UNIT V

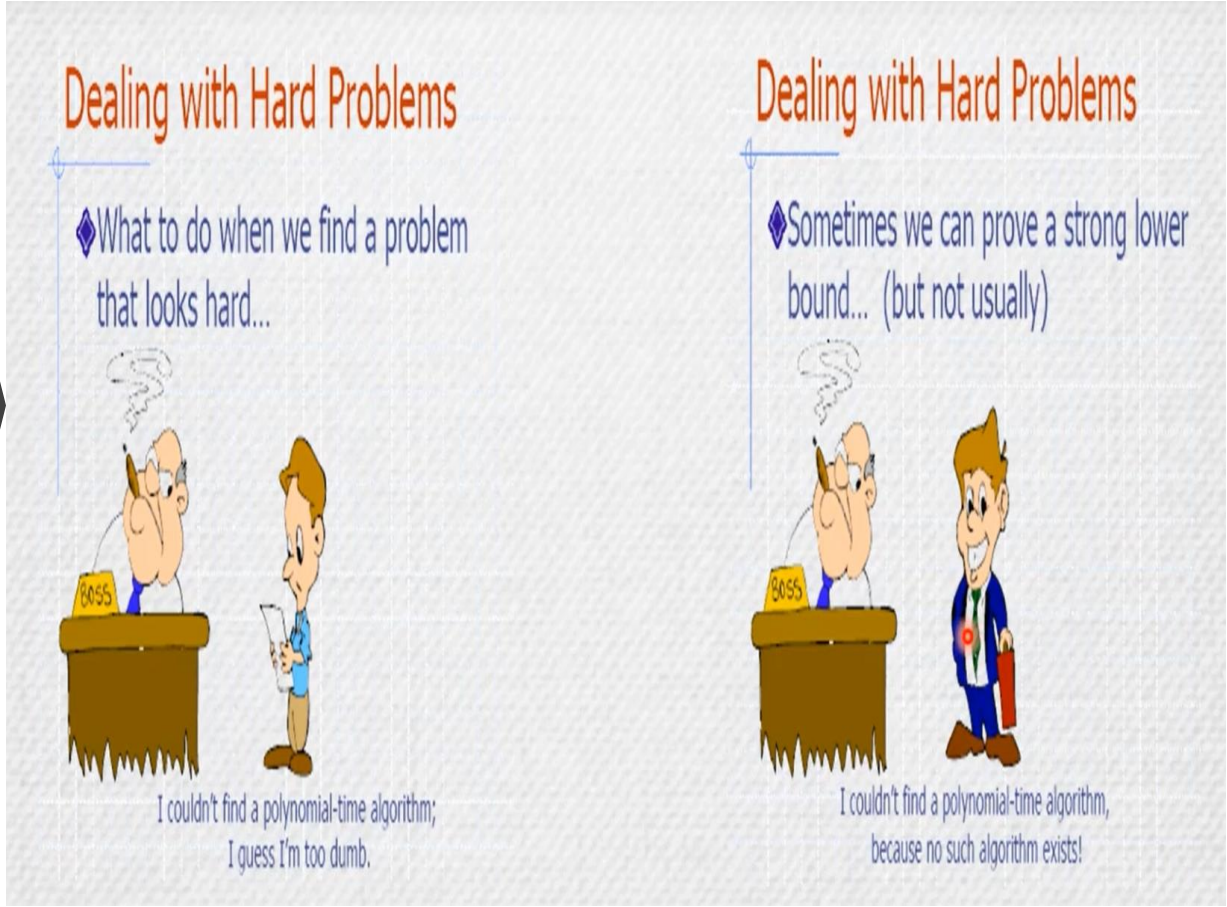
NP Hard and NP Completeness

Introduction

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

What to do if your boss gives you a hard problem?

What to do if your boss gives you a hard problem?



One morning in the office

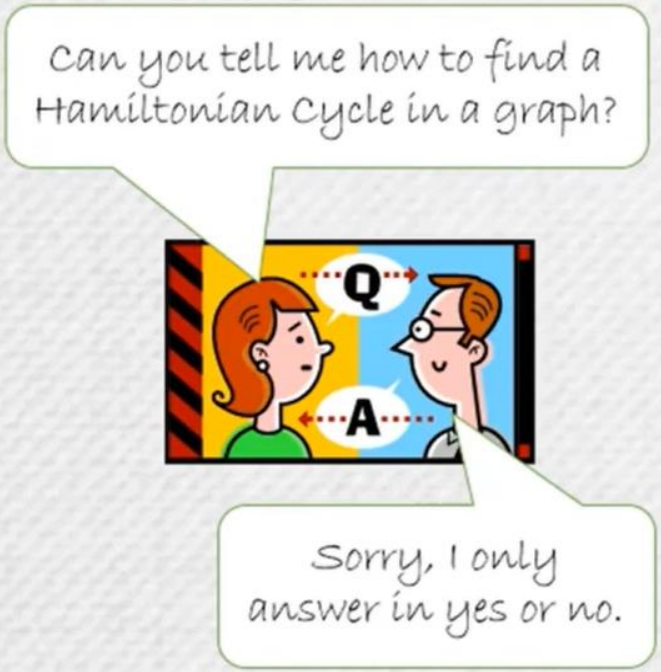
After a few tea breaks and brain storming sessions

After a few tea breaks and brain storming sessions

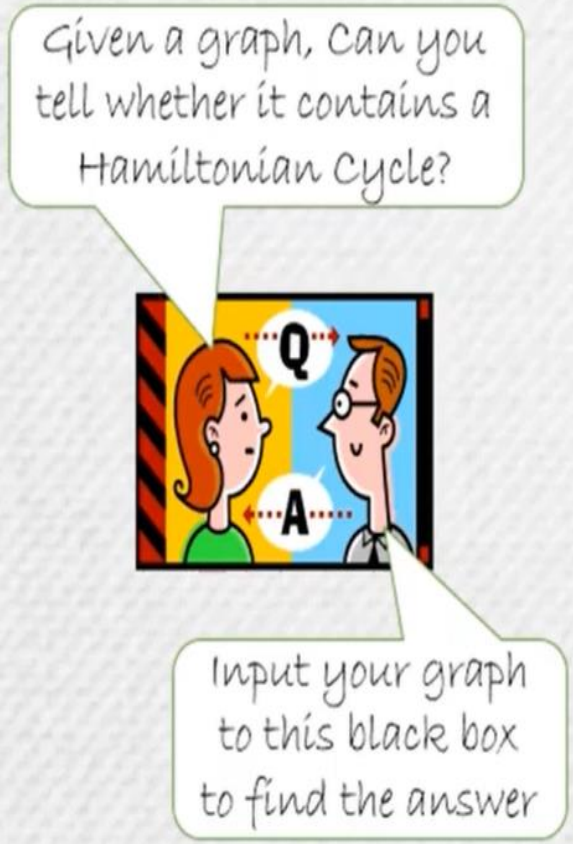
Class P

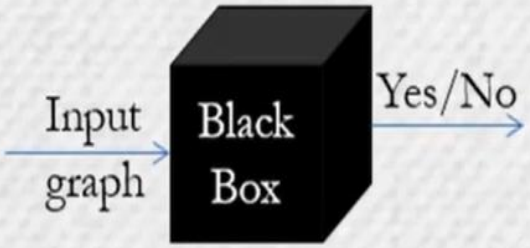
- The class P consists of those problems that can be solved in polynomial time. i.e. these problems can be solved in time $O(n^k)$ where k is a positive constant and n is the input size.
- For example, Linear Search - O(n), Quick Sort - $O(n\log_2 n)$ and Matrix Chain Multiplication - $O(n^3)$ belong to the P class of problems.
- We say that an "efficient" algorithm exists for all P problems. In other words, all algorithms running in polynomial time are "efficient".
- A problem, for which an efficient algorithm is not known, is termed as "hard".

Decision Problem



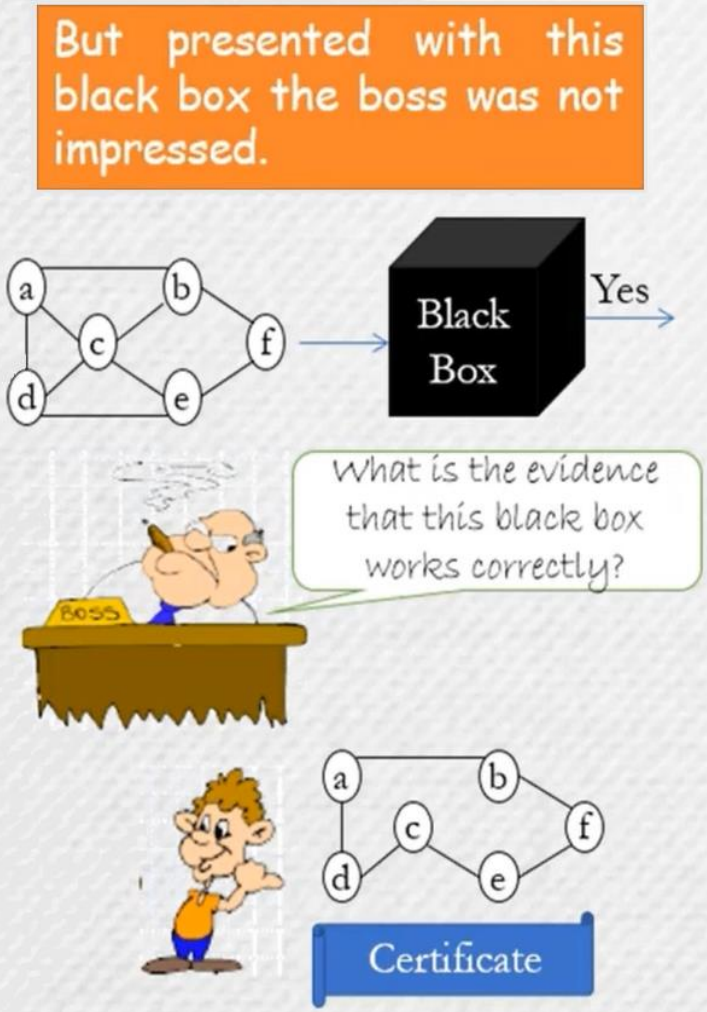
- Problems for which answer is either "yes" or "no"
- Example:
  1. Given a string X and a string Y, does X appear as a sub-string of Y?
  2. Given two sets S & T, do S and T contain the same set of elements?
  3. Given a graph G with integer weights on its edges and an integer k, does G have a minimum spanning tree of weight at most k?
- Example 3 illustrates how we can turn an optimization problem into a decision problem.
- For the sake of simplicity we will only focus on Decision Problems.





- Given a decision problem X, its input can be encoded as a finite binary string s; we say that $s \in X$.
- Let us suppose that we want to find a Hamiltonian cycle in a graph. The corresponding decision problem (X) may be stated as: *"Given a graph, does it contain a Hamiltonian cycle?"* Its input s is the binary representation of the given graph (i.e. its adjacency matrix).
- An algorithm A for X, receives an input string s and returns the value "yes" or "no". We will denote the returned value by A(s). A is said to solve X, if A(s) = yes ($s \in X$).
- For example if an algorithm A, answers the question *"Given a graph, does it contain a Hamiltonian cycle?"* as yes, then it solves the problem (i.e. it finds a Hamiltonian cycle in the graph).

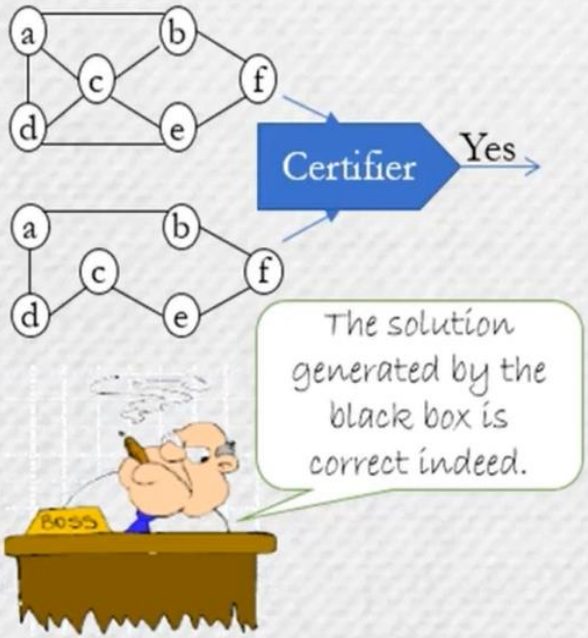
Certificate



- In order to check the correctness of a solution, we need the input string s, as well as a "certificate" string t, that contains the evidence that s is a "yes" instance of X.
- For example, let X be the problem *"Given a graph, does it contain a Hamiltonian cycle?"*. Let its input s be the graph, shown in the picture.
- Let an algorithm A of X answer A(s) = yes, and as an evidence the (binary representation of) string t = "abfecda" is given, i.e. it is claimed that "abfecda" forms a Hamiltonian cycle in the given graph. Then t is called a certificate for the problem X and input s.

Efficient certifier

Given an input and a certificate, the boss had a certifier to check the correctness of the algorithm.



- Given a decision problem X and an algorithm A, with input s and certificate t, such that A(s) = yes, we would want to verify whether the solution is correct.
- Let B be the verification algorithm, which takes two input strings s and t and replies "yes" if indeed A produces a correct solution of X.
- B is said to be an efficient certifier if it runs in polynomial time and $|t| \leq p(|s|)$, where p is a polynomial function.
- Note that, existence of an efficient certifier B gives us a brute force method for solving X: *"On input s, try all possible strings t of length* $\leq p(|s|)$*, and check if B(s, t) = yes"*. But it may not be an efficient algorithm, since it is up to us to find a string t, which will cause B(s, t) to say "yes" and there are exponentially many possibilities for t.

Class NP

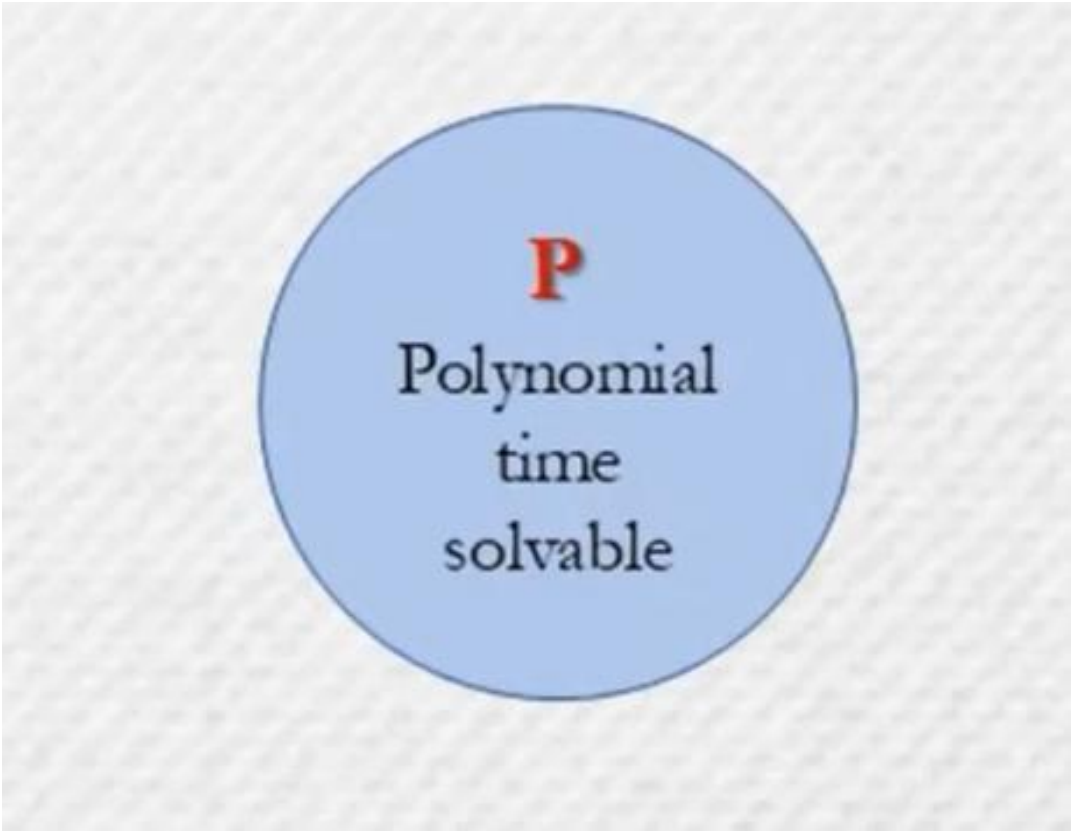
- NP is a set of all problems for which there exists an efficient certifier. In other words, given a decision problem X, with an input s and a certificate t, such that $|t| \leq p(|s|)$, if we can verify that s is a yes instance of X in polynomial time, then we say $X \in NP$.
- ***Theorem:*** $P \subseteq NP$
- ***Proof:***

  Let $X \in P$, then there exists an algorithm A for X, which solves X in polynomial time, i.e. $A(s) = yes$, $s \in X$.

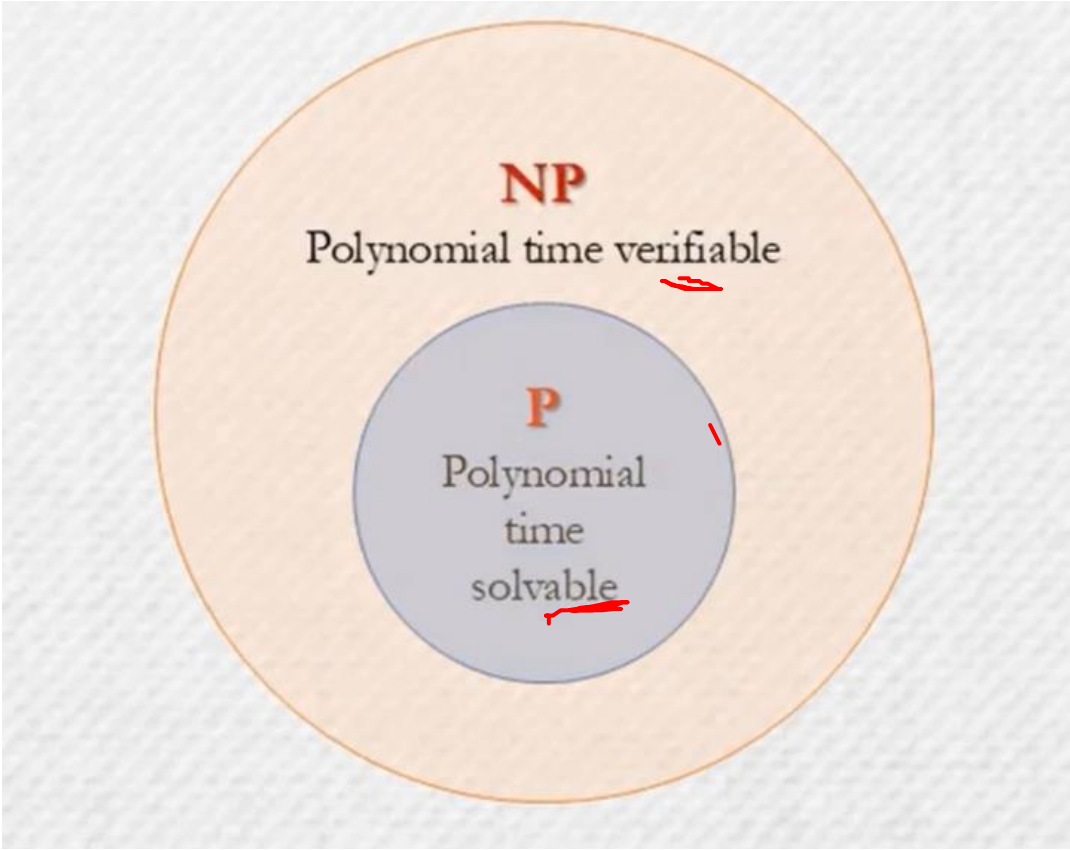
  We can design a certifier B(s, t) for X, which ignores t and solves X using A, and checks whether $A(s) = yes$. Clearly B runs in polynomial time, since A does.

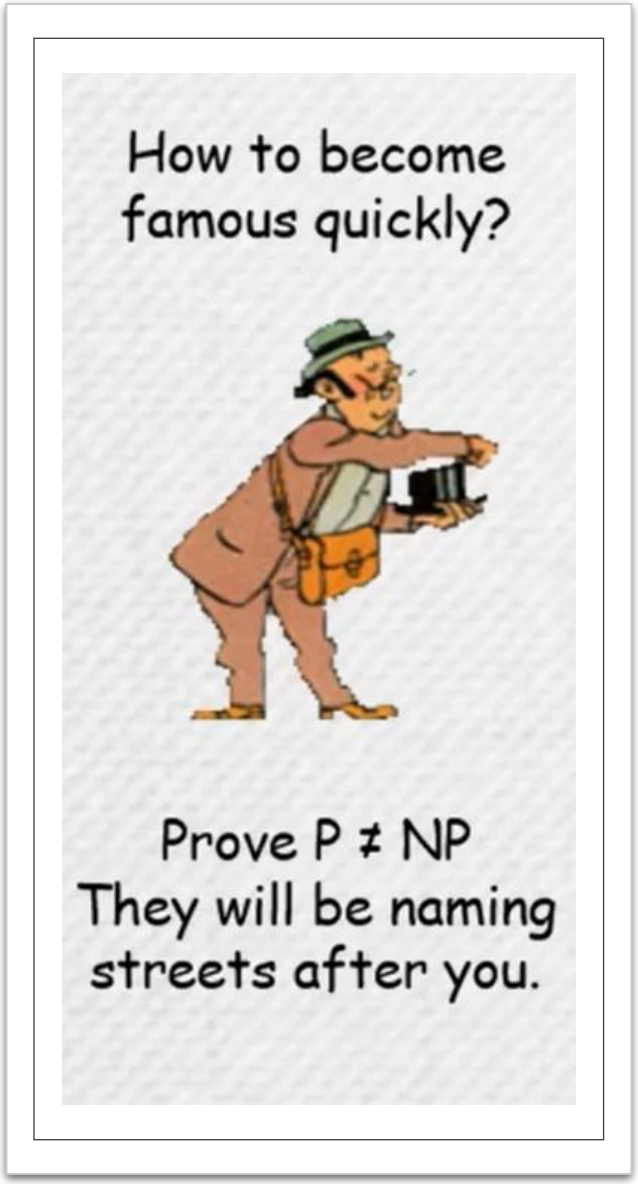
  So, B is an efficient certifier of X, and thus $X \in NP$ ■

P Vs. NP



P Vs. NP




How to become famous quickly?
Prove P ≠ NP
They will be naming streets after you.

P=NP?

Nondeterministic algorithms

Deterministic algorithms

- Algorithms with uniquely defined results
- Predictable in terms of output for a certain input

Nondeterministic algorithms are allowed to contain operations whose outcomes are limited to a given set of possibilities instead of being uniquely defined

Nondeterministic algorithms are specified with the help of three new O(1) functions

1. choice (S)
   - Arbitrarily chooses one of the elements of set S
   - x = choice(1,n) can result in x being assigned any of the integers in the range [1, n], in a completely arbitrary manner
   - No rule to specify how this choice is to be made
2. failure()
   - Signals unsuccessful completion of a computation
   - Cannot be used as a return value
3. success()
   - Signals successful completion of a computation
   - Cannot be used as a return value
   - If there is a set of choices that leads to a successful completion, then one choice from this set must be made

- A nondeterministic algorithm terminates unsuccessfully iff there exist no set of choices leading to a success signal
- A machine capable of executing a nondeterministic algorithm as above is called a nondeterministic machine
- Nondeterministic search of x in an unordered array A with $n \geq 1$ elements

* Determine an index j such that A[j] = x or j = −1 if x doesn't belong to A

```
algorithm nd_search ( A, n, x )
{ // Non-deterministic search
  // Input: A: Array to be searched
  // Input: n: Number of elements in A
  // Input: x: Item to be searched for
  // Output: Returns -1 if item does not exist, index of
     item otherwise
    int j = choice ( 0, n-1 );
    if ( A[j] == x )
    {
          cout << j;
          success();
    }
    cout << -1;
    failure();
}
```

* By the definition of nondeterministic algorithm, the output is -1 iff there is no j such that A[j] = x
* Since A is not ordered, every deterministic search algorithm is of complexity Ω(n), whereas the nondeterministic algorithm has the complexity as O(1)

Nondeterministic sorting algorithm // Sort n positive integers in nondecreasing order

```
algorithm nd_sort ( A, n )
{// Initialize B[]; B is used for convenience
  // It is initialized to 0 though any value not in A[] will suffice
    for ( i = 0; i < n; B[i++] = 0; );
    for ( i = 0; i < n; i++ )
    {
          j = choice ( 0, n - 1 ); // Make sure that B[j] has not been used already
          if ( B[j] != 0 )
                failure();
          B[j] = A[i];
    } // Verify order
    for ( i = 0; i < n-1; i++ )
          if ( B[i] > B[i+1] )
                failure();
          write ( B );
          success();
}
```

* Complexity of nd_sort is Θ(n)
* Best-known deterministic sorting algorithm has a complexity of Ω(n lg n)

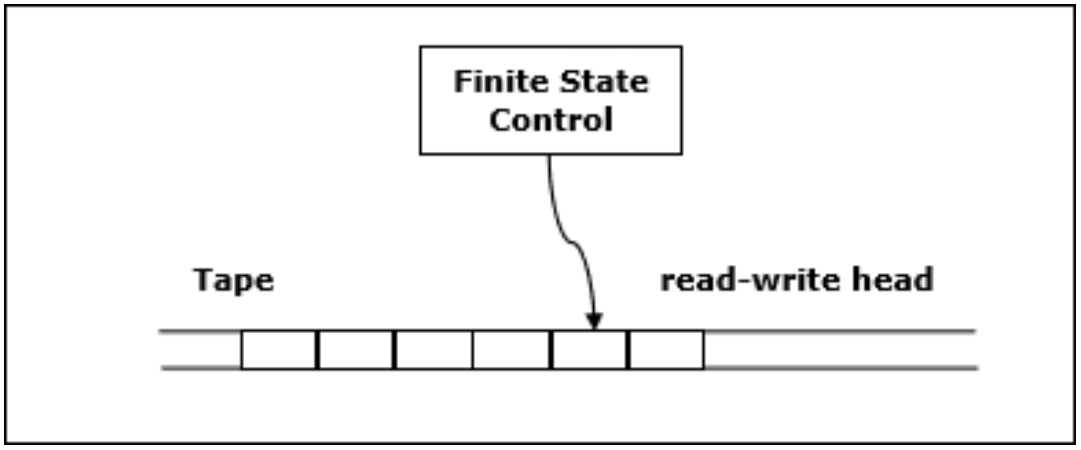
- Nondeterministic knapsack decision problem:

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

A successful termination is possible iff the answer to the decision problem is yes. The time conplexity is O(n).

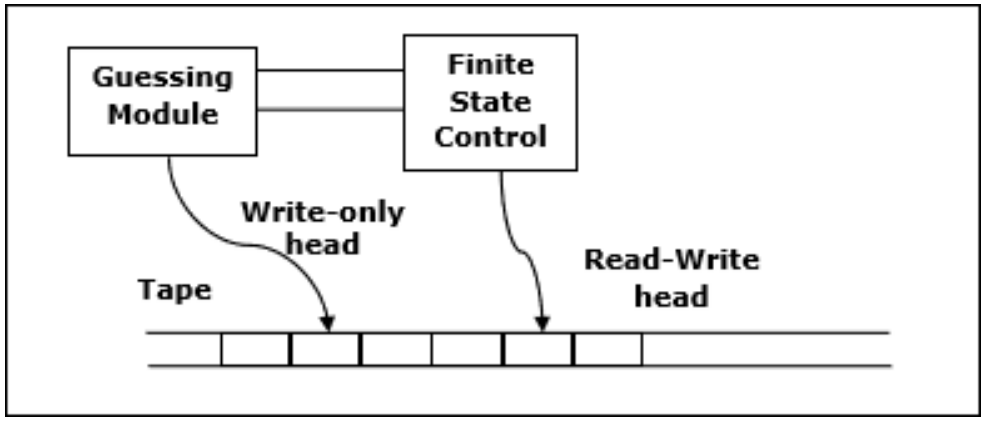
Deterministic vs. Nondeterministic Computations

- Deterministic Turing Machine
- One of these models is deterministic one-tape Turing machine. This machine consists of a finite state control, a read-write head and a two-way tape with infinite sequence.
- Following is the schematic diagram of a deterministic one-tape Turing machine.



- A program for a deterministic Turing machine specifies the following information −
  - A finite set of tape symbols (input symbols and a blank symbol)
  - A finite set of states
  - A transition function
- In algorithmic analysis, if a problem is solvable in polynomial time by a deterministic one tape Turing machine, the problem belongs to P class.

- Nondeterministic Turing Machine
- To solve the computational problem, another model is the Non-deterministic Turing Machine (NDTM). The structure of NDTM is similar to DTM, however here we have one additional module known as the guessing module, which is associated with one write-only head.
- Following is the schematic diagram.



- If the problem is solvable in polynomial time by a non-deterministic Turing machine, the problem belongs to NP class.

- Some problems are *intractable (hard)* :
  as they grow large, we are unable to solve them in reasonable time
- What constitutes reasonable time?
  - Standard working definition: *polynomial time*
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k

  Suppose if we have $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$, $O(2^n)$, $O(n^n)$, $O(n!)$ then
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Polynomial-Time Algorithms

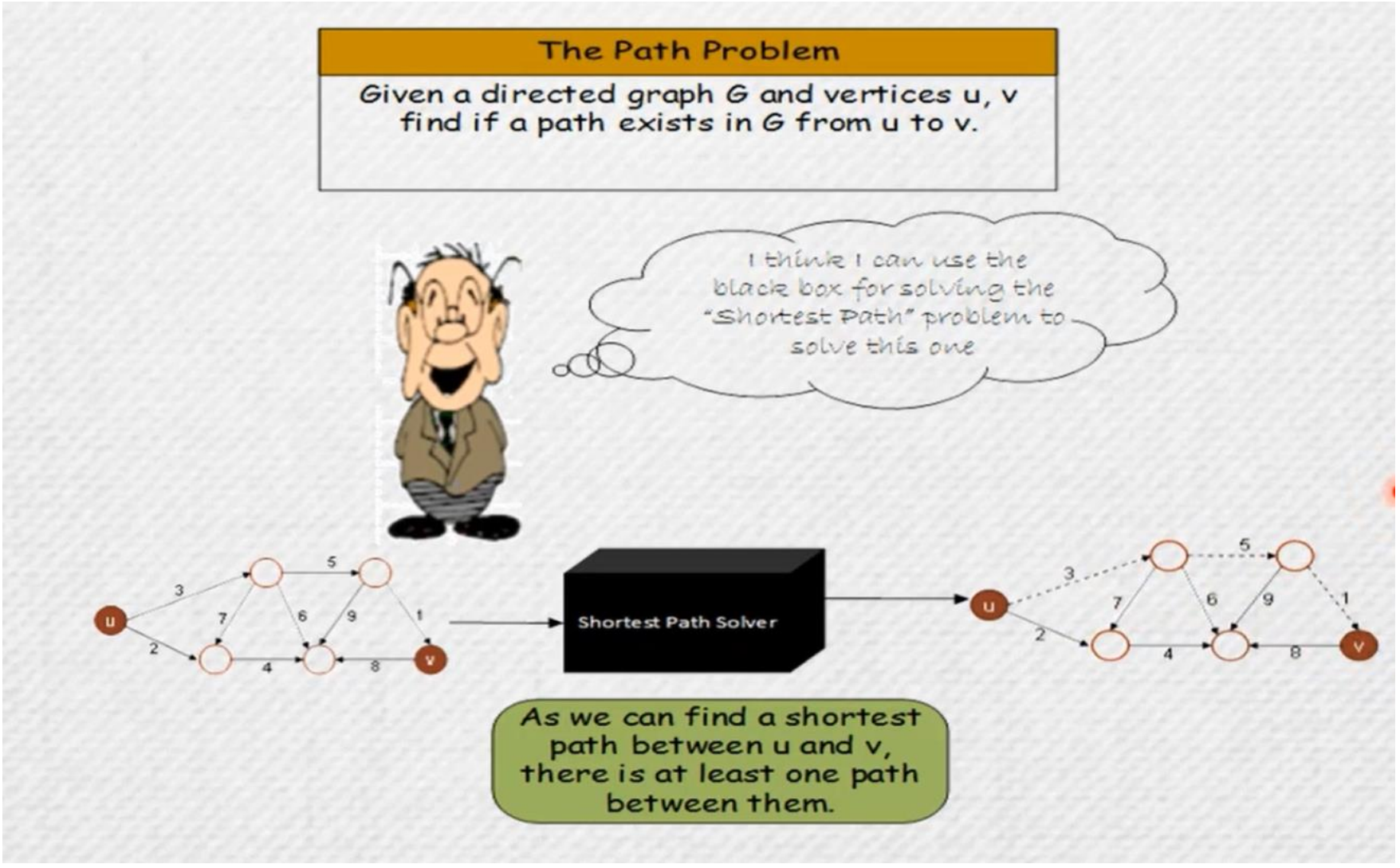
- Are some problems solvable in polynomial time?
  - Of course: many algorithms we've studied provide polynomial-time solutions to some problems
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
- Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

Optimization/Decision Problems

- Optimization Problems
  - An optimization problem is one which asks, “What is the optimal solution to problem X?”
  - Examples:
    - 0-1 Knapsack
    - Fractional Knapsack
    - Minimum Spanning Tree
- Decision Problems
  - An decision problem is one with yes/no answer
  - Examples:
    - Whether a given graph can be colored by only 4-colors.
    - Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

- An optimization problem tries to find an optimal solution
- A decision problem tries to answer a yes/no question
- Many problems will have decision and optimization versions
  - Eg: Traveling salesman problem
    - optimization: find hamiltonian cycle of minimum weight
    - decision: is there a hamiltonian cycle of weight $\leqslant$ k
- Some problems are decidable, but *intractable*:
  as they grow large, we are unable to solve them in reasonable time

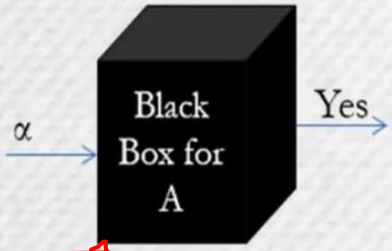
Using a black box to solve another problem

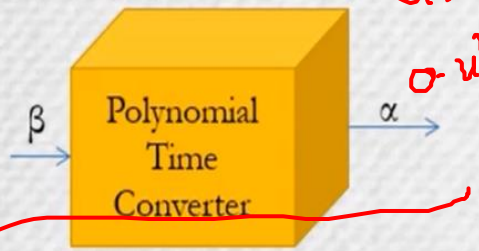


Relative difficulty of problems

- If we have a black box capable of solving the "shortest path" problem, we can use that to solve the "path" problem.
- In general, we say that a particular problem X is at least **as hard as** some other problem Y, if a "black box" capable of solving X, also solves Y.
- In other words, X is powerful enough to let us also solve Y.

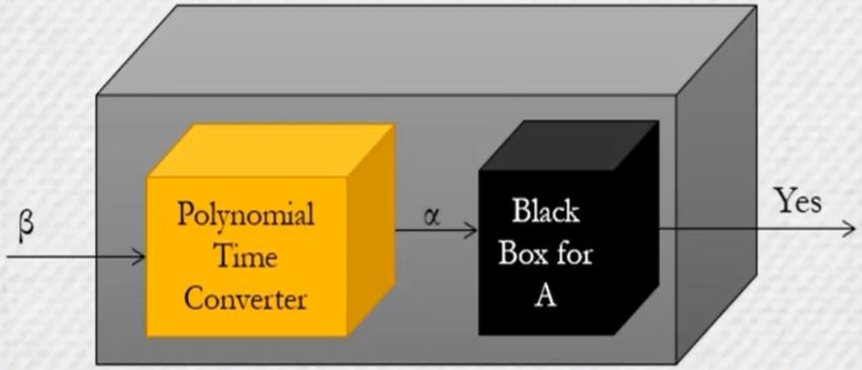
Polynomial time reduction



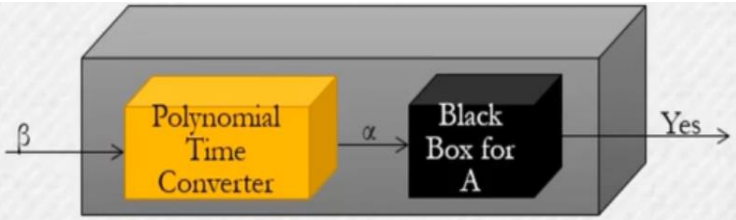


- Let's say we have a black box for solving some decision problem A. That means, if we supply it with an input α of A, it will produce an output, say yes.
- Let B be another decision problem.
- Let us suppose that there is procedure which transforms any input β of B into some input α of A, with the following characteristics:
  i. The transformation happens in polynomial time.
  ii. The outputs are same. That is, the output for α is yes, if and only if the output for β is also yes.
- Such a transformation is called a Polynomial Time Reduction. We say, $\mathbf{B} \leq_{\mathbf{p}} \mathbf{A}$ (i.e. B is polynomial time reducible to A).

- Note that, if $B \leq_p A$, then we have the following way of using the black box for solving A, to solve B.
  i. Given an input β of B transform to an input α of A in polynomial time
  ii. Use Black Box of A to find the output for input α
  iii. Use the output of the Black Box of A for input α as the output for problem B when the input is β
- Hence $B \leq_p A$ means that A is at least as hard (i.e. may be more) as B.



Using the Black Box for A to construct a Black Box for solving B



- If $X \leq_p Y$, then $Y \in P \Rightarrow X \in P$
- *Proof:*

  $X \leq_p Y$ means we can build an algorithm A for solving X, which has the following two steps.

  1. Use a polynomial time transformer to convert an input of X to an input of Y.
  2. Use the black box for Y to get the solution for the given input.

  By definition step 1 runs in polynomial time. We are given that $Y \in P$, so step 2 runs in polynomial time too.

  Since both the steps in the algorithm A for X can be executed in polynomial time, A is a polynomial time algorithm.

  Thus $X \in P$

- If $X \leq_p Y$, then $X \notin P \Rightarrow Y \notin P$
- *Proof:*

  $X \leq_p Y$ means we can build an algorithm A for solving X, which has the following two steps.

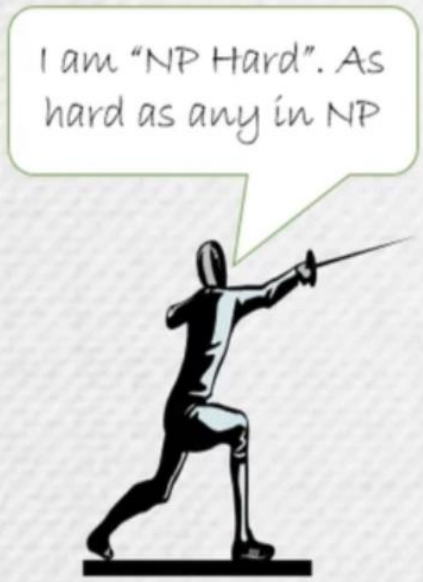
  1. Use a polynomial time transformer to convert an input of X to an input of Y.
  2. Use the black box for Y to get the solution for the given input.

  By definition step 1 runs in polynomial time. We are given that $X \notin P$, so algorithm A does not run in polynomial time.

  Since A consists of two steps and it is non-polynomial, step 2 will necessarily have to be non-polynomial, as step 1 executes in polynomial time. Which means that the black box for Y does not run in polynomial time.

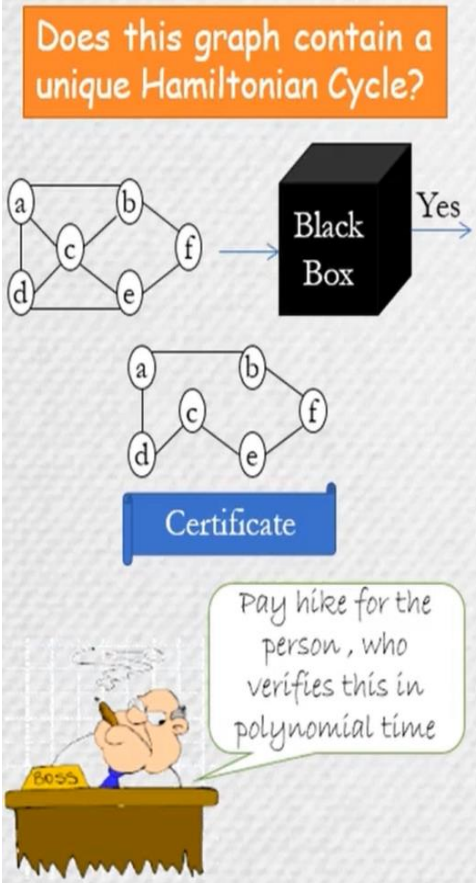
  Thus $Y \notin P$

Class NP Hard



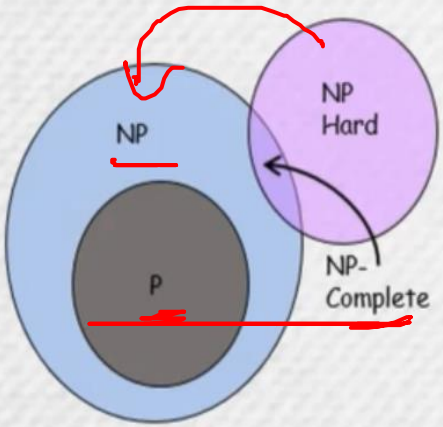
- Let $A \in NP$
- Let B be a problem, such that $A \leq_P B$
- What does that tell us about problem B? It tells us that B is at least as hard as problem A.
- But A is an arbitrarily chosen problem from NP.
- So, what it really means is that B is as hard as any (i.e. the hardest) problem in NP.
- We call such a problem B, an **NP Hard** problem.

Not all NP Hard problems are ion NP



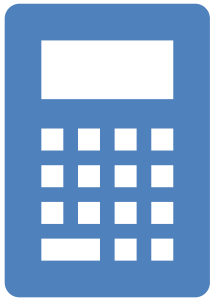
- Let X be an NP Hard Problem.
- Note that it does not necessarily mean that X is in NP.
- Recall that a problem A is in NP if it has an efficient certifier, which means that given an input s and a certificate t, we can verify in polynomial time that s is a yes instance of A.
- But from its definition we can't make such a claim about X.
- For example, suppose that we are given a decision problem: "*For a given graph find if it has a unique Hamiltonian cycle*".
- Let us say that for the input graph s, the black box answers yes, and as evidence we are given the certificate graph t (as shown in the picture).
- Can we verify in polynomial time that the answer is correct?
- Obviously we can't, because we can see that t is a Hamiltonian cycle in s, but whether it is unique, i.e. whether s contains any other Hamiltonian cycle or not cannot be ascertained in polynomial time.
- So the given problem does not belong to NP.

Class NP Complete



- We now know that not all NP Hard problems are in NP, meaning some are and some aren't.
- Let's focus on those NP Hard problems which are also in NP.
- A problem X is called **NP Complete**, if and only if:
  1. *X is NP Hard, and*
  2. $X \in NP$
- In other words, NP Complete problems are the hardest problems in NP.

The Classes NP-hard and NP-complete:

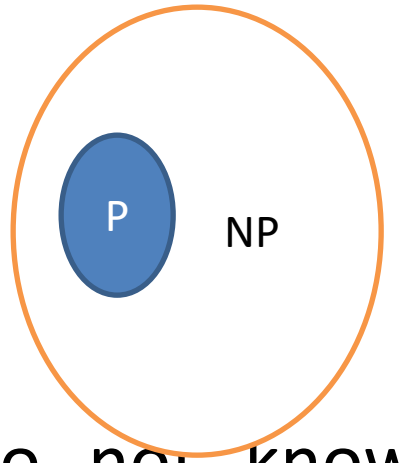


P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. That is the problems can be solved in polynomial time but have a polynomial time checking algorithm i.e., given a solution we can check in polynomial time if that the solution what we are looking for.

- Since deterministic algorithms are just a special case of non deterministic algorithm once we conclude that P is a subset of NP.

  $P \subseteq NP$



- What we do not know and what has become perhaps the most famous unsolved problem in computer science, is perhaps P=NP or P≠NP

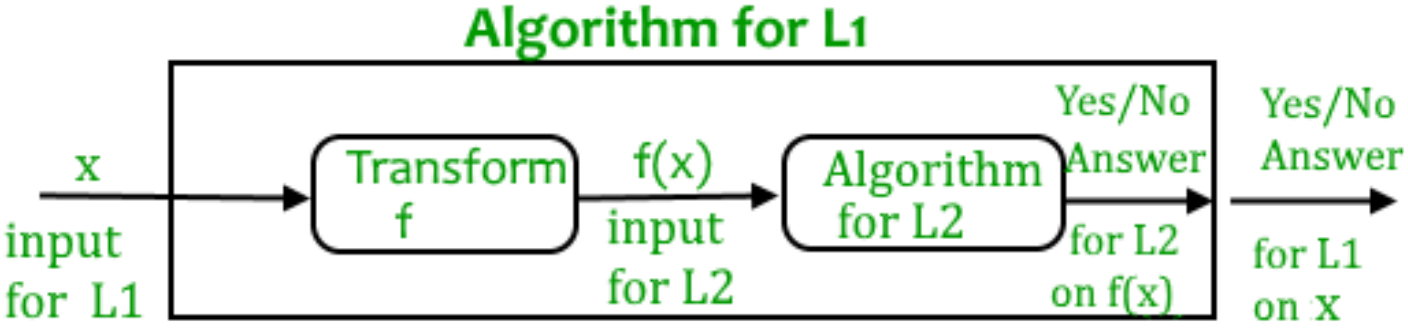
COOK'S THEOREM

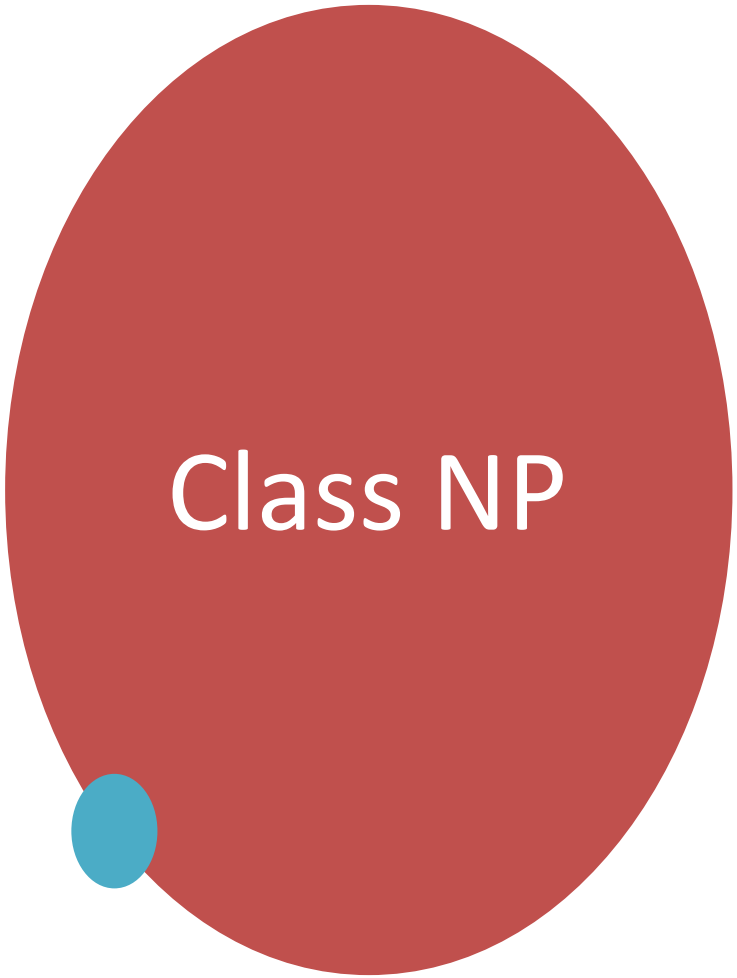
- S. Cook formulated the following question: Is there any single problem in NP such that if we showed it to be in P, then that would imply that P=NP? Cook answered his own question in the affirmative with the following theorem.
- Satisfiability is in P if and only if P=NP.

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.

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



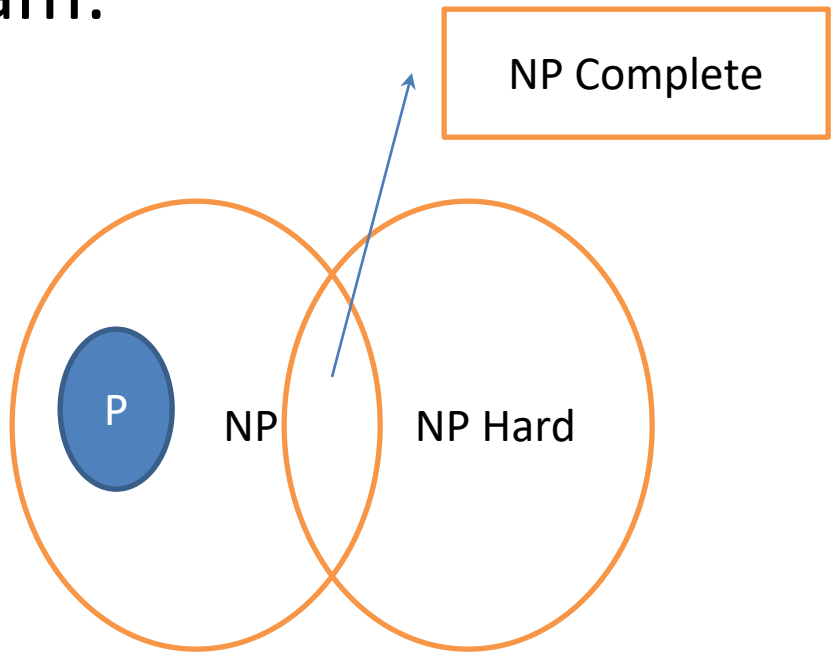


- NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
  1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
  2) Every problem in NP is reducible to L in polynomial.

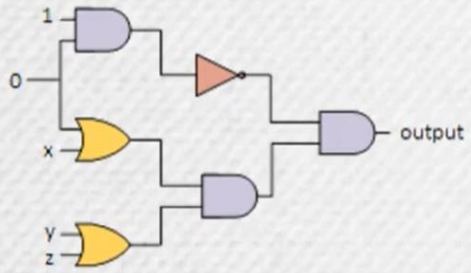
Class NP Hard

- A problem is NP-Hard if it follows property 2 (Every problem in NP is reducible to L in polynomial) mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.

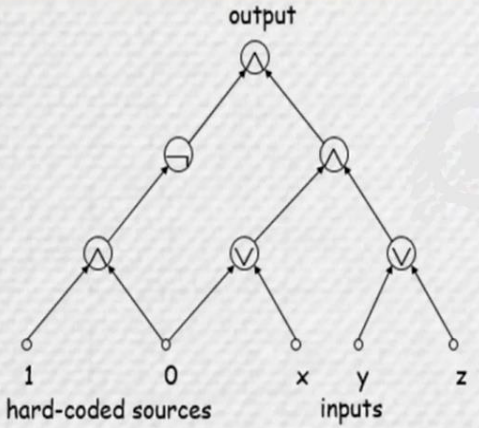
- Venn Diagram:



The Circuit Satisfiability Problem (CSAT)



Is there any assignment of values to inputs x, y, z which would cause the output to be 1?



Let us consider a circuit K, made up of the standard Boolean GATEs, $\wedge$ (AND), $\neg$ (NOT) and $\vee$ (OR). The circuit K can be thought of as a labeled, directed acyclic graph, with the following properties:

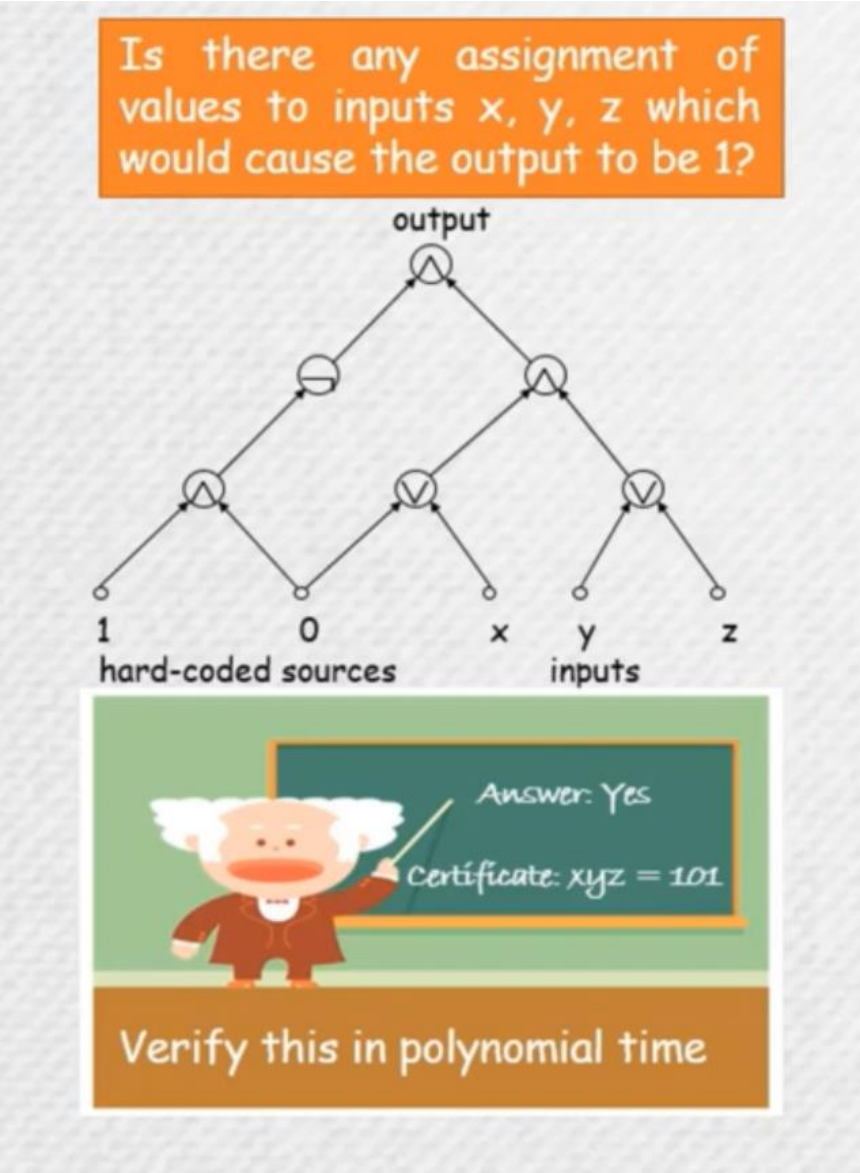
I. The sources in K (i.e. the nodes with no incoming edges) are either labeled with 0/1 or a distinct variable (x/y/z). The variable sources are referred to as inputs of k.

II. Every other node is labeled with one of the Boolean operators $\wedge$, $\neg$, $\vee$.

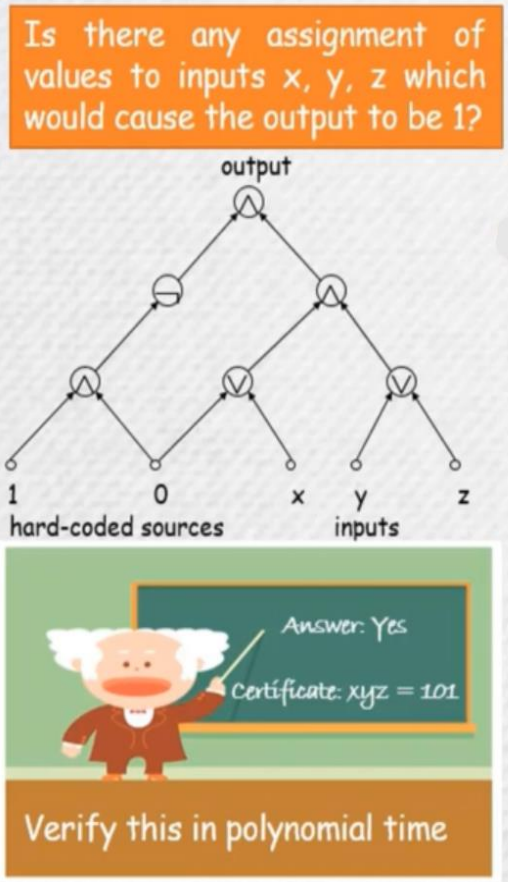
III. There is a single node with no outgoing edges, called the output.

we need to decide whether there is an assignment of values to the inputs that causes the output to be 1.

Try to prove that CSAT belongs to NP



CSAT belongs to NP



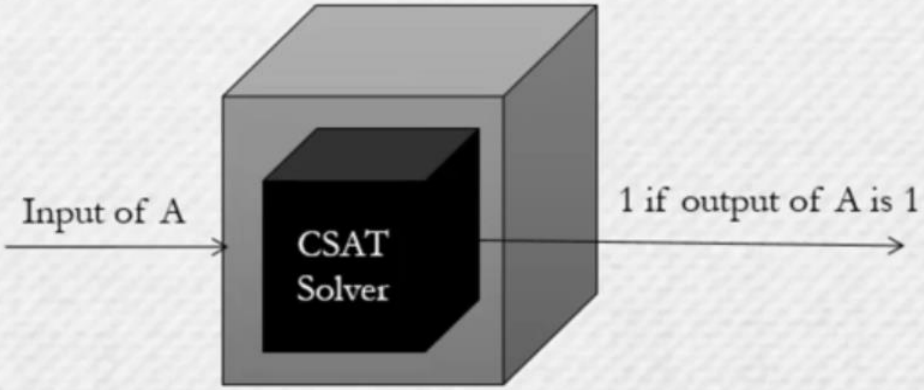
- Let us consider the case where we are supplied with an answer to CSAT problem (for the given graph in the picture) as yes. The certificate given as evidence for the answer is: x = 1, y = 0, z = 1. Let us verify the correctness of the solution.
- If the inputs are assigned the values 1, 0, 1 from left to right, then we get values 0, 1, 1 for the gates in the second row, values 1, 1 for the gates in the third row, and the value 1 for the output.
- The verification process runs in linear time in the number of gates in the circuit.
- Hence CSAT $\in$ NP

How to show CSAT is NP Hard?

Let A be a problem in NP.
i.e. A ∈ NP

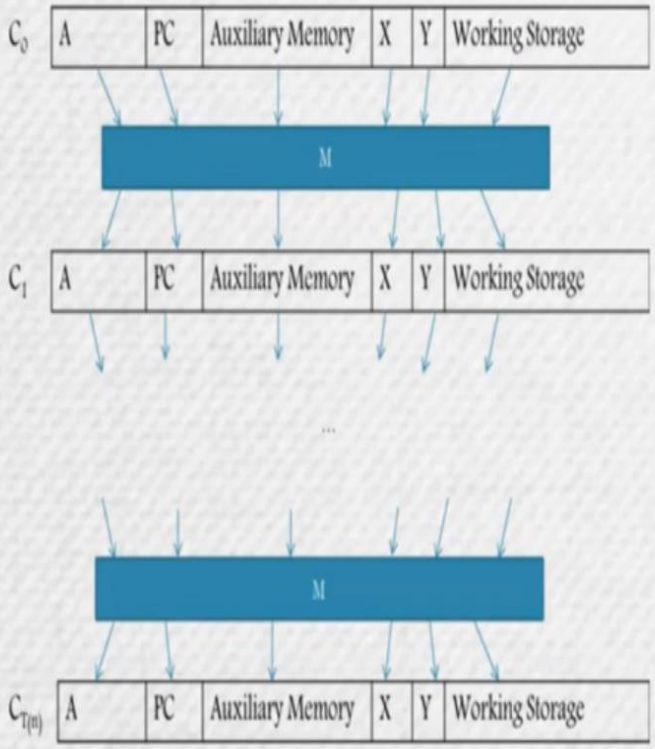
and let us assume that we have a CSAT solver black box

If using the CSAT solver black box, we can build a black box that solves A



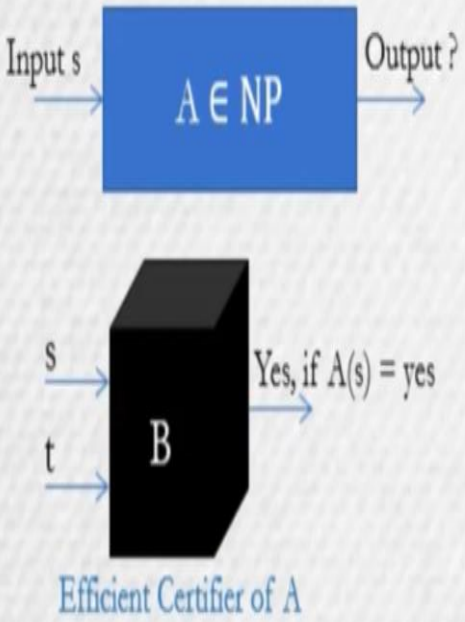
Then that would mean CSAT is at least as hard as A, i.e. $A \leq_P$ CSAT
But as A is in NP CSAT is then NP Hard.

Algorithms and Circuits



- Any algorithm that takes a fixed number of bits n as input and produces a yes/no answer can be represented by such a circuit. Moreover, if algorithm takes polynomial-time, then circuit is of polynomial-size.
- Notes:
  - A computer program is a series of instructions
  - A special memory location called the program counter keeps track of which instruction is to be executed next
  - At any point of time during the execution the entire state of computation is represented in computer's memory, let us call any such state a configuration C_i
  - The execution of a program can be viewed as mapping M from one configuration to another
  - The computer hardware that accomplishes this mapping is a Boolean combinational circuit

To prove CSAT is NP Hard

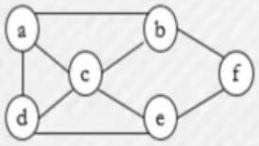


- Let A be in NP, then it must have an efficient certifier B(s, t), which takes an input s of A (i.e. $s \in A$) and a certificate t, and verifies in polynomial time that s is a yes instance of A.
- As, B is a polynomial time algorithm, we can build a polynomial size circuit K for B, with $|s + t|$ sources, such that the first $|s|$ sources of K are hardcoded with the bits of s and the remaining $|t|$ inputs are labeled with variables representing the bits of t.

To prove CSAT is NP Hard

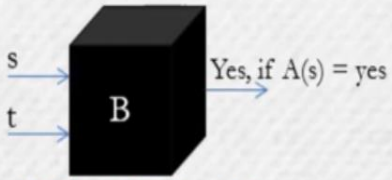
- For example let A be the following problem:

Does this graph contain a Hamiltonian Cycle?



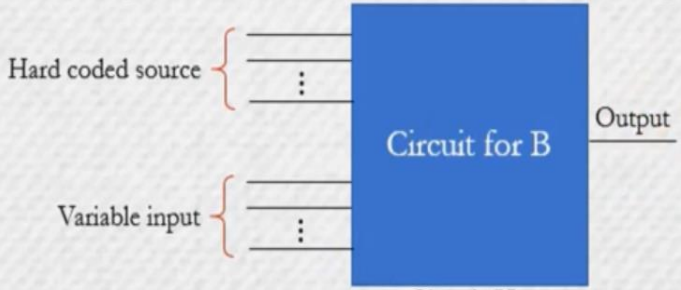
Graph G

- Since A is a problem in NP it must have an efficient certifier algorithm B, which takes an input s of A and a certificate t, and verifies if the certificate is correct in polynomial time.



Efficient Certifier of A

- As B is a polynomial time algorithm, we can build its circuit K in polynomial time.

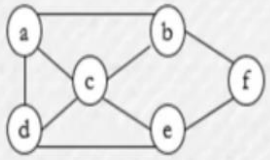


Circuit K

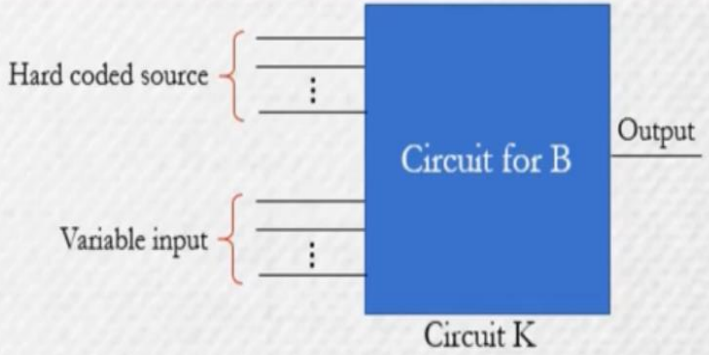
To prove CSAT is NP Hard

- Now let's try to solve A using the circuit K

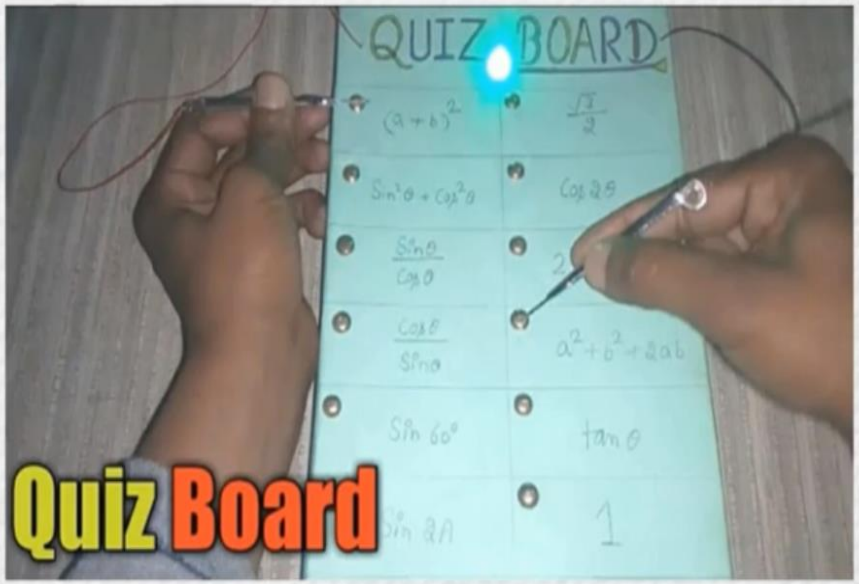
Does this graph contain a Hamiltonian Cycle?



Graph G



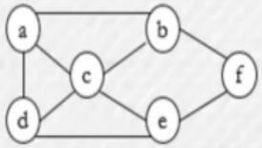
- We will put the input s of A (which is the binary encoding of the graph G) in the hard coded sources of K and try all possible permutations of its vertices one by one in the variable inputs of K.
- Similar to an electrical matching quiz.



To prove CSAT is NP Hard

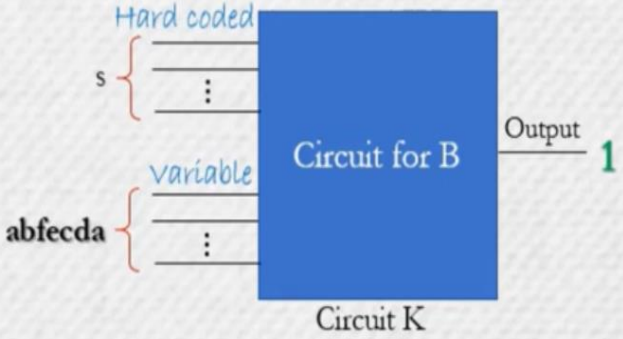
- Now let's try to solve A using the circuit K

Does this graph contain a Hamiltonian Cycle?

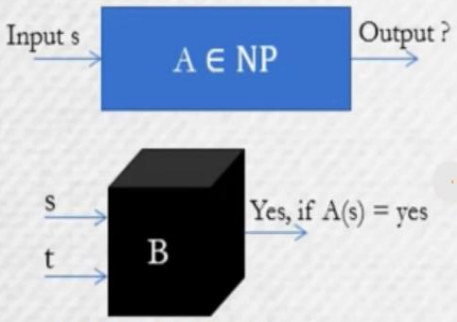


Graph G

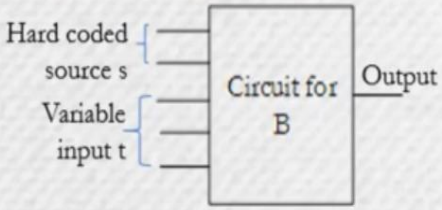
- We will put the input s of A (which is the binary encoding of the graph G) in the hard coded sources of K and try all possible permutations of its vertices one by one in the variable inputs of K.



CSAT is NP Complete problem



Efficient Certifier of A



Hard code source s & try all possible values of t of length ≤ p(|s|). If for some t, the output of the circuit is 1, then t is a solution of A for input s.

Solving A with the circuit for B

- Let A be in NP, then it must have an efficient certifier B(s, t), which takes an input s of A (i.e. $s \in A$) and a certificate t, and verifies in polynomial time that s is a yes instance of A.
- As, B is a polynomial time algorithm, we can build a polynomial size circuit K for B, with $|s + t|$ sources, such that the first $|s|$ sources of K are hardcoded with the bits of s and the remaining $|t|$ inputs are labeled with variables representing the bits of t.
- Now we observe that K's output is 1 if and only if t is a solution of A for input s.
- Thus, $A \leq_p$ CSAT and Hence CSAT is NP Hard.
- But we have also proved that CSAT $\in$ NP, Hence CSAT is NP Complete.

3-CNF Satisfiability (3SAT) Problem

- Let us suppose we are given a set of literals (Boolean variables) $X = \{x_i\}$, each of which can take a value of 0 or 1.
- A clause C is simply a disjunction (OR) of distinct literals or their negations, e.g. $C = (x_1 \vee x_2 \vee \overline{x_3} \vee x_4 \vee \overline{x_5})$
- A Boolean formula is in conjunctive normal form (CNF), if it is expressed as a conjunction (AND) of clauses, e.g. $C_1 \wedge C_2 \wedge C_3 \wedge C_4$
- A Boolean formula is in 3-CNF if each clause has exactly three distinct literals, e.g. $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3})$
- An assignment of Boolean values (0/1) to the literals is said to satisfy a 3-CNF formula if it causes the conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ to evaluate to 1, where k is the number of clauses in the formula.
- The 3SAT problem is then stated as:

  Given a conjunction of a set of k clauses $C_1, C_2, \ldots, C_k$ *each of length 3, over a set of literals* $X = \{x_i\}$*, does there exist a satisfying assignment of the literals?*

3SAT belongs to NP

- A certificate for 3SAT consisting of a satisfying assignment for an input formula can be verified in polynomial time.
- The verification algorithm simply replaces each variable in the formula with its corresponding value and evaluates the expression.
- For example, we can easily verify that an assignment of 101 to x_1, x_2, x_3 satisfies the 3-CNF formula $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3})$
- As we can see, this task is doable in O(k) time, k being the number of clauses.
- Hence 3-SAT is in NP.

3SAT is NP Complete

We will prove that 3SAT is NP Complete by proving that CSAT $\leq_p$ 3SAT

Let us take an arbitrary circuit k, as shown in the picture. We will reduce k (which is an instance of CSAT) to an instance of 3SAT.

We start by associating a variable x_j with each node j in k.

First, we need to encode the requirement that the circuit computes values correctly at each gate from the input values.

There will be three cases depending on the three types of gates:

$x_2 = \neg\, x_3 \rightarrow$ we add 2 clauses $(x_2 \vee x_3), (\overline{x_2} \vee \overline{x_3})$

$x_1 = x_4 \vee x_5 \rightarrow$ we add 3 clauses $(x_1 \vee \overline{x_4}), (x_1 \vee \overline{x_5}), (\overline{x_1} \vee x_4 \vee x_5)$

$x_0 = x_1 \wedge x_2 \rightarrow$ we add 3 clauses $(\overline{x_0} \vee x_1), (\overline{x_0} \vee x_2), (x_0 \vee \overline{x_1} \vee \overline{x_2})$

Next, we add clauses corresponding to hard-coded sources and output

e.g. $x_5 = 0 \rightarrow$ we add the clause $\overline{x_5}$

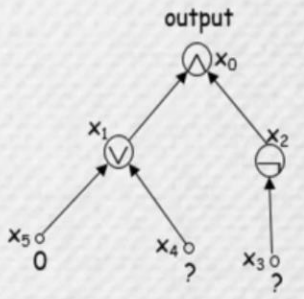
For the output node, we add the clause x_o which requires the output to be 1.

Finally, we turn clauses of length < 3 to clauses of length 3 using following rules:

If $C_i = (c_1 \vee c_2) \rightarrow$ we add 2 clauses $(c_1 \vee c_2 \vee q), (c_1 \vee c_2 \vee \bar{q})$

If C_i has one literal c $\rightarrow$ we add 4 clauses: $(c \vee q_1 \vee q_2), (c \vee q_1 \vee \overline{q_2}), (c \vee \overline{q_1} \vee q_2), (c \vee \overline{q_1} \vee \overline{q_2})$

Thus we have reduced an arbitrary instance of CSAT to 3SAT, hence CSAT $\leq_p$ 3SAT

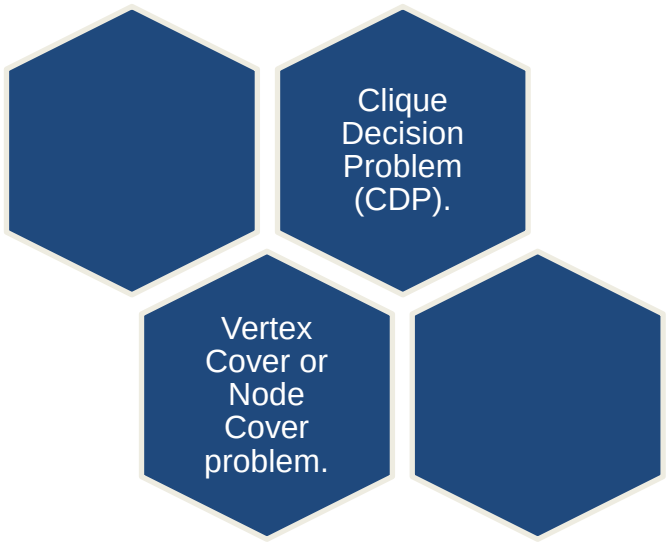


NP Hard Graph Problems

To prove that a problem is NP Complete, we must:

- Prove that it belongs to NP Class.
- Prove that it is a NP Hard problem.
- The strategy we adopt to show that a problem L2 is NP Hard is:
- Pick a problem L1 already known to be NP Hard
- Show how to obtain (in polynomial deterministic time) an instance I' of L2 from any instance I of L1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I of L1.
- Conclude from step 2 that L1αL2.
- Conclude from steps 1 and 3, and the transitivity of α that L2 is NP Hard.

NP Hard Graph Problems



Clique

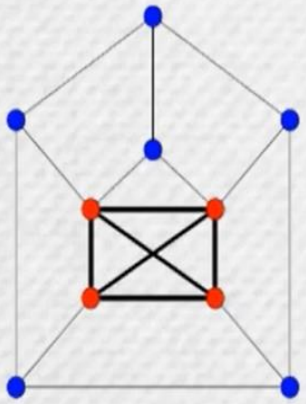
- An instance of a clique problem gives you 2 things as input
  - Graph
  - Some positive integer k
- Question being asked = do we have a clique of size k in this graph
- Why can't I just go through and pick all possible k-subsets?

Reducing 3CNF SAT to CLIQUE

- Given – A boolean formula in 3 CNF SAT
- Goal – Produce a graph (in polynomial time) such that

  Satisfiabiality $\Leftrightarrow$ Clique of a certain size
- We will construct a graph where satisfying formula with k clauses is equivalent to finding a k vertex clique

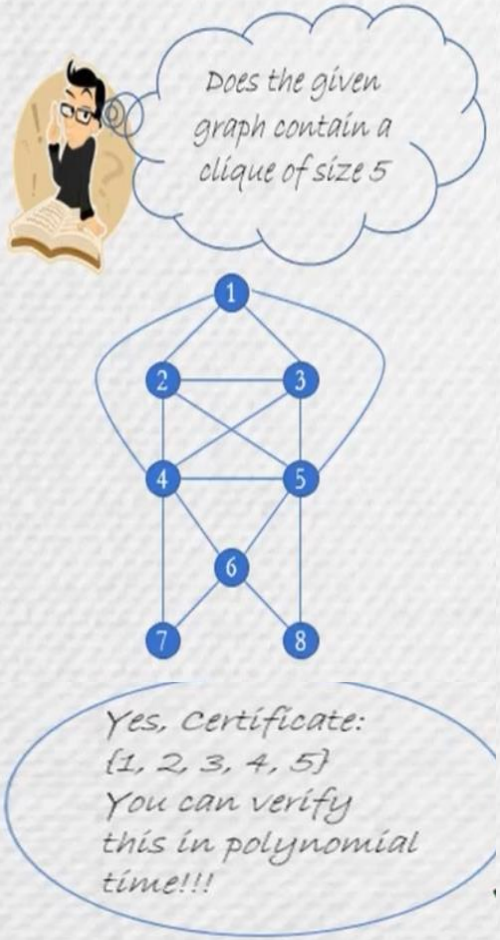
Click Decision Problem (CDP)



A graph with a clique of size 4.

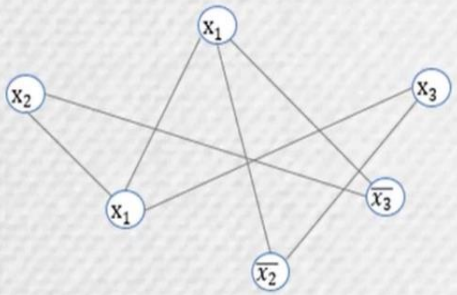
- Let G(V, E) be any given graph.
- A clique is a sub-graph $G_1(V_1, E_1)$ of G such that every pair of vertices in V_1 are adjacent (a complete sub-graph).
- **Optimization problem:** Find a clique of maximum size in G.
- **Decision problem:** Does G contain a clique of size k?

CDP belongs to NP



- To show that CDP ∈ NP for a given graph G(V, E) we use the set $V_1 \subseteq V$ of vertices as a certificate. In other words, it is claimed that the set of vertices V_1 forms a clique in G.
- Checking whether V_1 is a clique can be accomplished in polynomial time by checking whether for every pair u, v ∈ V_1 , the edge (u, v) belongs to E
- This verification algorithm runs in $O(n^2)$ time, where n is the number of vertices in V_1.
- Hence CDP is in NP

CDP is NP Complete



Example:
$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})$

Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a Boolean formula in 3-CNF with k clauses

The clause C_r has exactly 3 literals l_{r1}, l_{r2} & l_{r3}

We will construct a graph G such that F is satisfiable if and only if G has a clique of size k using the following rules:

1. For each clause $C_r = (l_{r1} \vee l_{r2} \vee l_{r3})$ add 3 vertices v_{r1}, v_{r2} & v_{r3}
2. Add an edge between v_{ri} & v_{sj} if
   a. The literals l_{ri} and l_{sj} are in different clauses
   b. l_{ri} is not a negation of l_{sj}

Let us now prove that formation of F into G is a reduction.

If F has a satisfying assignment then each clause C_r contains at least one literal l_{ri} that is assigned to 1, and such a literal corresponds to vertex v_{ri} in G.

Picking one such literal from each clause yields a set V_1 of k vertices.

For any two vertices v_{ri} and v_{sj} in V_1, where r is not equal to s (as they belong to different clauses), both correspond to l_{ri} and l_{sj} mapped to 1, so they can't be complements.

Hence edge (v_{ri}, v_{sj}) belongs to the edge set E of G(V, E)

Thus G has a complete sub-graph of k vertices, which belong to V_1.

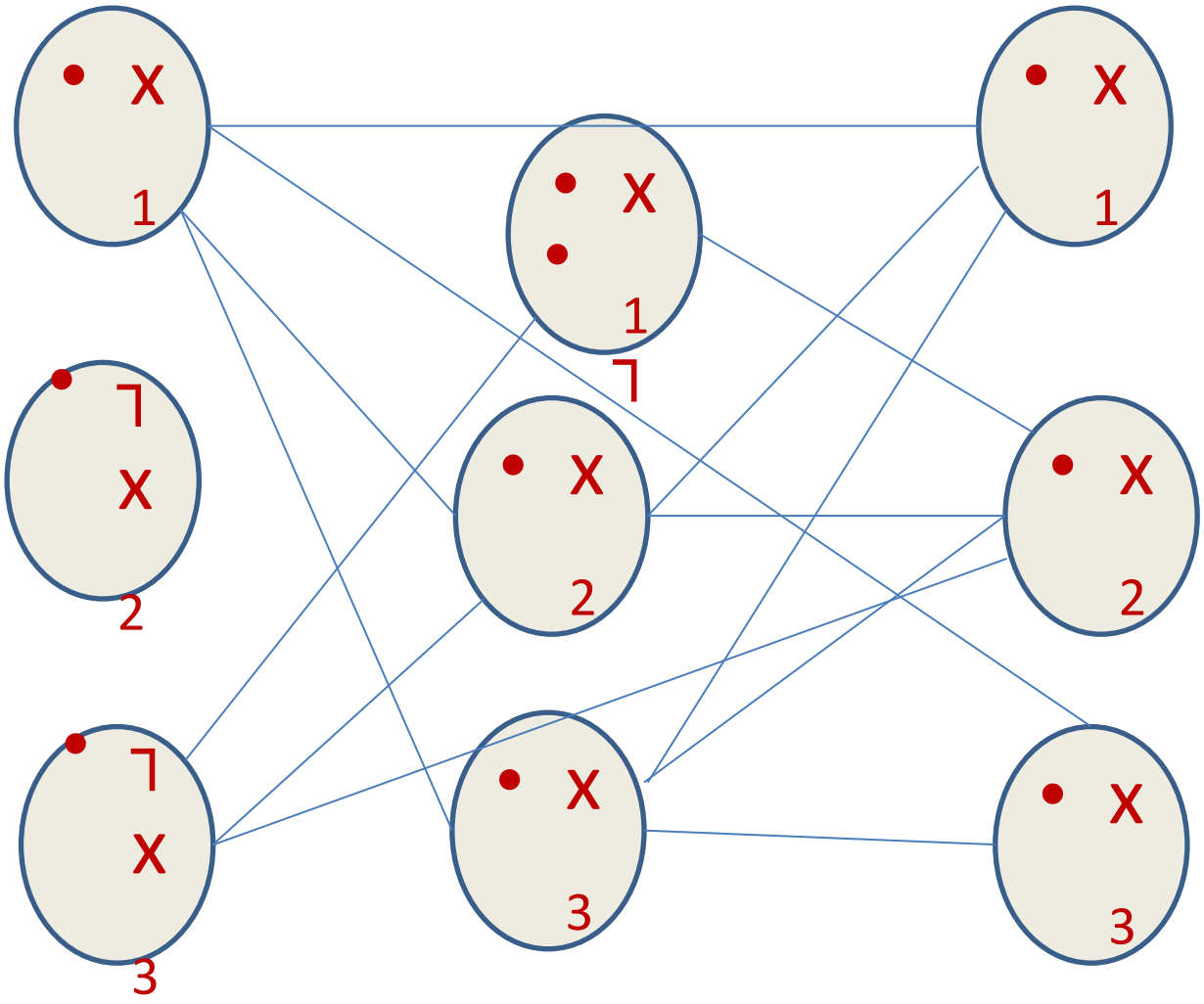
We conclude from the above that $3SAT \leq_P CDP$

Hence CDP is NP Complete

REDUCE 3-CNF SAT to CLIQUE

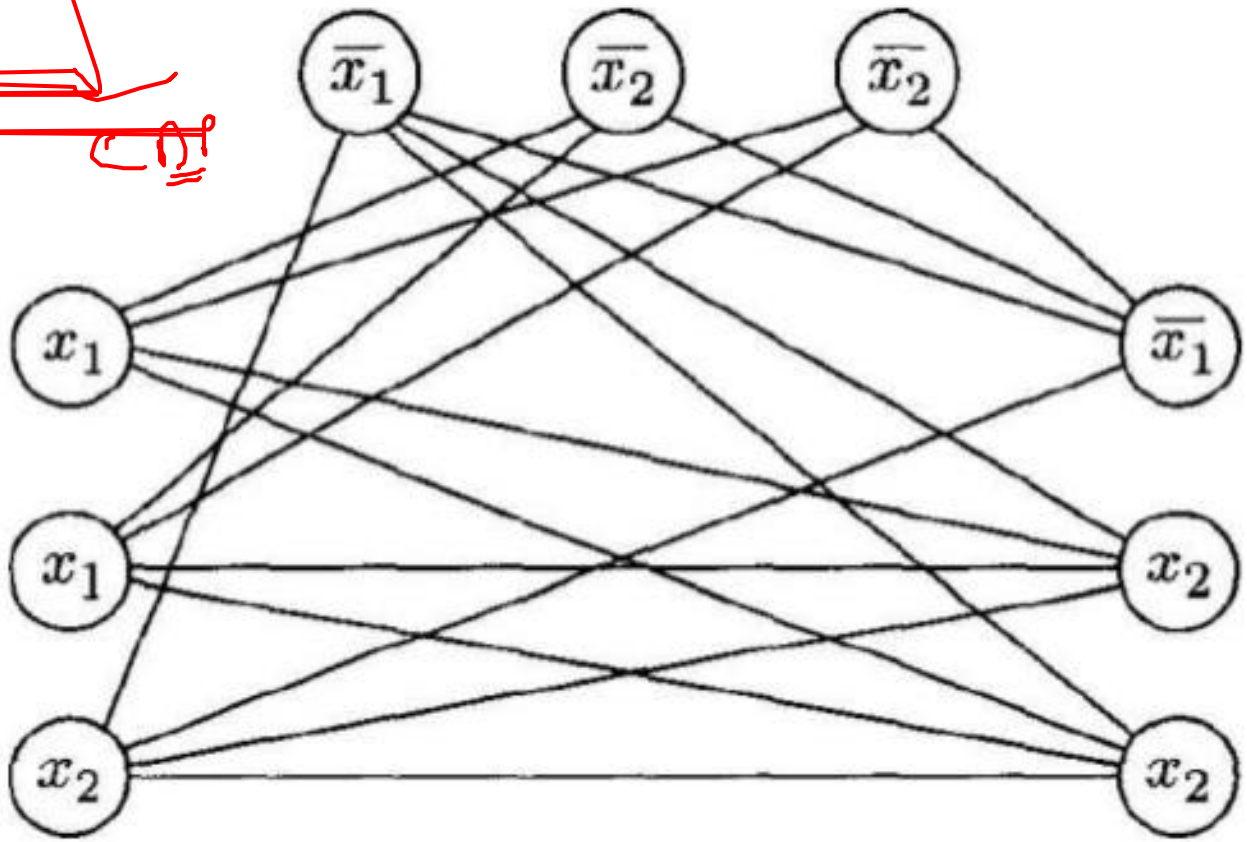
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

- For each clause, create a vertex for each literal
- For the edges
- Connect vertices if they come from different clauses
- Even if the vertices come from different clauses, do not connect if it results in incompatibility. No variable should be connected to its not.



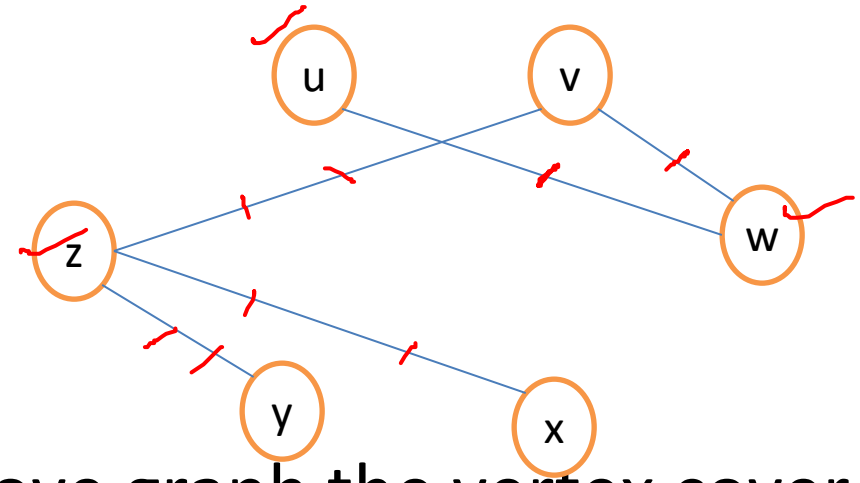
Still there are some edges to complete the graph.

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



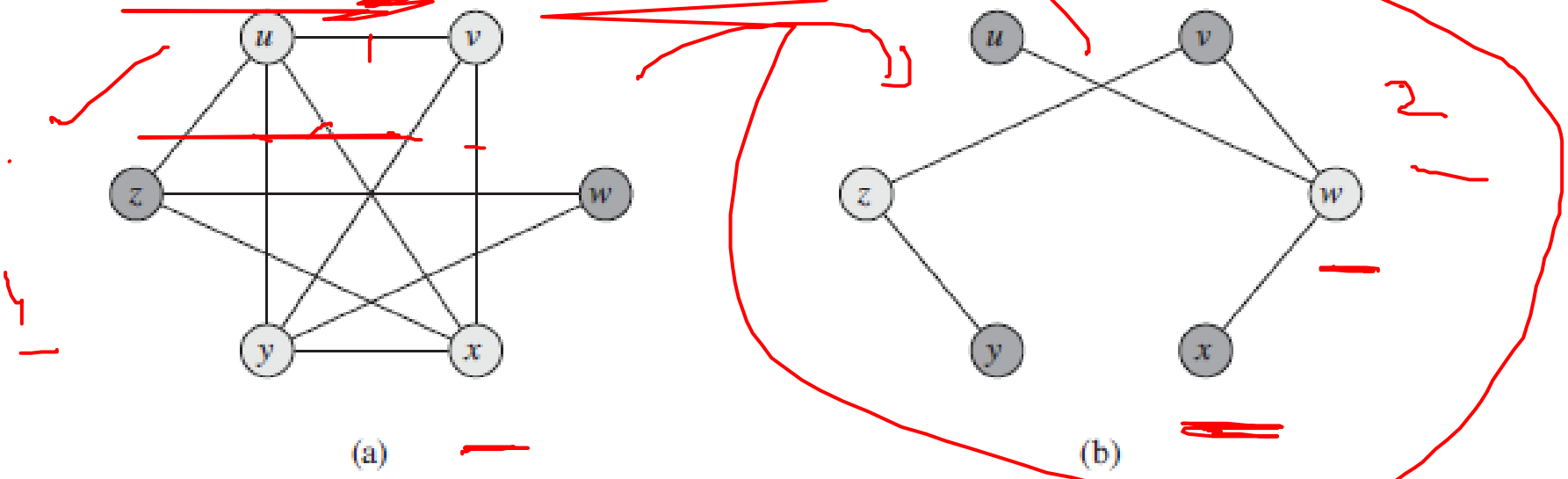
Vertex Cover or Node Cover problem

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



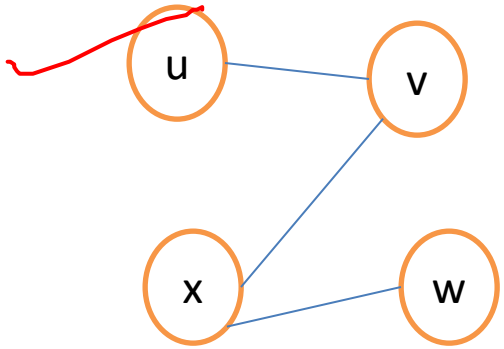
- For the above graph the vertex cover is {w,z} whose size=2.
- Because between w and z, they cover all the edges of the graph

Vertex Cover or Node Cover Problem



- Clique of size k in G exists iff a vertex cover of size |V| - k exists in G' where G' is the complement graph (vertices that had an edge between then in G do not have one in G' and vice versa)
- The original graph has a u,v,x,y CLIQUE. That is a clique of size 4
  The complement graph has a vertex cover of size 6 (number of vertices) – 4 (clique size). z,w is one such vertex cover.

- Given a graph G’



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

NP Hard Code Generation Problems

- A function of a compiler is to translate programs written in some source language into an equivalent assembly language or machine language.
- We look at the problem of translating arithmetic expressions in a language such as C++ into assembly language code.
- We assume a very simple machine A, which has only one register called the accumulator. All arithmetic has to be performed in this register.
- If ʘ represents a binary operator such as +,_,*,/ then the left operand of ʘ must be in accumulator.

- The relevant assembly language instructions are:

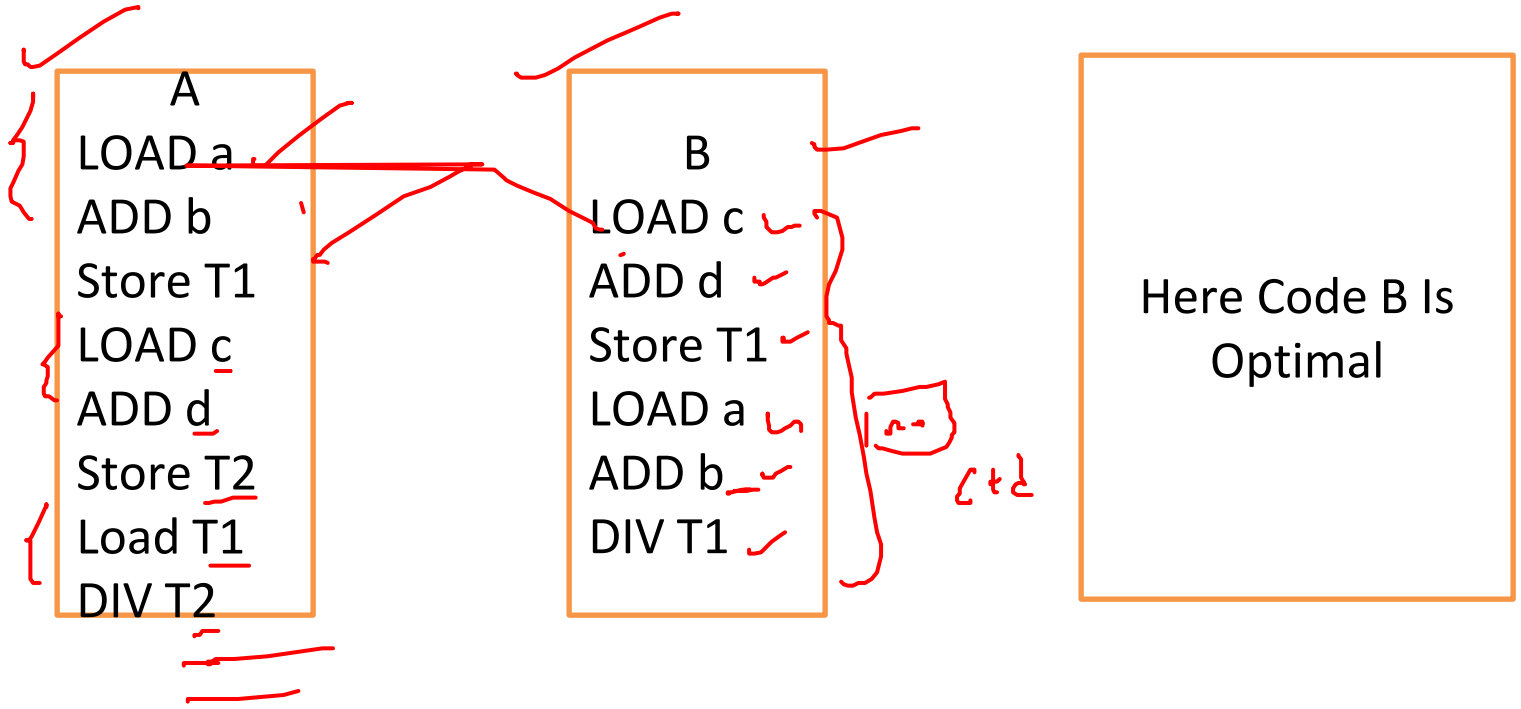
  LOAD X // Load accumulator with contents of memory location X

  STORE X //Store contents of accumulator into memory location X

  OP X // OP may be ADD,SUB,MPY or DIV

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

- As an example consider the arithmetic expression: (a+b)/(c+d)
- Let T1,T2 be temporary storage areas in memory. Two possible assembly language versions of this expression are



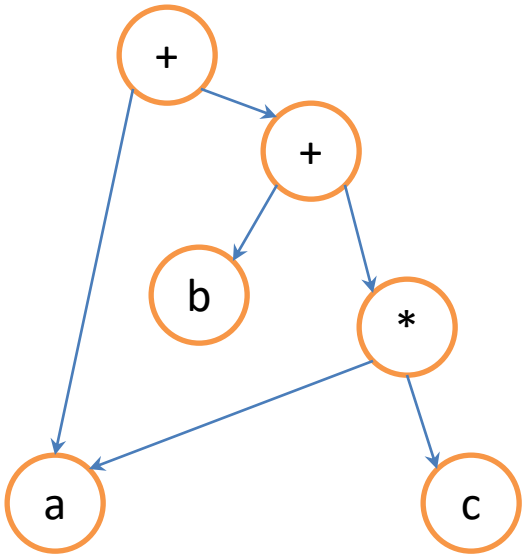
Code generations with common subexpressions

- When arithmetic expressions have common subexpressions, they can be represented by a directed acyclic graph (DAG).
- Every internal node (node with non-zero out degree) in the dag represents an operator.
- Assuming the expression contains only binary operators, each internal node P has out-degree two.
- The two nodes adjacent from P are called the left and right children of P respectively. The children of P are the roots of the dags for the left and right operands of P.

Definitions:

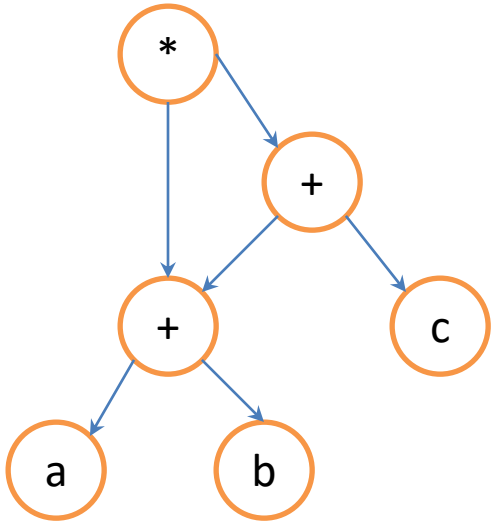
- A leaf is a node with out-degree zero.
- A level-one node is a node both of whose children are leaves.
- A shared node is a node with more than one parent.
- A leaf dag is a dag in which all shared nodes are leaves.
- A level-one dag is a dag in which all shared nodes are level-one nodes.

- a+(b+a*c)



Leaf dag

(a+b)*(a+b+c)



level-one dag

- The problem of generating optimal code for level-one dags is NP-Hard even when the machine for which code is being generated has only one register.
- Determining the minimum number of registers needed to evaluate a dag with no STOREs is also NP-hard.
- To prove the above statements we use the feedback node set (FNS) problem that is already proved to be NP-Hard.

- Feedback node set problem (FNS):

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

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

NP-Hard Scheduling Problem

- To prove the problems of scheduling as NP-hard we have to use the NP-hard problem called partition.
- This problem requires us to decide whether a given multi-set A = $\{a_1,a_2,a_3,\ldots..,a_n\}$ of n positive integers has a partition P such that

  $\sum_{i\varepsilon P} a_i = \sum_{i\notin P} a_i$.
- Partition problem can be shown as NP-hard by first showing the sum of subsets problem to be NP-hard.
- Sum of subsets problem is a problem which determines whether A = $\{a_1,a_2,\ldots..,a_n\}$ has a subset S that sums to a given integer M.

Job Shop Scheduling

- A job shop has m different processors. The n jobs to be scheduled require the completion of several tasks. The time of the jth task for job J_i is $t_{k,i,j}$. Task j is be performed on processor P_k. The tasks for any job J_i are to be carried out in the order 1,2,3,…. and so on. Task j cannot begin until task j-1 (if j>1) has been completed.
- It is quite possible for a job to have many tasks that are to be performed on the same processor
- In a non preemptive schedule, a task once begun is processed without interruption until it is completed.
- The proof for non preemptive schedule is very simple.
- Partition α minimum finish time preemptive job shop schedule (m>1).

Overview of NP Complete problems