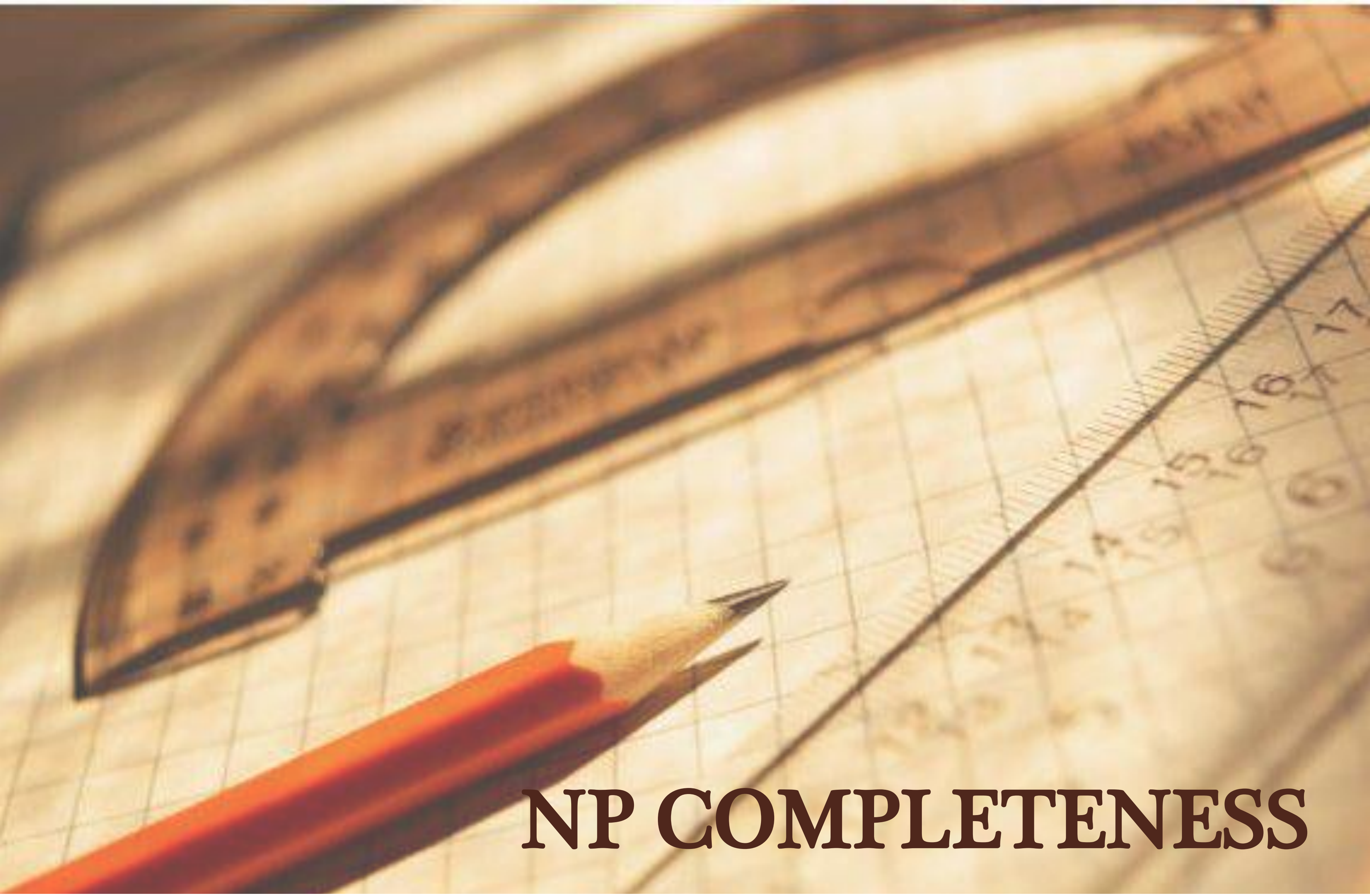
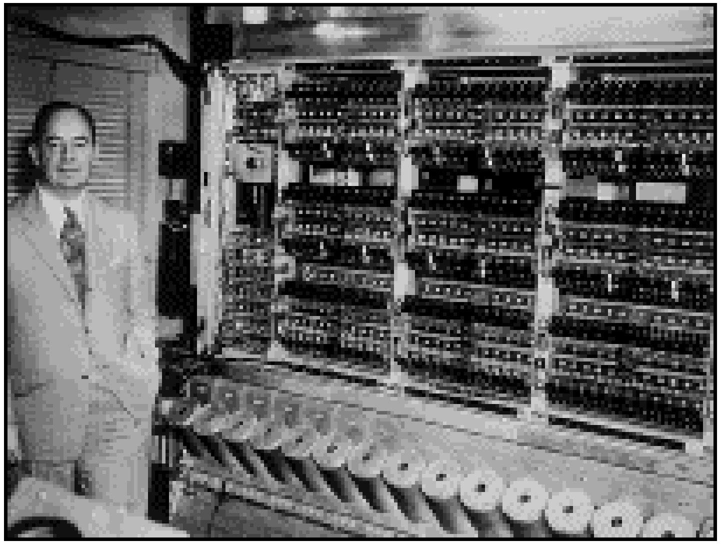


NP COMPLETENESS

By: Tamal Chakraborty

Background

- Before 1950s:
  - computers will solve anything
- 1950s & 1960s: The wall
  - Computers can't solve basic problems.
- Today:
  - The wall still stands



John von Neumann, 1950

The Classes P and NP

- 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 constant and n is the input size.
- The class NP consists of problems which are verifiable in polynomial time. i.e. If we are given a certificate of a solution, we can verify that the certificate is true in polynomial time.
- A problem in P is also in NP.
- A problem is NP Complete if it is in NP and as hard as any problem in NP.

How to show that a problem is NP complete

- We are not trying to prove the existence of an efficient algorithm, but rather that no efficient algorithm is likely to exist

Decision Problems

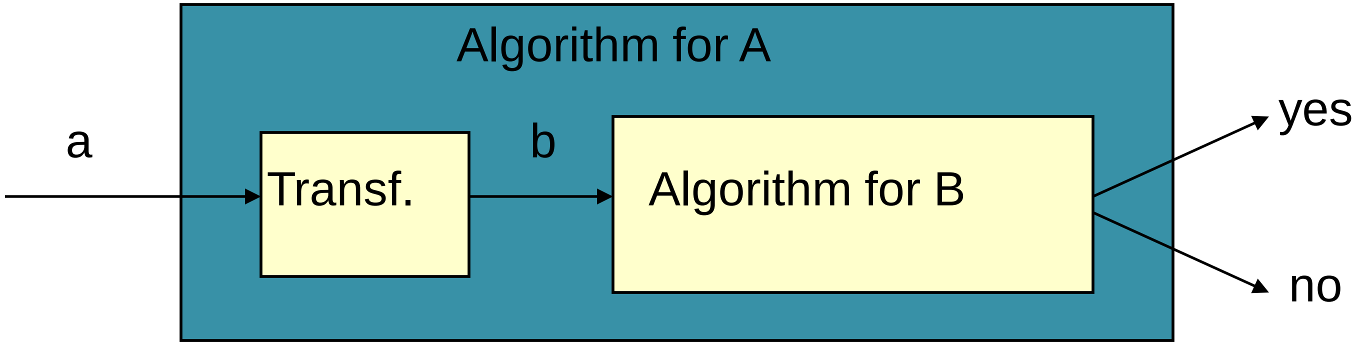
- Computer Problems for which answer is either "yes" or "no"
- Example:
  1. Given a string X and a string Y, does X appear as a sub-string of Y?
  2. Given two sets S & T, do S and T contain the same set of elements?
  3. Given a graph G with integer weights on its edges and an integer k, does G have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

Reductions

- Let us consider a decision problem A which we want to solve in polynomial time
- Now say, there is a different decision problem B which we know can be solved in polynomial time
- Let there be a procedure that transforms any instance a of A into some instance b of B with the following characteristics:
    1. The transformation takes polynomial time
    2. The answers are the same, that is if the answer for b is yes, the answer for a is also yes
- We call such a procedure a polynomial time reduction algorithm

Reductions

- The polynomial time reduction algorithm provides a way for solving A in polynomial time as given below:
  1. Given an instance a of A transform to an instance b of B in polynomial time
  2. Run the polynomial time decision algorithm B on instance b
  3. Use the answer for b as the answer for a



Reductions

- Suppose that there is an algorithm A, for which no polynomial time algorithm exist
- Suppose further that we can have polynomial time reduction from an instance of A to an instance of B
- Then no polynomial time algorithm can exist for B

Abstract decision problems and encodings

- An abstract decision problem can be viewed as a function that maps the instance set I to the solution set {0, 1}
- If a computer program is to solve an abstract decision problem, problem instances must be represented in such a way that the program understands
- An encoding of a set S of abstract objects is a mapping e from S to the set of binary strings
- We call a problem whose instance set is a set of binary strings as a concrete problem
- So P is the set of concrete decision problems that are solvable in polynomial time

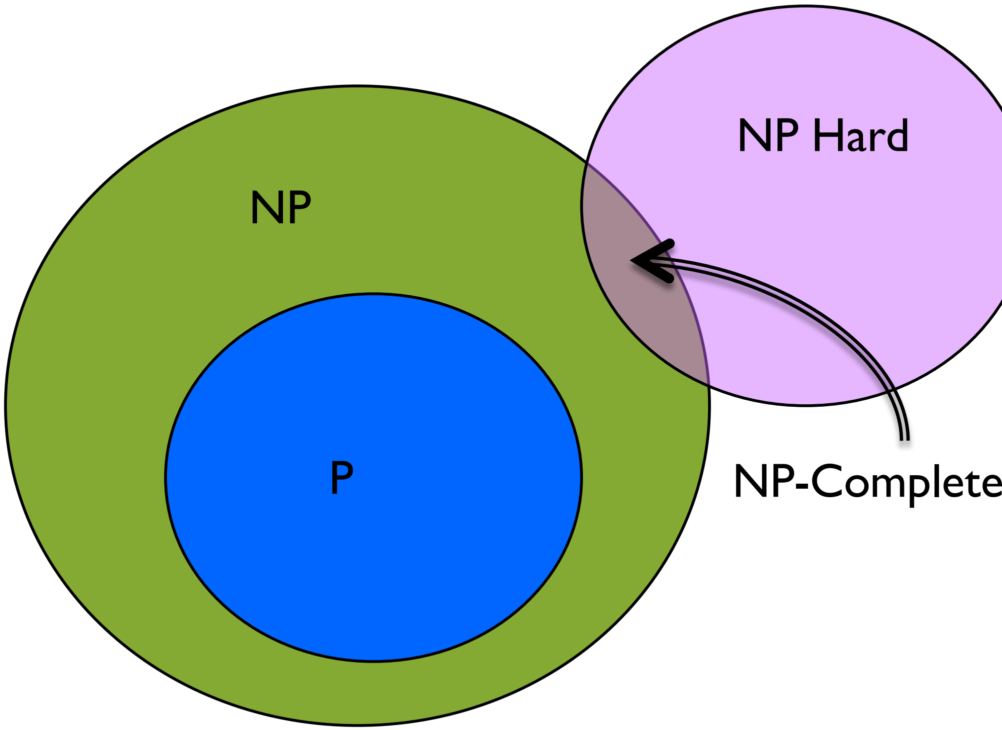
A formal language framework

- An alphabet $\sum$ is a finite set of symbols
- A language L over $\sum$ is any set of strings made up of symbols from $\sum$
- For example if $\sum = \{0, 1\}$ then the set $L = \{10, 01, 100, 010, \ldots\}$ can be a language
- We denote empty string by ε and empty language by ϕ
- The language of all strings over $\sum$ is denoted by $\sum^*$
- We define complement of L by $\sum^* - L$
- Let U be the set of all possible inputs for a decision problem
- Let $L \subseteq U$ be the set of all inputs for which the answer to the problem is yes.
- We calls L the language corresponding to the problem.

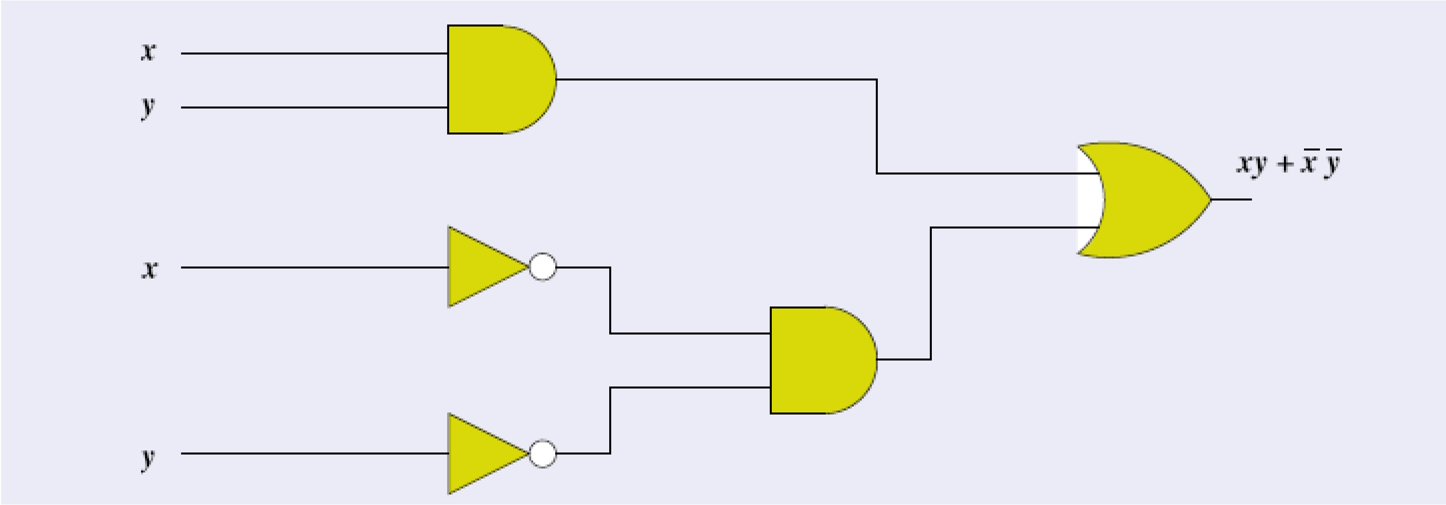
NP Completeness and reducibility

- Let L1 and L2 be two languages from input spaces U1 and U2. L1 is **polynomially reducible** to L2 if there exists a polynomial time algorithm that coverts each u1 ε U1 to another input u2 ε U2, such that u1 ε L1 if and only if u2 ε L2.
- Polynomial time reductions provide a formal means for showing that one problem is at least as hard as another.
- A language $L \subseteq \{0, 1\}*$ is **NP Complete** if
  1. L ε NP and
  2. L1 is polynomial time reducible to L, for all L1 ε NP
- If a language L satisfies property 2, but not necessarily property 1, we say L is **NP Hard**.

NP Completeness



Circuit Satisfiability Problem



Is there a set of input values, such that output is 1?

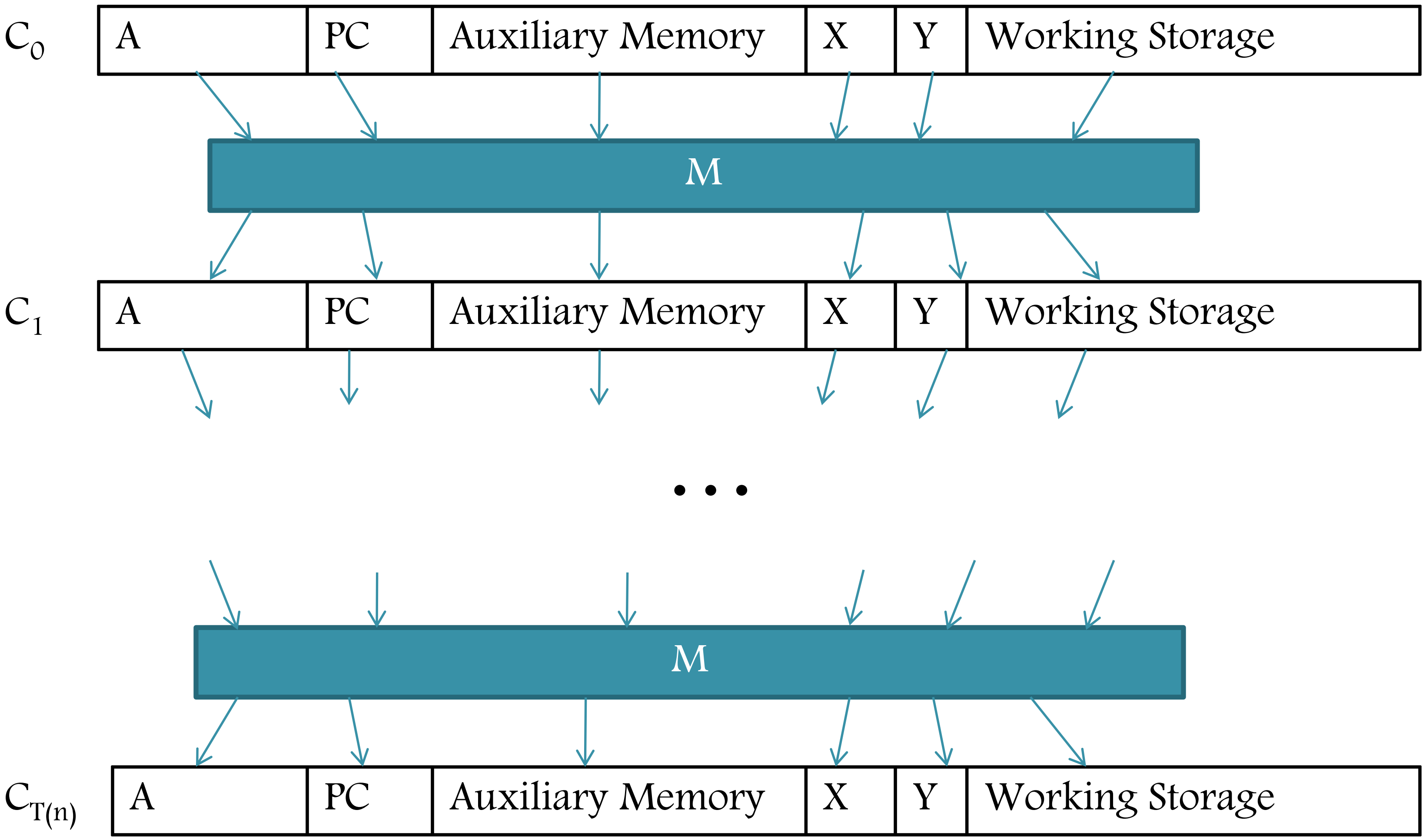
Circuit Satisfiability Problem

- Given a set of inputs it is easy to check whether the output is 1.
- The algorithm runs in linear time in the number of logic Gates in the circuit.
- Hence CIRCUIT-SAT is in NP.
- To prove that CIRCUIT-SAT is NP Complete we must also prove that for all A in NP, A can be reduced to CIRCUIT-SAT in polynomial time.

Circuit Satisfiability Problem

- Let us start with the understanding that **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
  - The execution of a program can be viewed as mapping one configuration to another
  - The computer hardware that accomplishes this mapping is a boolean combinational circuit

Circuit Satisfiability Problem



Circuit Satisfiability Problem

- To prove that CIRCUIT-SAT is NP Hard we have to show that: **$\forall A \in NP, A \leq_p$ CIRCUIT SAT**

Since $A \in NP$, there is an algorithm $C(s, t)$ such that:

- C checks, given an instance s and a certificate t, whether or not t is a solution of s.
- C runs in polynomial time.

In polynomial time, build a circuit D with input size $|s + t|$ such that:

- First $|s|$ bits of the input are hardcoded with s.
- Remaining bits of input represent the bits of t.
- C's answer is given at the output gate of D.
- Size of D is polynomial in the number of inputs.
- D's output is true if and only if t is a solution of s.

3-SAT

- A literal in a boolean formula is an occurrence of a variable or its negation.
- A boolean formula is in conjunctive normal form (CNF) if it is expressed as an AND of clauses, each of which is OR of one or more literals.
- A boolean formula is in 3-CNF if each clause has exactly three distinct literals.
- For example: $(x1 \vee \neg x2 \vee x3) \wedge (x1 \vee x2 \vee \neg x3)$
- In 3-SAT we are asked whether a given boolean formula in 3-CNF is satisfiable.

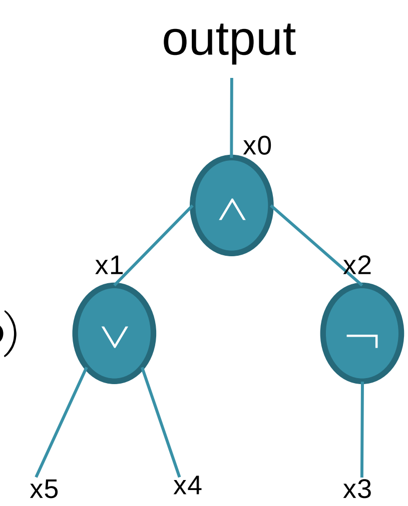
3-SAT is NP Complete

- A certificate for 3-SAT 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.
- This task is easily doable in polynomial time.
- Hence 3-SAT is in NP.
- To prove that 3-SAT is NP-Hard we will reduce CRCUIT-SAT to 3-SAT in polynomial time.

CIRCUIT SAT $\leq_p$ 3-SAT

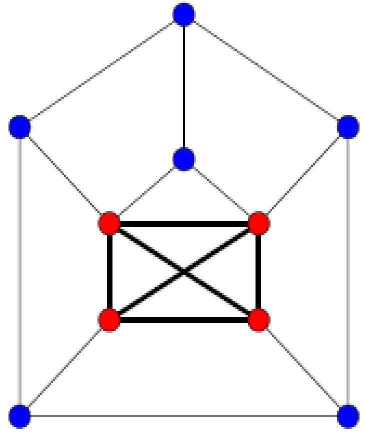
- Let K be any circuit
- Create a 3-SAT variable x_j for a circuit wire j.
- Make circuit compute correct values at each node
  - $x2 = \neg\ x3 \Rightarrow$ add 2 clauses $(x2 \vee x3)$, $(\neg x2 \vee \neg\ x3)$
  - $x1 = x4 \vee x5 \Rightarrow$ add 3 clauses $(x1 \vee \neg x4)$, $(x1 \vee \neg x5)$, $(\neg x1 \vee x4 \vee x5)$
  - $x0 = x1 \wedge x2 \Rightarrow$ add 3 clauses $(\neg x0 \vee x1)$, $(\neg x0 \vee x2)$, $(x0 \vee \neg x1 \vee \neg x2)$
- Add clauses corresponding to hard-coded input values and output
  - For example if input $x5 = 0$, add 1 clause $\neg x5$
  - For output $x0$, add 1 clause $x0$
- Turn clauses of length < 3 to clauses of length 3 using rules:
  - If $C_i = (c1 \vee c2) \Rightarrow$ add 2 clauses $(c1 \vee c2 \vee p)$, $(c1 \vee c2 \vee \neg p)$
  - If C_i has 1 literal c $\Rightarrow$ add 4 clauses:

    $(c \vee p \vee q)$, $(c \vee p \vee \neg q)$, $(c \vee \neg p \vee q)$, $(c \vee \neg p \vee \neg q)$



Clique decision problem

- Let G(V, E) be any given graph.
- A clique is a sub-graph G1(V1, E1) of G such that every pair of vertices in V1 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?

CLIQUE is NP Complete

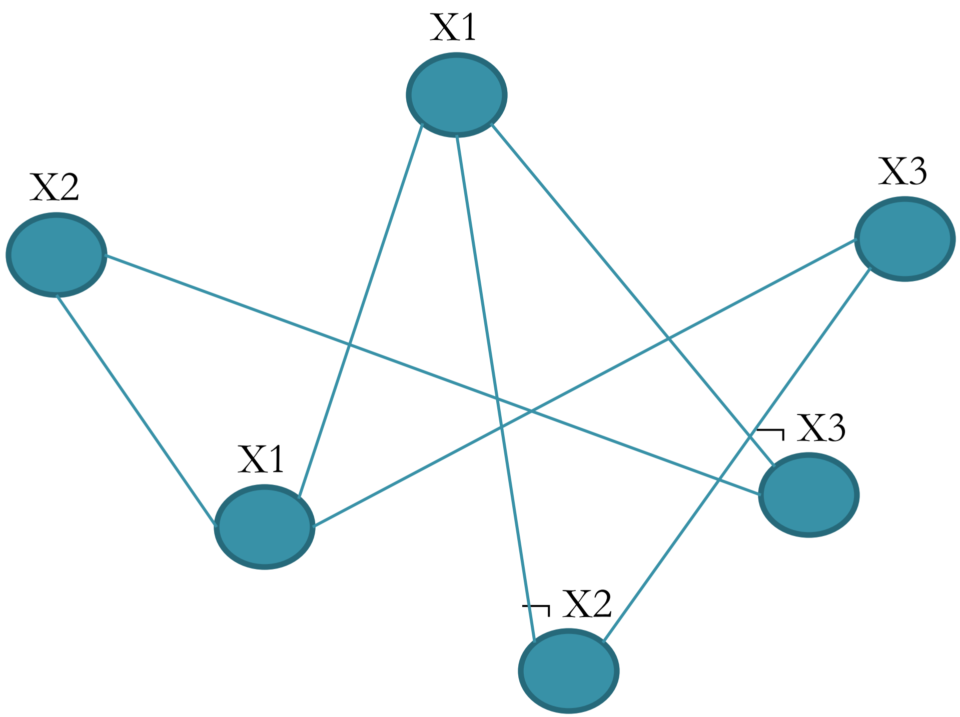
- To show that CLIQUE ε NP for a given graph (V, E) we use the set V1$\subseteq V$ of vertices in the clique as a certificate for G.
- Checking whether V1 is a clique can be accomplished in polynomial time by checking whether for every pair u, v ε V1 , the edge (u, v) belongs to E
- Hence CLIIQUE is in NP
- To prove that CLIQUE is NP Hard we will show that 3-SAT can be reduced to CLIQUE in polynomial time

3-SAT $\leq_p$ CLIQUE

- 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 clique of size k using the following rules:
  - For each clause $C_r = (l_{r1} \vee l_{r2} \vee l_{r2})$ $\underset{r \neq s}{\text{add}}$ 3 vertices v_{r1}, v_{r2} & v_{r3}
  - Add an edge between v_{ri} & v_{sj} if
    1. The vertices are in different triples, i.e.
    2. The vertices are consistent, i.e. l_{ri} is not a negation of l_{sj}

3-SAT $\leq_p$ CLIQUE

$(x1 \vee x2 \vee x3) \wedge (x1 \vee \neg\, x2 \vee \neg\, x3)$



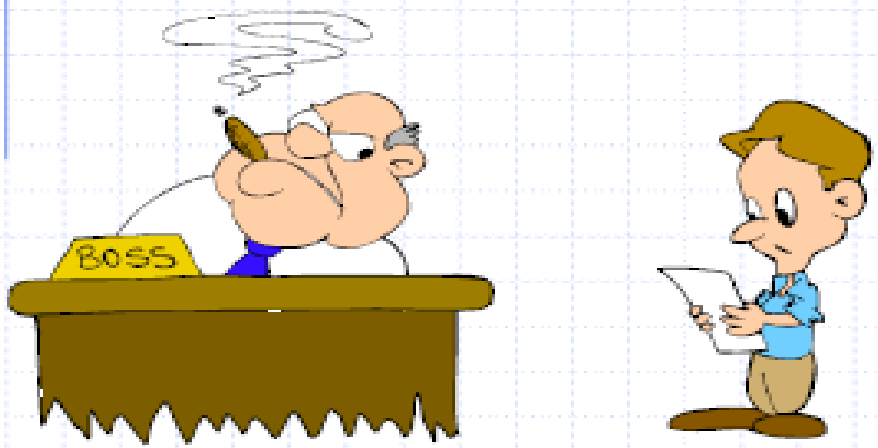
3-SAT $\leq_p$ CLIQUE

- 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 V1 of k vertices.
- For any two vertices v_{ri} and v_{sj} in V1, where r is not equal to s, both correspond to l_{ri} and l_{sj} mapped to 1, and they can't be complements.
- Hence edge (v_{ri}, v_{sj}) belongs to the edge set E of G(V, E)
- Thus V1 is a CLIQUE of size k.

Summary

Dealing with Hard Problems

- What to do when we find a problem that looks hard…

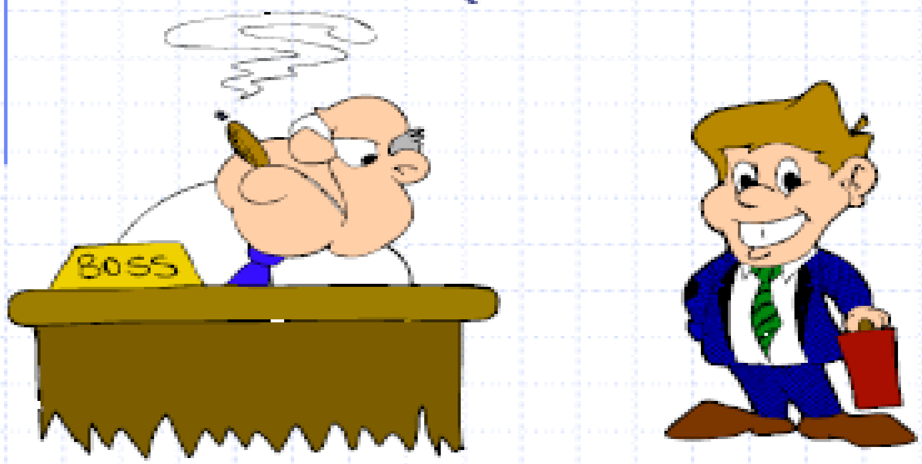


I couldn't find a polynomial-time algorithm;
I guess I'm too dumb.

Summary

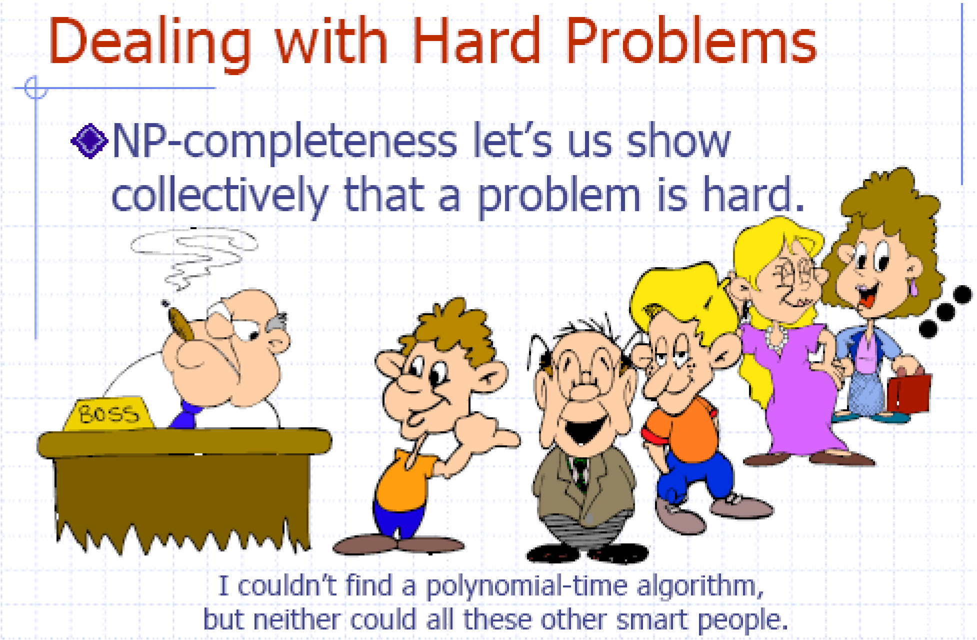
Dealing with Hard Problems

- Sometimes we can prove a strong lower bound… (but not usually)



I couldn't find a polynomial-time algorithm,
because no such algorithm exists!

Summary


Dealing with Hard Problems
NP-completeness let's us show collectively that a problem is hard.
BOSS
I couldn't find a polynomial-time algorithm,
but neither could all these other smart people.

Approximation Algorithms

Need for Approximation

- If a problem is NP Complete we take two approaches towards solving it:
  1. If inputs are small, algorithms with exponential running time are satisfactory.
  2. It may be possible to find near optimal solutions in polynomial time
- An algorithm that returns near optimal solutions is called an **Approximation Algorithm**.

Ratio Bound

- Let there be a problem with input size n.
- Let C be the cost of the solution produced by the approximation algorithm.
- Let C* be the cost of the optimal solution.
- We say an approximation algorithm for the given problem has a **ratio bound** of p(n) if

$$\max(C/C^*, C^*/C) \le p(n)$$

Note: p(n) is never less than 1

Relative Error

- For any input the **relative error** of the approximation algorithm is defined as $|C - C^*|/C^*$
- An approximation algorithm has relative error bound $\varepsilon(n)$ if

$$|C - C^*|/C^* \le \varepsilon(n)$$

Note: relative error is always non-negative

Approximation Scheme

- An **approximation scheme** for an optimization problem is an approximation algorithm that takes as an input an instance of the problem and a const $\varepsilon > 0$
- A polynomial time approximation scheme runs in polynomial time for a fixed $\varepsilon > 0$.
- If ε decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor. i.e. The running time should be a polynomial in $1/\varepsilon$ as well as n.
- An approximation scheme is a fully polynomial time approximation scheme if its running time is polynomial both in $1/\varepsilon$ & n.

0/1 Knapsack Problem

$KnapsackApprox(p, w, m, n, k)$

{

$P_{max} = 0;$

for all combinations I of size $\leq$ k & weight $\leq$ m do

{

$$P_I = \sum_{i \in I} P_i;$$

$$P_{max} = \max(P_{max}, P_I + LBound(I, p, w, m, n));$$

}

return $P_{max};$

}

p[ ] = set of profits
w[ ] = set of weights
m = knapsack capacity
n = number of items
k = non-negative integer

0/1 Knapsack Problem

$LBound(I, p, w, m, n)$

$\{$

$s = 0;$

$t = m - \sum_{i \in I} w_i;$

$\text{for } i = 1 \text{ to } n \text{ do}$

$\text{if } (i \notin I) \text{ and } (w[i] \le t) \text{ then}$

$\{$

$s = s + p[i];$

$t = t - w[i];$

$\}$

$\text{return } s;$

$\}$

0/1 Knapsack Problem

p = {11, 21, 31, 33, 43, 53, 55, 65}	w = {1, 11, 21, 23, 33, 45, 55}	m = 110	n = 8

- Optimal solution P* = 159
- Optimal weight = 109

- If k = 0, then Pmax = Lbound(ϕ, p, w, m, n)
- i.e. Pmax = 139, w = 89
- Elements taken: x = {1, 1, 1, 1, 1, 0, 0, 0}
- Relative error = (P* - Pmax) / P* = 0.126

0/1 Knapsack Problem

K = 1

p = {11, 21, 31, 33, 43, 53, 55, 65}	w = {1, 11, 21, 23, 33, 45, 55}	m = 110	n = 8

I	Pmax	P_I	LBound	P_{MAX}=max{P_{max}, P_I+Lbound}	x
ϕ	0	11	128	139	(1,1,1,1,1,0,0,0)
6	139	53	96	149	(1,1,1,1,0,1,0,0)
7	149	55	96	151	(1,1,1,1,0,0,1,0)
8	151	65	63	151	(1,1,1,1,0,0,1,0)

- If k = 1, then Pmax = 151, w = 101
- Elements taken: x = {1, 1, 1, 1, 0, 0, 1, 0}
- Relative error = (P* - Pmax) / P* = 0.05

0/1 Knapsack Problem

K = 2

$p = \{11, 21, 31, 33, 43, 53, 55, 65\}$	$w = \{1, 11, 21, 23, 33, 45, 55\}$	$m = 110$	$n = 8$

I	Pmax	P_I	LBound	$P_{MAX}=\max\{P_{max}, P_I+Lbound\}$	x
5,6	151	96	63	159	(1,1,1,0,1,1,0,0)

- If $k = 2$, then $Pmax = P^*$, $w = 109$
- Elements taken: $x = \{1, 1, 1, 0, 1, 1, 0, 0\}$
- Total number of subsets tried is given by:
- $\sum_{i=0}^{k} n^i = n^{k+1} - 1/n - 1 = O(n^k)$
- The Lbound function has complexity $O(n)$
- So Total time is $O(n^{k+1})$

THANK YOU!