and redefine the class **NP**.

Definition 4. A *decision problem* $S \subseteq \{0,1\}^*$ is efficiently solvable if there is a polynomial time bounded algorithm (or Turing machine) A such that for all $x \in \{0,1\}^*$ the machine halts and

$$A(x) = \begin{cases} 1 & \text{if } x \in S, \\ 0 & \text{if } x \notin S. \end{cases}$$

There are many natural problems that belong to the class **P**. But there are also many natural problems that are **not known** to be in **P**. But they have an *efficiently verifiable proof system.*
A set $S \subseteq \{0,1\}^*$ has a *proof system*

(a) If there is a *proof* of the membership of $x \in S$ (*completeness*).

(b) Also the proof system does not fail. If $x \notin S$, there is no *false proof* of x (*soundness*).

Definition 5. A *decision problem* $S \subseteq \{0,1\}^*$ has an *efficiently verifiable proof* system if there is a polynomial p and a polynomial time bounded algorithm V so that

(a) *Completeness:* for every $x \in S$ there is an $y \in \{0,1\}^*$ of length $p(|x|)$, called *certificate/witness/proof*, such that $V(x,y) = 1$.

(b) *Soundness:* for every $x \notin S$ and for every y, $V(x,y) = 0$.

For the *soundness* part the lengths of y is also bounded by some polynomial as the verifier will stop in polynomial time.

Such a *decision problem* belongs to the class **NP**. Following are a few examples.
Example 1.

1. $\mathsf{COMPOSITE} = \{n \in \mathbb{N} : \ \exists p, q \in \mathbb{N} \ (p, q > 1 \text{ and } n = p \cdot q)\}$. The certificate is a pair of factors which cannot be longer than $c\lfloor \cdot \log_2 n \rfloor$, where the input is of length $\lceil \log_2 n \rceil$.

2. Whether a graph G has an *independent set* of certain size.
   $\mathsf{INDSET} = \{< G, k >: \ \exists S \subseteq V(G), \ |S| \geq k, \text{ and } \forall u, v \in S, \ \{u, v\} \notin E(G)\}$, $k \in \mathbb{N}$ specifies the size of the *independent set*. The certificate is a set of vertices.

3. $\mathsf{TSP} = \{< G = (V, E), d : E \to \mathbb{N}, k >:$ there is a traveling salesperson's tour of distance $\leq k\}$. $V = \{1, 2, \cdots, n\}$ is a set of nodes, $\binom{n}{2}$ distances d_{ij} between nodes i and j, and k is a number. Decide whether there is a tour that visits every node exactly once and the total length traversed is at most k. The certificate is a sequence of such nodes.

4. GRAPH-ISO $= \{< G_1, G_2 >:$ graph G_1 and graph G_2 are isomorphic $\}$. Given two adjacency matrices E_1 and E_2 corresponding to G_1 and G_2, decide whether there is a permutation $\pi : V_1 \to V_2$ so that E_1 after reordering is same as E_2.

The class $\mathbf{P} \subseteq \mathbf{NP}$. If $S \in \mathbf{P}$, then there is a polynomial time algorithm A to decide the membership of $x \in S$. But then for all y, the *verifier* V can ignore y and behave like A i.e. $V(x, y) = A(x)$ for all y.
It is unknown whether $\mathbf{NP} \subseteq \mathbf{P}$[5]. Note that if we can produce the witness y for the membership of $x \in S$ efficiently, then $\mathbf{NP} \subseteq \mathbf{P}$!

1.2.1 Two Definitions of NP and Their Equivalence

The classical definition of the class **NP** is the collection of sets $S \subseteq \{0,1\}^*$, for which the membership question can be answered by a polynomial time bounded nondeterministic Turing machine (NTM).
A set $S \subseteq \{0,1\}^*$ is in **NP** if there is a polynomial time bounded NTM N so that on input $x \in \{0,1\}^*$, at least one computation path of N *accepts* x if $x \in S$; but no computation path accepts x if $x \notin L$.

Two definitions of **NP** are equivalent.
Let there be a polynomial time NTM N for S. For each $x \in S$, there is a sequence of choice of transitions of N that leads to acceptance. The length of the sequence cannot be longer than *poly*$(|x|)$. Given the description of N a verifier V can be designed so that if $x \in S$ and w is the choice sequence of transition of an accepting run of N on x, then $V(x, w) = 1$. But for each $x \notin S$, for all $y \in \{0,1\}^*$, $V(x, y) = 0$ as there is no accepting run.

On the other hand, given the polynomial time verifier V for $S \in \mathbf{NP}$ we can design the following NTM N. N: input x

1. Nondeterministically creates a witness string w so that $< x, w >$ is accepted by V. If $x \in L$, then such a witness string must exist. The length of w is bounded by $p(|x|)$.

2. Run the verifier on the pair of input x and the NTM generated certificate w, if $V(< x, w >) = 1$, accept, else *reject*.

The running time of N for every choice of w is bounded by *poly*$(|x|)$.

If $L \subseteq \{0,1\}^*$ belongs to **NP**, then $\overline{L} = \{x \in \{0,1\}^* : x \notin L\}$ belongs to a class called **coNP**.
It is more difficult to certify the negation of some claim. A Boolean formula is *satisfiable* can be established by providing a list of satisfying assignment. But what can be a short (polynomial bounded) certificate to establish the *unsatisfiable* of a Boolean formula.
The class **P** is closed under complementation i.e. $L \in \mathbf{P}$ if and only if $\overline{L} \in \mathbf{P}$.
So $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$.

[5] But there is very strong belief that it is the case.

1.2.2 Two Interesting Problems

e define the following languages:

$$SAT^6 = \left\{ \phi = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{n_i} u_{ij} \right) : \ m, n_i > 0 \ \text{ and } \phi \text{ is satisfiable} \right\},$$

where u_{ij} is a *literal* and

$$3SAT = \left\{ \phi = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{3} u_{ij} \right) : \ m > 0 \text{and } \phi \text{ is satisfiable} \right\},$$

$$2SAT = \left\{ \phi = \bigwedge_{i=1}^{m} (l_{i1} \vee l_{i2}) : \ \phi \text{ is satisfiable} \right\},$$

It is well known that $3SAT$ is among the hardest problems in **NP** known as **NP**-*complete* problem. But $2SAT$ is in **P**.

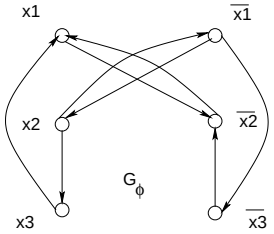
Let ϕ be a 2SAT formula. We construct a graph $G_\phi = (V_\phi, E_\phi)$, where $V_\phi = \{x, \overline{x} : \ x \text{ is a variable in } \phi\}$, and $E_\phi = \{(l_1, l_2) : \text{if } (l_2 \vee \overline{l_1}) \text{ (or } (\overline{l_1} \vee l_2)) \text{ is a clause in } \phi\}$.
Each edge in G_ϕ captures a clause in ϕ as a logical implication. Note that $(v \vee \overline{u})$, $(\overline{u} \vee v)$ and $(u \Rightarrow v)$ are logically equivalent.
There is a symmetry in the graph. A clause $(l_1 \vee l_2) = (\overline{\overline{l_1}}, \overline{\overline{l_2}})$ gives rise to two edges: $(\overline{l_1} \vee l_2)$ and $(\overline{l_2}, l_1)$. If there is a path from some literal $l_1 \to \cdots \to l_k$, $k \geq 1$ in the graph, then by the transitivity of implication we have $(l_1 \Rightarrow l_k)$. If there is a path from l_1 to l_k, then there is a path from $\overline{l_k}$ to $\overline{l_1}$.

Following the semantics of implication, if l_1 is assigned the value *true*, then every literal reachable from l_1 in G_ϕ should also be *true*. Symmetrically, if l_1 is assigned *false*, then all its predecessor literals will also be *false*.

A variable x cannot be assigned any truth value in a formula ϕ, if there is a path from x to $\overline{x}$ (equivalently a path from $\overline{x}$ to x) in G_ϕ, as it is same as $x \Leftrightarrow \overline{x}$ - a contradiction.

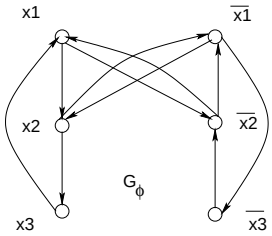
Example 2. Consider the following example,

$$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_2} \vee x_3).$$



There is an satisfying assignment, $x_1 = 1, x_2 = 0, x_3 = 1$. But if we include another clause, $(\overline{x_1} \vee x_2)$, then there is no satisfying assignment any more, as there will be a path from x_1 to $\overline{x_1}$ and also a path from $\overline{x_1}$ to x_1.

[6] One may define $SAT = \{\phi : \ \phi \text{ is satisfiable.}\}$.



Lemma 1. A 2SAT formula ϕ is *unsatisfiable* if and only if there is a variable x such that there is a path from x to $\overline{x}$ (also a path from $\overline{x}$ to x).
Proof: Let for some variable x there are two such paths and at the same time ϕ is satisfiable. So there is a truth value $v(x)$ for x. Let $v(x)$ is *true* and $v(\overline{x})$ is *false*. As there is a path from x to $\overline{x}$, there must be an edge (l_1, l_2) such that $v(l_1) = true$ but $v(l_2) = false$. The corresponding clause is $(\overline{l_1} \vee l_2)$ and is not satisfiable - a contradiction. Similar argument works for $v(\overline{x}) = false$.

In the other direction, we assume that there is no variable x with such pair of paths. The satisfying truth assignment of ϕ is as follows:
The following procedure will be repeated until all nodes are assigned truth values.
Take a literal l, a node in G_ϕ, that has not been assigned any truth value and there is no path from l to $\overline{l}$. Assign *true* to l and every literal reachable from the node of l. Assign *false* to the negation of these literals. In other words, if a node is assigned *false* then its predecessor is also assigned *false*. If l' is reachable from l, then $v(l') = true$. If the node $\overline{l}$ is reachable from $\overline{l'}$, then both have value *false*.

We claim that the process cannot assign same truth value to l' and $\overline{l'}$ i.e. nodes of both l' and $\overline{l'}$ cannot be reachable from l. If that was possible then $\overline{l}$ would have been reachable from both of them resulting a path from l to $\overline{l}$. QED.

Proposition 1. 2SAT $\in$ **P**
Proof: The steps of the algorithm are as follows:
M: input ϕ

1. Build the graph G_ϕ.
2. For each variable x, test whether $\overline{x}$ is reachable and vice versa.
3. *Accept* if no such path exist; otherwise *reject*.

It is an $O(n^2)$ algorithm. QED.

Another pair of similar problems are related to graph colouring. The languages are

$3COL = \{< G >:\ G$ is a graph whose vertices can be coloured with 3 colours$\}$.

It is known that $3COL$ is **NP**-*complete*. But

$2COL = \{< G >:\ G$ is a graph whose vertices can be coloured with 2 colours$\}$.

is in **P**. A grapg that can be coloured with 2 colours is a bipartite graph. That can be tested using BFS in linear time.

1.3 PF versus PC and P versus NP

There is an equivalence between the **PF** versus **PC** question and the **P** versus **NP** question. That equivalence is demonstrated by the following theorem.

Theorem 2. $\mathbf{PC} \subseteq \mathbf{PF}$ if and only if $\mathbf{NP} \subseteq \mathbf{P}$ ($\mathbf{NP} = \mathbf{P}$).

Every search problem $R \in \mathbf{PC}$ (checkable in polynomial time) is solvable in polynomial-time i.e. $R \in \mathbf{PF}$ if and only if every decision problem $S \in \mathbf{NP}$ has a polynomial-time decision procedure i.e. $S \in \mathbf{P}$.

Outline of th Proof:

To prove that $\mathbf{NP} \subseteq \mathbf{P}$ we start with the assumption $\mathbf{PC} \subseteq \mathbf{PF}$. Consider a set $S \in \mathbf{NP}$. Build a relation $R_S \in \mathbf{PC}$. The relation by assumption is also in **PF**. So there is an efficient algorithm to find a solution $y \in R_S(x)$. Finally $x \in S$ if and only if $\exists y$ so that $(x, y) \in R_S$.

To prove that $\mathbf{PC} \subseteq \mathbf{PF}$ we start with the assumption $\mathbf{NP} \subseteq \mathbf{P}$. Consider a relation $R \in \mathbf{PC}$. Build a set $S'_R \in \mathbf{NP}$. The set S'_R by assumption is also in **P**. The polynomial time decision procedure is used to find a solution of x in polynomial time. So $R \in \mathbf{PF}$.

Proof:

Suppose $\mathbf{PC} \subseteq \mathbf{PF}$:

Let $S \in \mathbf{NP}$ and V be the verifier for S.

Define the relation $R_S = \{(x, y) \in \{0,1\}^* \times \{0,1\}^*,\ V(x, y) = 1\}$ i.e. $(x, y) \in R_S$ if and only if $x \in S$. It is a polynomial bounded relation and can be checked in polynomial time by the verifier V, so $R_S \in \mathbf{PC} \Rightarrow R_S \in \mathbf{PF}$ (by the assumption $PC \subseteq PF$).

So the search problem of R_S is efficiently solvable and there is a polynomial time algorithm M_S to search for a solution of $x \in S$.

$M_S(x)$ returns a $y \in R_S(x)$ if $R_S(x) \neq \emptyset$; otherwise it returns $\perp$. The polynomial time decider for S using M_S as an oracle is as follows:

D_x: input x

(a) Call M_S with x as the parameter.

(b) If $M_S(x) \neq \perp$, *accept*.

(c) Else *reject*.

This makes $\mathbf{NP} \subseteq \mathbf{P}$. It is equivalent to say $\mathbf{P} = \mathbf{NP}$ (as $\mathbf{P} \subseteq \mathbf{NP}$).

Suppose $\mathbf{P} = \mathbf{NP}$:

Let $R \in \mathbf{PC}$. We define the set (not a relation) $S'_R = \{< x, y' >: \exists y''\ (x, y'y'') \in R\}$ where y' in $< x, y' >$ is a prefix of some solution of the given x ($< x, y' >$ is the encoding of (x, y)).

R is a polynomial time checkable relation and there is a efficient algorithm M_R to check the membership of $(x, y) \in R$. So the set $S'_R \in \mathbf{NP}$ as given $(< x, y' >, y'')$ we can efficiently check whether $(x, y'y'') \in R$.

By our assumption $\mathbf{NP} \subseteq \mathbf{P} \Rightarrow S'_R \in \mathbf{P}$. So there is an efficient decision procedure A'_S for S'_R.

We use A'_S (as an oracle) to design an efficient search algorithm in R.

Start with $A'_S(< x, \epsilon >)$. If $A'_S(< x, \epsilon >) = 0$ i.e. $< x, \varepsilon > \notin S'_R$, then there is no solution y for x i.e. $R(x) = \emptyset$. The output of the search algorithm is '$\perp$'.

Otherwise we extend the current y' to $y' \leftarrow y'0$ if $A'_S(< x, y'0 >) = 1$ i.e. $< x, y'0 > \in S'_R$. Otherwise, extend the current y' to $y' \leftarrow y'1$ if $A'_S(< x, y'1 >) = 1$.

After some polynomial amount of time none of the conditions will hold and current $y' = y$ satisfying $(x, y) \in R$. The solution of x is y.
As the time is bounded by *poly*(x), $R \in \mathbf{PF}$.
E_R: input x

(a) $y \leftarrow \varepsilon$

(b) Call A'_S with input $< x, y >$.

(c) If A'_S rejects, return '$\perp$'.

(d) Else repeat the following steps.

(e) Call A'_S with input $< x, y0 >$.

(f) If A'_S accepts, $y \leftarrow y0$ and continue.

(g) Else call A'_S with input $< x, y1 >$.

(h) If A'_S accepts, $y \leftarrow y1$ and continue.

(i) Else return y.

QED.

1.4 Polynomial-Time Reduction

The basic idea of *reduction* is to use an *external subprogram* to solve a computational problem. The input-output functionality of the external subprogram is known, but its actual computation is like a *black box*. Such computation is modeled as an *Oracle Turing machine (OTM)*. The main algorithm sends queries to the *Oracle* during its computation. If an OTM solves a problem A using the oracle to solve the task B, then we say that A is reduced to B.

The running time of an OTM is the number of steps of its own computation. Each query and reply from oracle is treated as a single step.

Definition 6. (*Cook-reduction*) A problem A is *Cook-reducible* to a problem B if there is a polynomial time OTM M such that for every function f that solves B, the oracle machine M^f solves A. Where $M^f(x)$ is the output of M on input x with the oracle access to f.

Note that when B is a decision problem of the set S, then $f : \{0,1\}^* \to \{0,1\}$ so that $f(x) = 1$ if $x \in S$ and $f(x) = 0$ if $x \notin S$.
But the oracle function is not uniquely determined if it deals with a relation R. $R(x) = \{y \in \{0,1\}^* : (x, y) \in R\}$ and the oracle function is $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$. So $f(x) \in R(x)$ if $x \in S_R = \{w : \exists y(w, y) \in R\}$, but otherwise $f(x) = \perp$.

The Karp-reduction of a decision problem to another decision problem is defined as follows.

Definition 7. (*Karp-reduction*) A polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ is a *Karp-reduction* from $A \subseteq \{0,1\}^*$ to $B \subseteq \{0,1\}^*$ if for all $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$.

In terms of oracle TM we may view Karp-reduction as follows where M_B is a decider for the set B.
M_A: input x

(a) Compute $f(x) = x'$.

(b) Ask $x' \stackrel{?}{\in} B$ to M_B.

(c) return the answer.

A Karp-reduction is augmented to reduce a search problem to another search problem.
Let R and R' be two search problems. The reduction can be represented by a pair of polynomial-time computable functions f and g.
To search for a solution in R for the input x using the oracle of R', computes $f(x) = x'$. Makes a query for x' to the oracle of R'. It obtains y' such that $(x', y') \in R'$. Uses y' to compute the solution $y = g(x, y')$ so that $(x, y) \in R$.

Definition 8. (*Levin-reduction*) A pair of polynomial-time computable function f and g is called a Levin-reduction of R to R' if f is a Karp-reduction from $S_R = \{x : \exists y \ s.t. \ (x, y) \in R\}$ to $S'_R = \{x' : \exists y \ s.t. \ (x', y') \in R'\}$. Also for every $x \in S_R$ and $y' \in R'(f(x))$ it holds that $(x, g(x, y')) \in R$ where $R'(x') = \{y' : (x', y') \in R'\}$.

1.4.1 Equivalence of Search and Decision Problems

Consider a search problem viewed as the relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ and the corresponding decision problem $S_R = \{x : \exists y \ s.t. \ (x, y) \in R\}$.

An efficient reduction of the decision problem S_R to the search problem R is simple.

1. Send the query x (searching for a solution of x) to the oracle of R.
2. If the oracle returns $y \in R(x)$, *accept* i.e. $(x, y) \in R$ implies $x \in S_R$. Otherwise (oracle returns '$\perp$') i.e. $R(x) = \emptyset$, *reject*.

If $R \in \mathbf{PC}$ then the reduction shows that $S_R \in \mathbf{NP}$. The reduction of S_R to R is based on whether the oracle of R returns '$\perp$' or some $y \in R(x)$.
But the reduction of R to S_R requirs the value of $y \in R(x)$ if $R(x)$ is non-empty. So the number of queries in this reduction are more. Following is an example of the reduction of R_{SAT} to SAT.
Example 3. Consider $SAT = \{\psi : \ \psi$ is a satisfiable CNF Boolean formula$\}$ and $R_{SAT} = \{(\psi, \sigma) : \ \sigma$ is a satisfying assignment of $\psi\}$.

We design the following oracle TM M^{SAT}. Let the SAT formula ψ has k variables $(x_1, x_2, \cdots, x_k)$.

M^{SAT}: input $\psi(x_1, x_2, \cdots, x_k)$

1. Call oracle SAT on ψ.
2. If the oracle returns 0 (ψ is not satisfiable), return $\perp$.
   Otherwise, ψ is satisfiable and we build the assignment string.
3. $\sigma \leftarrow \varepsilon$ (empty truth-value string).
4. Do the following for $i = 1, \cdots, k$.
   (a) If $\psi = 1(true)$, break (σ contains the assignments and also don't cares).

(b) Evaluate $\psi' \leftarrow \psi[x_i \leftarrow 0]$

(c) Call oracle SAT on ψ'.

(d) If the oracle returns 1, $\sigma \leftarrow \sigma 0$.

(e) Otherwise, evaluate $\psi' \leftarrow \psi[x_i \leftarrow 1]$ and $\sigma \leftarrow \sigma 1$.

(f) $\psi \leftarrow \psi'$.

At the end $\sigma = \sigma_1 \cdots \sigma_l$, $l \leq k$ contains the assignments of $x_1 \cdots x_l$. If $l < k$, then $x_{l+1} \cdots x_k$ can take any Boolean value. During the reduction of the search problem R to the decision problem S_R the solution is built iteratively starting from the prefix of a valid solution.

In general every for every relation in $R \in \mathbf{PC}$, the corresponding decision problem $S'_R = \{(x, y') : \exists y'' \; s.t. (x, y'y'') \in R\} \in \mathbf{NP}$.
But for specific problem e.g. $R = R_{SAT}$ where the decision problem SAT is **NP**-*complete*. In such case deciding the membership of $S'_R \in \mathbf{NP}$ is reducible to SAT. But that is not true for every $(R \in \mathbf{PC}, S_R)$ pair.

1.4.2 Optimization Problems As Search Problems

A search problem may have different solutions with different costs associated with the solutions. It may be of interest to search for a solution of maximum value or minimum cost. The other possibility is to find a solution of cost above or below a threshold. Examples of such problems are finding the largest clique of a graph or finding the clique above a given size. Finding the minimal spanning tree of a weighted graph etc.

We consider maximization problems. The value attached to the solution should be above a threshold or it should be maximum. These are two different types of problems. We have a binary relation $R \subseteq \{0.1\}^* \times \{0,1\}^*$ of problem-solution pairs and a value function $f : \{0.1\}^* \times \{0,1\} \rightarrow \mathbb{N}_0$ attached to every pair.

1. *Solution above the threshold v*: Given a pair (x, v) it is necessary to find $y \in R(x)$ so that $f(x, y) \geq v$. This may be viewed as a search problem of the relation

$$R_f = \{(< x, v >, y) : (x, y) \in R \wedge f(x, y) \geq v\}.$$

2. *Maximization problem:* Given x it is necessary to find y so that $y \in R(x)$ and $f(x, y) = v_x \geq f(x, y')$ for all $y' \in R(x)$. This may be viewed as the following search problem,

$$R'_f = \{(x, y) \in R : f(x, y) = \max\{f(x, y') : (x, y') \in R\}\}.$$

Theorem 3. Let $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \mathbb{N}_0$ be a polynomial time computable function and $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a polynomial bounded binary relation. The search problem of R_f can be reduced to the search problem of R'_f and vice versa. These two problems are computationally equivalent.

Proof. The algorithm for R_f uses the solver M'_f (oracle) of R'_f as follows:
M_f: input $< x, v >$

(a) Send a query x to M'_f.

(b) If the oracle returns '$\perp$', $R(x) = \emptyset$, return $\perp$ i.e. $R_f(< x, v >) = \emptyset$.

(c) Else the oracle returns $y_0 \in R(x)$ so that $f(x, y_0)$ is the largest.

(d) Compute $f(x, y_0) = u$.

(e) If $u \geq v$, return y_0 i.e. $R_f(< x, v >) = y_0$.

(f) Else return $\perp$ i.e. $R_f(< x, v >) = \emptyset$.

The reduction in the other direction i.e. R'_f to R_f is as follows.

(a) The function f is computable in polynomial time. So there is $l = poly(|x|)$ such that $f(x, y) \leq 2^l - 1$. The binary representation of the *values* can have at most l-bits $b_{l-1}b_{l-2}\cdots l_2 l_1 l_0$.

(b) The value of $v_x = \max\{f(x, y) : (x, y) \in R\}$ is found out by a binary search on l bits. l queries of the form $< x, v_i >$, where $l > i \geq 0$, are sent to the oracle of R_f. The bits $b_{l-1}\cdots b_{i+1}$ are already set and the value of $v_x \geq b_{l-1}\cdots b_{i+1}\overbrace{\mathbf{0}0\cdots 0}^{i+1}$. The value $v_i = b_{l-1}\cdots b_{i+1}\overbrace{\mathbf{1}0\cdots 0}^{i+1}$ is sent to the oracle. If the oracle returns a y, b_i remains 1. If it returns '$\perp$', $b_i \leftarrow 0$ and the next iteration starts.

(c) The first iteration starts with $i \leftarrow l - 1$

(d) $v_x < v$ if and only if $R_f(< x, v >) = \emptyset$ and the oracle returns '$\perp$'.

(e) If $R(x) = \emptyset$, then all answers to all queries will be '$\perp$' and $v_x = 0$.

(f) If the value of $v_x > 0$, a query $< x, v_x >$ to R_f will return y such that $f(x, y) = v_x$.

Let $R \in \mathbf{PC}$ and f is polynomial time computable.
Given the input $< x, v >$ and a solution y, the membership of $y \in R_f(< x, v >)$ can be checked in polynomial time. So $R_f \in \mathbf{PC}$. But R'_f may not be in $\mathbf{PC}$. As an example searching for a clique of size k in a graph G is in $\mathbf{PC}$ as given k vertices it can be checked in polynomial time.

But status of searching for the largest clique in a graph is not known to be in $\mathbf{PC}$ (people believe it to be unlikely). Though it can be *Cook-reducible* to $\mathbf{PC}$.

1.5 NP-Completeness

The question of $\mathbf{P}$ versus $\mathbf{NP}$ is not yet resolved. So at best one can say that a hard problem $A \in \mathbf{NP}$ is not in $\mathbf{P}$ under the assumption that $\mathbf{P}$ is a proper subset of $\mathbf{NP}$. We prove the hardness of A by showing that all problems in $\mathbf{NP}$ are *polynomial-time reducible* to A[7]. So a polynomial-time solution of A proves that $\mathbf{P} = \mathbf{NP}$.

[7]Under the assumption of $\mathbf{P} \neq \mathbf{NP}$ it can be proved that there are problems in $\mathbf{NP}$ which does not belong to $\mathbf{P}$ and also not $\mathbf{NP}$-*hard*.

Definition 9. A set S is **NP**-*hard* if every set $B \in \mathbf{NP}$ is *Karp-reducible* to S.
A set S is **NP**-*complete* if (i) $S \in \mathbf{NP}$ and (ii) S is **NP**-*hard*.

A binary relation $R \subset \{0,1\}^* \times \{0,1\}^*$ is **PC**-*complete* if (i) $R \in \mathbf{PC}$ and (ii) every binary relation $S \in \mathbf{PC}$ is *Levin-reducible* to R.

We often abuse the definition and say that a search proble R is **NP**-*complete* (or **NP**-*hard*) insted of **PC**-*complete* (or **PC**-*hard*).

The existence of an **NP**-*complete* problem tells us that a single problem can *encodes* a very wide range of apparently unrelated problems!

Theorem 4. There exists **NP**-*complete* set and **PC**-*complete* relation.
Proof: We prove that there is a set in **NP** (a relation in **PC**) that can encode every problem in **NP** (in **PC**). We call the set S_u and the corresponding relation as R_u. They are defined as follows.
The set $S_u = \{< M, x, 1^t >: M \text{ is a DTM and } \exists y \text{ s.t. } M \text{ accepts } < x, y > \text{within } t \text{ steps, where } |y| \leq t \}$. The corresponding relatio $R_u = \{(< M, x, 1^t >, y) : M \text{ is a DTM and } M \text{ accepts } (x, y) \text{ within } t \text{ steps, where } |y| \leq t \}$..

Claim: $R_u \in \mathbf{PC}$ and $S_u \in \mathbf{NP}$. An universal Turing machine used as a verifier. It takes the input $(< M, x, 1^t >, y)$, runs M on $< x, y >$ for t steps. If $< x, y >$ is accepted, then $< M, x, 1^t > \in S_u$ or $(< M, x, 1^t >, y) \in R_u$. The number of execution steps of M on $< x, y >$ is linear in $(< M, x, 1^t >, y)$ (due to the presence of 1^t). The simulation is $poly(| < M, x, 1^t > | + |y|)$. So $S_u \in \mathbf{NP}$ and $R_u \in \mathbf{PC}$.

Now we prove that S_u (resp. R_u) is **NP**-hard (resp. **PC**-hard).
Let $S \in \mathbf{NP}$ and its *witness relation* $R \in \mathbf{PC}$. An $x \in S$ if and only $\exists y$ s.t $(x, y) \in R$. Let R be bounded by the polynomial $p_R()$. The DTM M_R decides the membership of $(x, y) \in R$ in polynomial time $t_R()$. Then the *Karp-reduction* is

$$f : x \mapsto < M_R, x, 1^{(t_{R(|x|)} + p_{R(|x|)})} > .$$

$$x \in S \text{ if and only if } < M_R, x, 1^{(t_{R(|x|)} + p_{R(|x|)})} > \in S_u.$$

This mapping can be computed in polynomial time as encoding of M_R is of fixed length, encoding of x is $O(|x|)$, encoding of the strings of 1's is of length $poly(|x|)$.
$x \in S$ if and only if there is y of length $\leq p_R(|x|)$ and $(x, y) \in R$ if and only if M_R accepts (x, y) in $t_r(|x|+|y|)$ steps, if and only if $< M_R, x, 1^{(t_{R(|x|)} + p_{R(|x|)})} > \in S_u$.

Consider an $R \in \mathbf{PC}$. A search for $x \in R$ is equivalent to a search for $< M_R, x, 1^{(t_{R(|x|)} + p_{R(|x|)})} >$. That is

$$(x, y) \in R \text{ if and only if } (< M_R, x, 1^{(t_{R(|x|)} + p_{R(|x|)})}, y) \in R_u.$$

The *Levin-reduction* is a pair (f, g) so that $f(x) =< M_R, x, 1^{(t_{R(|x|)} + p_{R(|x|)})} >$ and $g(x, y) = y$ i.e.

$$(x, g(x, y)) = (x, y) \in R \text{ if and poly if } (f(x), y) \in R_u.$$

QED.

1.5.1 Circuit Satisfiability: CSAT

CSAT is a Boolean circuits, a DAG with internal nodes labeled by Boolean operators { and ($\wedge$), or ($\vee$), not ($\neg$)} with in-degrees $(2, 2, 1)$. There are several

input nodes and one output node. If $x \in \{0,1\}^n$ is the input to a circuit C, then the output is $C(x)$. $CSAT = \{C : C \text{ is a Boolean circuit and } C(x) = 1\}$. The corresponding search relation is $R_{CSAT} = \{(C, x) : C \text{ is a Boolean circuit and } x \text{ is an satisfying assignment i.e. } C(x) = 1\}$.

Theorem 5. Let $f : \mathbb{N}_0 \to \mathbb{N}_0$ be a function such that $f(n) \geq n$. The set $L \subseteq \{0,1\}^*$ is in $\mathbf{TIME}(f(n))$, then $L \in \mathbf{SIZE}(O(f^2(n)))$.

The outline of the proof goes as follows: Let M be the Turing machine that decides L in time $f(n)$. For each input of length n of M, a Boolean circuit C_n can be constructed from the computation of M. The computation can be captured by the sequence of configurations of length $f(n)$ and each of size $f(n)$. This forms an $f(n) \times f(n)$ matrix. Where the first row corresponds to the *start configuration* and the last row corresponds to the final configuration[8]. Output of the gates corresponding to the i^{th} configuration (i^{th} row of the matrix), are input to the gates of the $(i+1)^{th}$ row simulating the $(i+1)^{th}$ configuration.

Each cell of the matrix contains a tape symbol (element of Γ), or a composite symbol of state (Q) and tape (Γ) if the head is scanning that cell.
A configuration of the form $\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{q_5 1}\boxed{0}\boxed{0}\boxed{1}$ means that the machine is in state q_5, the head is scanning 5^{th} cell from the left containing '1' in it. The value of *cell*(i,j) is the content of the j^{th} cell of the i^{th} configuration.

We introduce Boolean variables *sym*(i,j,k), where k, a tape symbol or a composite symbol, and i,j corresponds to the *cell*(i,j). The variable *sym*(i,j,k) takes the value *true (1)* if the symbol k is present in *cell*(i,j), otherwise it is 0. We know that a cell may contain an element of $\Gamma \cup (Q \times \Gamma)$. Let $|\Gamma \cup (Q \times \Gamma)| = c$. So there are $cf(n)^2$ variables of type *sym*(i,j,k). A Boolean circuit will ensure that only one of c variables corresponding to the cell *cell*(i,j), $1 \leq i,j \leq f(n)$ will have the Boolean value *true (1)*.

The value of *sym*(i,j,k) depends on the the contents of *cell*$(i-1,j-1)$, *cell*$(i-1,j)$, *cell*$(i-1,j+1)$, and the transition function δ of the Turing machine M. Let the possible values of these three cells for which *cell*(i,j) has the symbol k. That is *sym*(i,j,k) is *true (1)* if symbols of the above mentioned cells are (a_1,b_1,c_1) or (a_2,b_2,c_2), $\cdots$ (a_m,b_m,c_m). Then the Boolean function for *sym*(i,j,k) is

$$sym(i,j,k) = \bigvee_{l=1}^{m} (sym(i-1,j-1,a_l) \wedge sym(i-1,j,b_l) \wedge sym(i-1,j+1,c_l)$$

This is repeated for all the variables of all the configurations except the start configuration.

Let the input be $x = x_1 x_2 \cdots x_n$: x_1 is directly connected to *sym*$(1,1,\boxed{q_0 1})$ and connected through a not gate to *sym*$(1,1,\boxed{q_0 0})$ i.e. *sym*$(1,1,\boxed{q_0 1})$ is *true(1)* if $x_1 = 1$ and *sym*$(1,1,\boxed{q_0 0})$ is *true(1)* if $x_1 = 0$. In both the cases the machine is at its start state q_0. Similarly, *sym*$(1,2,1)$, $\cdots$, *sym*$(1,n,1)$ are directly connected to $x_2, \cdots, x_n$ respectively. $x_2, \cdots, x_n$ are connected through not-gates to *sym*$(1,2,0)$, $\cdots$, *sym*$(1,n,0)$. Boolean '1' is connected to *sym*$(1,n+1,\sqcup) \cdots$ *sym*$(1,f(n),\sqcup)$ ($\sqcup$ stands for *empty cell* or *blank cell*). All other variables of first row are connected to Boolean '0'.

[8]The Turing machine M is such that in its accepting configuration its head moves to the leftmost cell after erasing the content of the tape. Moreover, if M halts before $f(n)$ steps, the same configurations are repeated.

The output gate corresponds to $sym(f(n), 1, \boxed{q_A \sqcup})$. Every cell has fixed number of symbols $sym(i, j, k)$, and every symbol is connected to the symbols of three neighboring cells of the previous row (except the first row) through fixed number of gates. So the size of this circuit is $O(f(n)^2)$.
Also note that the construction of C_n takes $O(f(n)^2)$ time.
Theorem 6. CSAT is **NP**-*complete*.
Proof: It is clear that CSAT is in **NP**. The certificate is an input that produces the output 1. This can be verified in linear time as the output of each gate can be computed in constant time.

To show that it is **NP**-*complete* we consider a set $L \subseteq \{0,1\}^*$ where $L \in \mathbf{NP}$. We design a polynomial time reduction function $f : \{01,\}^* \to \{0,1\}^*$ so that for all $x \in \Sigma^*$

$$x \in L \text{ if and only if } f(x) =< C > \text{ is satisfiable,}$$

As $L \in \mathbf{NP}$ there is a polynomial time verifier V and a polynomial $p : \mathbb{N}_0 \to \mathbb{N}_0$ so that for each $x \in \{0,1\}^*$, $x \in L$ if and only if $\exists w \in \{0,1\}^{p(|x|)}$, s.t. V accepts $< x, w >$ in polynomial time.

The reduction function f constructs a circuit C corresponding to the verifier V with input size $|x| + p(|x|)$ where the input x is plugged in. The remaining inputs correspond to the certificate.

If $x \in L$, then there is a certificate, and C is satisfiable.

The verifier V runs for n^k time, so the size of the circuit is $O(n^{2k})$. Connection to every gate is a constant amount of work. So the reduction is polynomial time bounded. QED.

1.5.2 A Few NP-*complete* Problems

We have already talked about the *Boolean circuits* called *Boolean formula*. These are tree structures with n-input and 1-output. A further restricted form of Boolean formulae are in *conjunctive normal form (CNF)*. The set $SAT = \{\psi :$ where ψ is a satisfiable Boolean formula in CNF $\}$. A CNF $\psi(x_1, \cdots, x_n) \in SAT$ if there is a satisfying truth assignment σ i.e. $\psi(\sigma) = 1$.
Corresponding relation is $R_{SAT} = \{(\psi, \sigma) : \psi(\sigma) = 1\}$.
Theorem 7. SAT is **NP**-*complete*. And R_{SAT} is **PC**-*complete*
Proof: $SAT \subseteq CSAT \in \mathbf{NP}$. So $SAT \in \mathbf{NP}$.
We prove that SAT is **NP**-*hard* by reducing $CSAT$ to SAT. A $CSAT$ circuit may have any depth. But SAT is a *depth-2* circuit. A $CSAT$ circuit can be transformed to an equivalent SAT circuit by introducing new variables for every internal gate (and, or, not: we assume 2,2,1 fan-in). These new variables should take values corresponding to the output of the Boolean operation of each internal gate. If there are n inputs to the $CSAT$ circuit and m internal gates, then there will be $m + n$ input to the equivalent SAT circuit.
Suppose a gate g has two input a and b, where a and b may be a primary input or output of other gates. We introduce a new variable g corresponding to the gate and generate Boolean expression as follows:

(a) If g is a *not*-gate with a as the only input, then equivalent Boolean formula is $(g \equiv \neg a)$.

(b) If g is an *or*-gate, then $(g \equiv (a \vee b))$.

(c) If g is an *and*-gate, then $(g \equiv (a \wedge b))$.

Note that each such clause is satisfiable if and only if the assignment matches with the functionality of the gate.
As an example $(1,1,1)$ satisfies $(g \equiv (a \wedge b))$, but $(1,0,1)$ does not. The valuation of g captures the outcome of the corresponding gate. If there is a satisfying assignment of the primary n variables of the $CSAT$ circuit C_n, then there is also a satisfying assignments of the derived Boolean formula with $n+m$ variables.

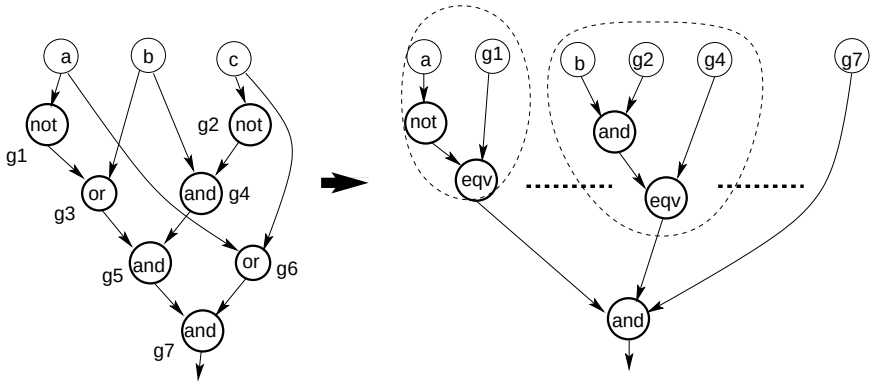
Each derived formula can be translated to three CNF clauses.

(a) $(g \equiv \neg a)$ is equivalent to $(g \Rightarrow \neg a) \wedge (\neg a \Rightarrow g)$ is equivalent to $(\neg g \vee \neg a) \wedge (a \Rightarrow g)$.

(b) $(g \equiv (a \vee b))$ is equivalent to $(g \Rightarrow (a \vee b)) \wedge ((a \vee b) \Rightarrow g)$ is equivalent to $(\neg g \vee (a \vee b)) \wedge (\neg (a \vee b) \vee g)$ is equivalent to $(\neg g \vee a \vee b) \wedge (\neg a \vee g) \wedge (\neg b \vee g)$.

(c) $(g \equiv (a \wedge b))$ is equivalent to $(g \Rightarrow (a \wedge b)) \wedge ((a \wedge b) \Rightarrow g)$ is equivalent to $(\neg g \vee (a \wedge b)) \wedge (\neg (a \wedge b) \vee g)$ is equivalent to $(\neg g \vee a) \wedge (\neg g \vee b) \wedge (\neg a \vee \neg b \vee g)$.

The final output is the conjunction of all eqivalent formulae and the output gate variable. This is a CNF formula ψ constructed from the circuit C_n in polynomial time i.e. $f(c_n) = \phi(x_1, \cdots, x_n, g_1, \cdots, g_m)$. QED.
Example 4.



Theorem 8. $3SAT$ is **NP**-*complet*, where $3SAT$ is the collection of CNF formulae with at most 3 literals per clause.
Proof: The previous reduction actually produces a $3SAT$ formula. So $3SAT$ is **NP**-*complete*. QED.

Definition 10. (*Set Cover*) $S_1, \cdots, S_m$ is a collection of finite sets and $k \leq m$ is a positive integer. Is there a collection of at most k sets $S_{i_1}, \cdots, S_{i_k}$ so that $\bigcup_{j=1}^{k} S_{i_j} = \bigcup_{i=1}^{m} S_i$.

Proposition 2. *Set Cover* is **NP**-*complete*.
Proof: *Set Cover* is in **NP** is not difficult to verify.
We reduce SAT to *Set Cover*. Let a CNF formula ψ has m clauses, $C_1, \cdots, C_m$ and n variables $x_1, \cdots, x_n$. We consider $2n$ sets $S_{1t}, S_{1f}, \cdots, S_{nt}, S_{nf} \subseteq \{1, \cdots, m\}$. $j \in S_{it}$ if the clause C_j has x_i ($x_i \leftarrow$ *true* makes C_j *true*). Similarly $j \in S_{if}$ if C_j has $\overline{x_i}$.

We define $2n$ sets as follows: $S_{2i-1} = S_{it} \cup \{i+m\}$ and $S_{2i} = S_{if} \cup \{i+m\}$, for $i = 1, \cdots, n$.

Note: $S_1 = S_{1t} \cup \{m+1\}$, $S_2 = S_{1f} \cup \{m+1\}$, $\cdots$, $S_{2n-1} = S_{nt} \cup \{m+n\}$, $S_{2n} = S_{nf} \cup \{m+n\}$.
The Karp mapping is $f(\psi) = (\{S_1, \cdots, S_{2n}\}, n)$. It can be computed in polynomial time.

Exactly one of S_{2i-1} or S_{2i} for each $i = 1, \cdots, n$ can be included to cover $\{m+1, \cdots, m+n\}$ and not to have more than n sets.
Let ψ is *satisfied* by $x_1 \leftarrow \sigma_1, \cdots, x_n \leftarrow \sigma_n$, $\sigma_i \in \{0,1\}$. Each clause C_j, $j = 1, \cdots, m$, evaluates to 1. Let the literal $l_i \leftarrow 1$ in the clause C_j. Depending on whether $l_i = x_i$ or $\overline{x_i}$, the clause number j is in S_{2i-1} or S_{2i}. We choose the appropriate one to cover j and $m+i$ in $\{1, \cdots, j, \cdots m, \cdots, m+i, \cdots m+n\}$.

On the other hand, let $S_{2i-\sigma_i}$, $\sigma_i \in \{0,1\}$, $i = 1, \cdots, n$ covers $\{1, \cdots, m, m+1, \cdots .m+n\}$. Each $j \in \{1, \cdots, m\}$ must be in some S_{2i-1} or S_{2i} for some $i \in \{1, \cdots, n\}$. Assign $x_i \leftarrow 1$ if $j \in S_{2i-1}$ or $x_i \leftarrow 0$ if $j \in S_{2i}$. This gives a satisfying assignment of all the variables of ψ. QED.

Definition 11. $3COL = \{< G >$: undirected graph G has a vertex colouring with at most three colours$\}$.

Proposition 3. $3COL$ is **NP**-*complete*.

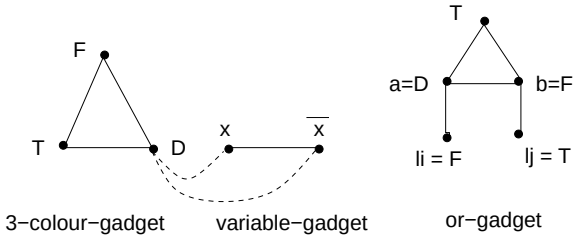
Proof: The certificate of 3COL is colouring of different vertices. The validity of colouring can be tested in polynomial time. So 3COL is in **NP**.

We reduce 3SAT to 3COL to show that 3COL is **NP**-*hard*. Let ϕ be a 3CNF formula with m clauses $C_1, \cdots, C_m$ and n variables $x_1, \cdots, x_n$. The construction of the graph G is as follows. The Boolean formula ϕ is *satisfiable* if and only if the grapg G is 3-colourable.

1. There is a pair of vertices x_i, $\overline{x_i}$ for every variable x_i and its negation $\overline{x_i}$.

2. Five vertices $u_1, \cdots, u_5$ for each clause C (different five vertices for each clause).

3. Three special vertices T, F, D for three colours *true*, *false* and D.



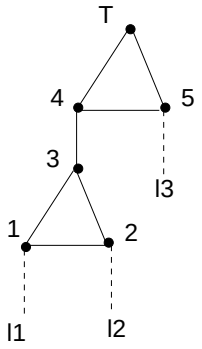
1. A triangle is formed with the vertices T, F, D to force three colours to colour them.

2. Triangles are formed with every pair of vertices x_i, $\overline{x_i}$ and D so that a vertex corresponding to a literal can take either colour T (*true*) or F (*false*) (not D).

3. It is necessary to ensure that at least one literal in every clause is *true* if and only if the graph is 3-colourable. We start with a graph of 3-vertices, a, b,and T forming a triangle. The vertex a is connected to a literal-vertex l_i and b to a literal-vertex l_j.
   In the triangle of a, b, T, the vertices a and b can be coloured only with F

and D.
Literal vertces can be coloured only with T and F as they are already connected to D. So one literal vertex connected to a or b must be coloured with T. This is called an *or-gadget*.

4. There will be five vertices $u_1, \cdots, u_5$ corresponding to each clause C (different 5 vertices for each clause). They are connected by the following edges: $\{u_1u_2, u_1u_3, u_2u_3\}\}$ (forms a triangle), $\{u_4u_5, u_4T, u_5T\}$ (forms another triangle), $\{u_3u_4\}$ (connects the triangles). The vertices corresponding to the literals of $C = l_1 \vee l_2 \vee l_3$ are connected as $\{u_1l_1, u_2l_2, u_5l_3\}$.

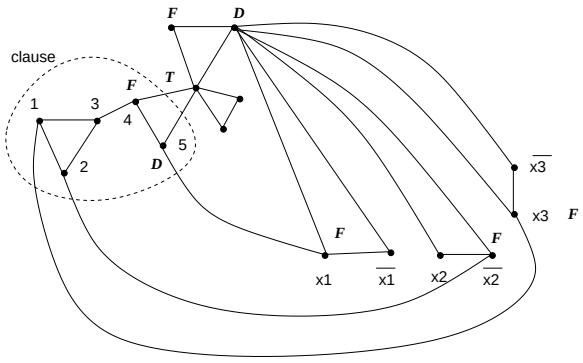


Following are the possible colour assignments:

u_5	u_4	u_3	u_1	u_2	Literal coloured T
F	D				l_3
D	F	T	F	D	l_1
D	F	T	D	F	l_2
D	F	D	T	F	l_2
D	F	D	F	T	l_1

So one literal must be coloured T. The claim is that 3CNF formula ϕ is satisfiable if and only if the graph is 3-colourable.

Following figure shows an example with a clause $C = x_1 \vee \overline{x_2} \vee x_3$.



If all three literals are *false*, then node 1 and 2 are coloured with T and D. But that needs a 4th colour for node 3. But the table shows that if one of the literal is *true* i.e. coloured with T. then the graph is 3-colourable. QED.

Definition 12. $SUBSET-SUM = \{< S, t >:\ S = \{x_1, \cdots, x_k : x_i \in \mathbb{N}\}$ is a multiset and for some $\{x_{i_1}, \cdots, x_{i_l}\} \subseteq S,\ \sum_{j=1}^{l} x_{i_j} = t\}$.

Proposition 4. $SUBSET-SUM$ is **NP**-*complete*.

Proof: The certificate for $SUBSET-SUM$ is C, a collection of elements of S. This can be verified in polynomial time. Following is a verifier.
$V =$ "On input $<< S, t >, C >$

1. Test whether $C \subseteq S$.
2. Test whether $\sum C = t$.
3. *Accept* if both are *true*, else *reject*."

So $SUBSET-SUM \in$ **NP**.

We reduce a $3SAT$ formula ϕ to an instance of a $SUBSET-SUM$ problem in polynomial time i.e. $f(\phi) =< S, t >$. The formula ϕ is satisfiable if and only if $< S, t > \in SUBSET-SUM$. This shows that $SUBSET-SUM$ is **NP**-*hard* and so it is **NP**-*complete*.
Let the variables of ϕ be $x_1, \cdots, x_l$ and the clauses be $C_1, \cdots, C_k$. Following table shows the elements of S and the value t constructed from the formula ϕ such that $< S, t > \in$ SUBSET-SUM if and only if ϕ is satisfiable.

Each row of the table corresponds to a decimal number. Numbers from each row other than t is a member of S. Elements of S use digits 0 and 1 and t uses digits 1 and 3. Each 'blank' correspond to zeros.

(a) For each variable x_i there are a pairs of numbers y_i and z_i. The digits of each of them is partitioned in to two parts, the *variable* part (left side) and the *clause* part.

(b) The digit in $T[y_i, x_i] = T[z_i, x_i] = 1$. All other digits in the *variable* part are 0's. If the truth value of $x_i \leftarrow 1$, y_i from S is selected, otherwise z_i is selected.

(c) The digits in $T[y_i, c_j] = 1$ if the clause c_j has the literal x_i. The digit in $T[z_i, c_j] = 1$ if the clause c_j has the literal $\overline{x_i}$. Other digits are 0's.

(d) S also contains a pair of numbers g_j and h_j for each clause c_j. The digit in $T[g_j, c_j] = T[h_j, c_j] = 1$. All other digits of these numbers are 0's.

(e) The digits in the *variable* part of t are all 1's and the digits in the *clause* part of t are all 3's.

(f) The target is to get the value of t after adding the selected numbres y_i or z_i for $i = 1, 2, \cdots, l$ (each variable) and zero, one or both of g_j, h_j for $j = 1, 2, \cdots, k$ (each clause).

Table $(T)VariablePart$

	Variables						*Clauses*				
	x_1	x_2	x_3	x_4	$\cdots$	x_l	c_1	c_2	c_3	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	1	1	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	0	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	1	$\cdots$	0
y_3			1	0	$\cdots$	0	0	1	0	$\cdots$	0
z_3			1	0	$\cdots$	0	1	0	1	$\cdots$	0
$\vdots$	$\vdots$		$\vdots$		$\vdots$			$\vdots$		$\vdots$	
y_l					$\cdots$	1	0	0	0	$\cdots$	0
z_l					$\cdots$	1	0	0	0	$\cdots$	0

Table (T) Clause Part

	Variables						*Clauses*				
	x_1	x_2	x_3	x_4	$\cdots$	x_l	c_1	c_2	c_3	$\cdots$	c_k
g_1					$\cdots$	0	1	0	0	$\cdots$	0
h_1					$\cdots$	0	1	0	0	$\cdots$	0
g_2					$\cdots$	0	0	1	0	$\cdots$	0
h_2					$\cdots$	0	0	1	0	$\cdots$	0
g_3					$\cdots$	0	0	0	1	$\cdots$	0
h_3					$\cdots$	0	0	0	1	$\cdots$	0
$\vdots$	$\vdots$		$\vdots$		$\vdots$			$\vdots$		$\vdots$	
g_k					$\cdots$	0	0	0	0	$\cdots$	0
h_k					$\cdots$	0	0	0	0	$\cdots$	0
t	1	1	1	1	$\cdots$	1	3	3	3	$\cdots$	3

Consider $\phi = C_1 \wedge C_2 \wedge C_3$ where $C_1 = x_1 \vee \overline{x_2} \vee \overline{x_3}$, $C_2 = \overline{x_1} \vee x_2 \vee x_3$ and $C_3 = \overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$. A satisfying assignment is $x_1 \leftarrow 1$, $x_2 \leftarrow 1$, and $x_3 \leftarrow 0$. We choose y_1, y_2, z_3 (ignore other rows and columns of the table). So far the sum is $100100 + 010010 + 001101 = 111211$. We also choose g_1, g_2, h_2, g_3 and h_3 to make the final sum $t = 111333$.

If ϕ is satisfiable: there is a truth assignment for each variable. If $x_i \leftarrow 1$, we choose the number y_i. If $x_i \leftarrow 0$, we choose thenumber z_i. Whatever be the case, when added we get 1 in first l digits of t.
At least one of the three literals of a clause C_j must be true. It may be due to l_i. If $l_i = x_i$ i.e. $x_i \leftarrow 1$, we have already chosen y_i which has 1 in its c_j column. If $l_i = \overline{x_i}$, we have chosen z_i and it has 1 in its c_j column. The sum of the digits of the column c_j for a satisfying assignment can be 1, 2, or 3. They can all be brought to 3 by adding g_j, h_j. But that is not possible if a clause is unsatisfiable.

If subset of S gives the sum t: for every i either y_i or z_i is chosen, but not both, as first l digits of t are all 1's. In column c_j at most 2 can be supplied from g_j and h_j. So 1 must come from the literal of a clause. So the clause is satisfied. QED.

Definition 13. $dHAMPATH = \{< G, s, d >: G$ is a directed graph with a Hamiltonian path from s to $d\}$.

Proposition 5. $dHAMPATH$ is **NP**-*complete.*

Proof: It is clear that dHAMPATH is in **NP**. A sequence of vertices on the path is a certificate. This can be verified in polynomial time.

We reduce 3SAT[9] to dHAMPATH in polynomial time. Consider a 3CNF formula with m clauses and n variables, $x_1, \cdots, x_n$.

There is a starting vertex labelled with s and an end vertex labelled with d.

For every variable x_k there is a *doubly linked chain-graph* of $3m+1$ vertices.

There is a vertex $s_{i(i+1)}$ between every pair of doubly linked chain-graphs corresponding to variables x_i and x_{i+1}, $1 \leq i < n$. There are directed edges, from s to the two ends of the *chain-graph* of x_1, from $s_{i(i+1)}$ to the two ends of the *chain-graph* of x_{i+1}, from the two ends of the *chain-graph* of x_i to $s_{i(i+1)}$, $1 \leq i < n$, and from two ends of the *chain-graph* of x_n to d.

For every clause there is a vertex. Call them $c_1, \cdots, c_m$. Each doubly linked chain-graph corresponding to a variable has a pair of nodes i corresponding to a clause. Every such pair is separated by a node, and there are two terminal nodes. This accounts for the number $3m + 1$,

$$\bigcirc \bigcirc_1 \bigcirc_1 \bigcirc \bigcirc_2 \bigcirc_2 \bigcirc \cdots \bigcirc \bigcirc_m \bigcirc_m \bigcirc.$$

If a clause c_j has x_i, then there is a directed edge from the left node of the pair $\bigcirc_i \bigcirc_i$ of the variable graph of x_i to c_j and a directed edge from c_j to the right node of the pair. If it is $\bar{x_i}$ then these two directed edges are reversed.

If there is a satisfying assignment of a 3CNF formula, then every variable x_i is either 1 or 0. If $x_i \leftarrow 1$, then the path starts from the left end of the *doubly linked graph* of x_i. If $x_i \leftarrow 0$, then the path starts from the right end of the *doubly linked graph* of x_i.

So there is a path from s through different variable nodes to d. To cover the nodes corresponding to the clauses, take one literal per clause that makes it *true*. Let the literal l_i (x_i or $\bar{x_i}$) is true for the clause c_j. Break the path of the *doubly linked graph* of x_i and include c_j in it.

In the other direction, if there is a Hamiltonian path from s to d, then there is a *truth value* assignment for the formula.

The number of vertices of the formula is $2 + m + (3m + 1) + (n - 1)$. So the encoding of the graph is a polynomial over the encoding of the formula.

QED.

1.6 coNP, EXP, and NEXP

The class **coNP** is the collection of subsets of $\{0,1\}^*$ whose complements are in **NP**.

$$\mathbf{coNP} = \{L \subseteq \{0,1\}^* : \overline{L} \in \mathbf{NP}\}.$$

The class **P** is closed under complementation, so $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$. We already know that the following language are in **coNP**.

- Any language in **P** e.g. PRIME.
- $\overline{SAT} = \{\phi : \phi \text{ is unsatisfiable}\}$.
- $\overline{INDSET}$, $\overline{VERTEX - COVER}$, $\overline{CLIQUE}$ etc.

We can define the class **coNP** using a deterministic verifier.

[9] Actually we reduce SAT formula.

Definition 14. A set $L \subseteq \{0,1\}^*$ is in **coNP** if and only if there is a polynomial $p : \mathbb{N}_0 \to \mathbb{N}_0$ and a polynomial time bounded DTM so that for all $x \in \{0,1\}^*$,

$$x \in L \text{ if and only if } \forall w \in \{0,1\}^{p(|x|)},\ M \text{ accepts } < x, w >.$$

This is actually negation of the definition of **NP**.
If $L \in$ **coNP** then $\overline{L} \in$ **NP**. For all $x \in \{0,1\}^*$,

$$x \notin L \text{ if and only if } x \in \overline{L}.$$

There is a polynomial time bounded deterministic Turing machine V, a polynomial $p(n)$, and a witness $w \in \{0,1\}^{p(|x|)}$, such that V accepts $< x, w >$ if and only if $x \in \overline{L}$ i.e.

$$x \notin L \text{ if and only if } \exists w \in \{0,1\}^{p(|x|)},\ V \text{ accepts } < x, w > .$$

Equivalently,

$$x \in L \text{ if and only if } \neg(\exists w \in \{0,1\}^{p(|x|)},\ V \text{ accepts } < x, w >),$$

i.e.

$$x \in L \text{ if and only if } \forall w \in \{0,1\}^{p(|x|)},\ \overline{V} \text{ accepts } < x, w >,$$

where $\overline{V}$ is same as V in all respect, but the *accept* and *reject* states exchanged.

A set L is **coNP** complete if (i) it is in **coNP**, and (ii) every set L' in **coNP** is Karp-reducible to L.

Proposition 6. Following language is **coNP**-*complete*.

$$TAUTOLOGY = \{\phi :\ \phi \text{ is a Boolean formula satisfiable by any assignment}\}.$$

Note that a formula $\phi \in$ TAUTOLOGY if and only if $\neg\phi$ is *unsatisfiable*.
Proof: If ϕ is a Boolean formula with n variables, then it is a *tautology* if and only if it is satisfied by any assignment of n variables. So there is a polynomial time Turing machine V such that for any $x \in \{0,1\}^n$, V will evaluate ϕ with x as assignment to its variables. The formula ϕ is a *tautology* if it evaluates to *true* (i.e. 1) for all x. So $TAUTOLOGY \in$ **coNP**.

We now show that every language $L \in$ **coNP** is Karp-reducible to TAUTOLOGY. We take $\overline{L}$, the complement of L. If $L \in$ **coNP**, then $\overline{L} \in$ **NP** i.e. $x \in L$ if and only if $x \notin \overline{L}$ if and only if $\overline{f}(x) = \phi_x$ is *unsatisfiable* if and only if $\neg\phi_x = f(x)$ is a *tautology*.
The reduction is, for all $x \in \{0,1\}^*$, create $\neg\phi_x$. QED.

It is clear that if **P** = **NP**, then **NP** = **coNP** = **P**.

Definition 15. We define the class **NEXP** = $\bigcup_{c\geq 1}$ **NTIME**(2^{n^c}). By definition we have **P** $\subseteq$ **NP** $\subseteq$ **EXP** $\subseteq$ **NEXP**. We prove the following proposition.

Proposition 7. If **EXP** $\neq$ **NEXP**, then **P** $\neq$ **NP**.
Proof: We prove the contrapositive statement. We assume **P** = **NP** and prove that **EXP** = **NEXP**.
Let $L \in$ **NTIME**(2^{n^c}). So a non-deterministic Turing machine N decides L in time 2^{n^c}. We define the language

$$L_{pad} = \left\{ < x, 1^{2^{|x|^c}} > :\ x \in L \right\},$$

and claim that $L_{pad} \in \mathbf{NP}$. The non-deterministic Turing machine N_{pad} for L_{pad} is as follows.
N_{pad}: input y

1. Nondeterministically it guesses a z, and computes $2^{|z|^c}$, so that $y =< z, 1^{2^{|z|^c}} >$. It *rejects* the input if no such z is found.
2. Otherwise, simulate N on z for $2^{|z|^c}$ steps.
3. If N accepts z, then *accept*, else *reject*.

The running time of N_{pad} is polynomial in $|y|$, so $L_{pad} \in \mathbf{NP}$. But according to our assumption $L_{pad} \in \mathbf{P}$. But then $z \in L$ if and only if $< z, 1^{2^{|x|^c}} > \in L_{pad}$. The padding string can be attached to z in exponential time and membership of $< z, 1^{2^{|x|^c}} > \in L_{pad}$ in L_{pad} can be tested in polynomial (on the length of $< x, 1^{2^{|x|^c}} >$) time.
Therefore the membership of x in L is determined in exponential (on the length of x). So $L \in \mathbf{EXP}$ i.e. $\mathbf{NEXP} \subseteq \mathbf{EXP}$. QED.

References

[OD] Computational Complexity *A Conceptual Perspective* by *Oded Goldreich*, Pub. Cambridge University Press, 2008, ISBN 978-0-521-88473-0.

[MS] *Theory of Computation* by Michael Sipser, (3rd. ed.) Pub. Cengage Learning, 2013, ISBN 978-81-315-2529-6a .

[SABB] *Computational Complexity, A Modern Approach* by Sanjeev Arora & Boaz Barak, Pub. Cambridge University Press, 2009, ISBN 978-0-521-42426-4.

[CHP] Computational Complexity by *Christos H Papadimitriou*, Pub. Addision-Wesley, 1994, ISBN 0-201-53082-1.

[AW] Mathematics + Computation *A Theory Revolutionizing Technology and Science* by *Avi Wigderson*, Pub. Princeton University Press, 2019, ISBN 978-0-691-18913-0.

[DCK1] *Theory of Computation* by Dexter C Kozen, Pub. Springer, 2006, ISBN 978-81-8128-696-3.