NP – Hard and NP – Complete Problems

- Decison Problem:
  - A decision problem is a problem that can be presented as a yes-no question of input values
  - E.g. Deciding whether a given natural number is prime
- Definition of P:
  - Set of all decision problems solvable in polynomial time by a deterministic algorithm (Turing machine)
  - Examples:
    - MULTIPLE: Is the integer y a multiple of x?
      - YES: (x, y) = (17, 51).
    - RELPRIME: Are the integers x and y relatively prime?
      - YES: (x, y) = (34, 39).
    - MEDIAN: Given integers x_1 , …, x_n , is the median value < M?
      - YES: (M, x_1 , x_2 , x_3 , x_4 , x_5) = (17, 2, 5, 17, 22, 104)

Def: P is the set of all decision problems solvable in polynomial time on **REAL** computers.

Example: ordered searching O(n)

Evaluation of polynomial O(n)

Sorting O(n log n)

There is another group of problems whose best known algorithms are non polynomial.

Example:

Traveling salesman problem,o((n^2)*(2^n))

Knapsack problem,o(2^(n/2))

Decision Problem

- *Any problem for which the answer is yes or no, is called a decision problem*.

Example: Knapsack problem:

Optimization Problem:

- Find the largest total profit of any subset of objects that fits in the knapsack.
- Find the minimum weight spanning tree of given weighted graph

Decision Problem: Given k, is there a subset of objects that fits in the knapsack and has total profit at least k?

Example : Subset sum problem:

Input is a positive integer C and n objects whose sizes are positive integers $s_1, s_2, \ldots s_n$

Optimization problem: Among subsets of the objects with sum atmost C what is largest subset sum.

Decision Problem: Is there a subset of the objects whose sizes add up to exactly C?

Idea of decision problems

- The decision problem can be solved in polynomial time if and only if the corresponding optimization problem can.

- If the decision problem cannot be solved in polynomial time then the optimization problem cannot either.

Non deterministic algorithms

Algorithms having operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities.

it has following functions :

Choice (S): arbitrarily chooses one of the elements of the set S.

Failure(): signals an unsuccessful case

Success(): signals a successful completion.

Example (Searching for an element x in a given set of elements A[1 : n] $n \geq 1$.)

Non deterministic algorithm:

```
1    J:= choice(1,n);
2    If A[j] = x then {write (j); success;}
3    Write (0); failure();
```

Number 0 can be a output if and only if there is no j such that A[j] = x

(bcoz nond algo terminates unsuccessfully iff there exists no choice which gives successful signal)

Algorithm is of nondeterministic complexity O(1)

However any deterministic search algo on unordered data is of $\Omega(n)$

Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithms terminates successfully/

When ever successful termination is possible , a nondeterministic machine makes a sequence of choices that is a shortest sequence leading to a successful termination.(remember machine is fictitious)

A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to success signal

Example

```
// Sort n positive integers
Algorithm Nsort(A,n)
{
  for i:= 1 to n
     do B[i] :=0;
  for i:= 1 to n do
{ j:= choice (1,n)
If B[j]  0 then failure();
B[j]:= A[i];
}
For i:= 1 to n-1 do
     if B[i]> B[i+1] then failure();
Write(B([1:n]);
Success();
}
```

It is O(n) while all determinisitic sorting algorithms have a complexity $\Omega(n \log n)$

Time of non deterministic algorithms

The time required by a nondeterministic algorithm performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion.

In case successful completion is not possible then the time required is O(1).

A nondeterministic algorithm is of complexity (f(n)) if for all inputs of size n

$n \geq n_0$ that results is a successful completion the time required is

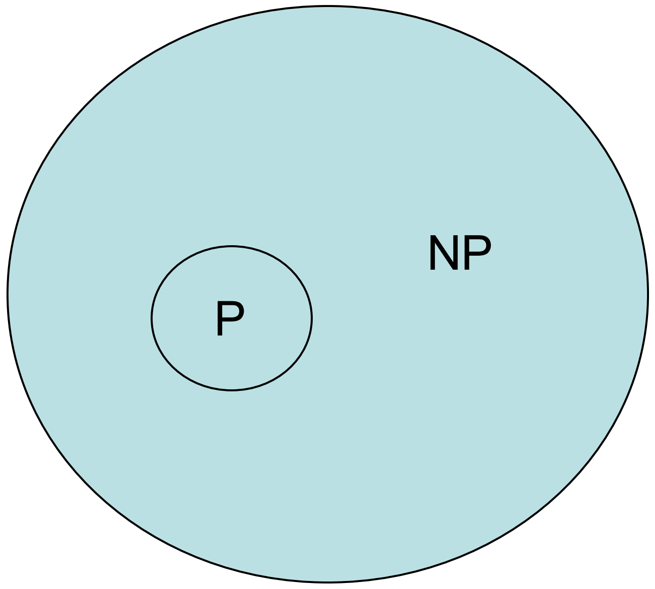
atmost cf(n) for some constant c and n_0

- Definition of NP:
  - Set of all decision problems solvable in polynomial time by a NONDETERMINISTIC Algorithm(Turing machine).
  - Definition is important because it links many fundamental problems
- Useful alternative definition
  - Set of all decision problems with efficient verification algorithms
    - efficient = polynomial number of steps on deterministic TM
  - Verifier: algorithm for decision problem with extra input

The class NP consists of those problems that are verifiable in polynomial time.

If any NP complete problem can be solved in polynomial time ,then every NP complete problem has a polynomial time algorithm.

Relation between P and NP



NP = set of decision problems with efficient verification algorithms

- Why doesn't this imply that all problems in NP can be solved efficiently?
  - BIG PROBLEM: need to know certificate ahead of time
    - real computers can simulate by guessing all possible certificates and verifying
    - naïve simulation takes exponential time unless you get "lucky"

Satisfiability Problem

The satisfiability problem is to determine whether a boolean formula is true for some assignment of truth values to the variables.

{boolean formula is an expression that can be constructed using literals and operators “and” , “or”.

CNF- satisfiabilty is the satisfiability problem for CNF(conjunctive)formula.

Nondeterministic algorithm for satisfiability problem

```
Algoritm Eval (E,n)
{
For i:=1 to n do
   $x_i$ := choice (false,true);
If E($x_1$,$x_2$…$x_n$) then success();
Else failure();
}
```

Time of the nondeterministic algorithm

O(n) time is required to choose the value of $(x_1,x_2\ldots x_n)$

+

Time needed to evaluate E for that assignment

(time is proportional to the length of E)

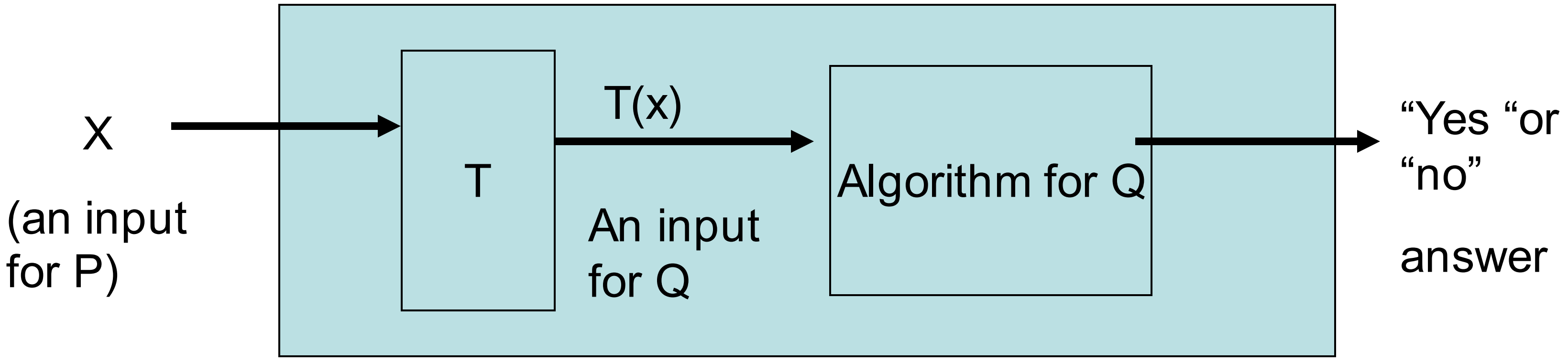
Reducibility

Let L_1 and L_2 be two problems.

Problem L_1 reduces to problem L_2 (written as $L_1 \alpha L_2$)

Iff there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.

Reducibility



Algorithm for P

Reducibility

A problem P_1 can be reduced to P_2 a follows:

Provide a mapping so that any instance of P_1 can be transformed to an instance of P_2.

Solve P_2 then map the answer back to the original.

For P1 to be polynomially reducible to P2 All the work associated with the transformations must be performed in polynomial time.

example

numbers are entered into pocket calculator in decimal. The decimal numbers are concerted in binary,

All calculations are done in binary.

Then the final answer is converted back to decimal for display.

Cook's theorem

Satisfiability is in P if and only if P =NP

So if a polynomial time algorithm can be obtained for satisfiability then

every problem of NP can also be solves by a polynomial time algorithm.

So P= NP

Note

A problem Q is NP- hard does not mean

“ in NP and hard”

It means

“ at least as hard as any problem in NP”

NP hard problem

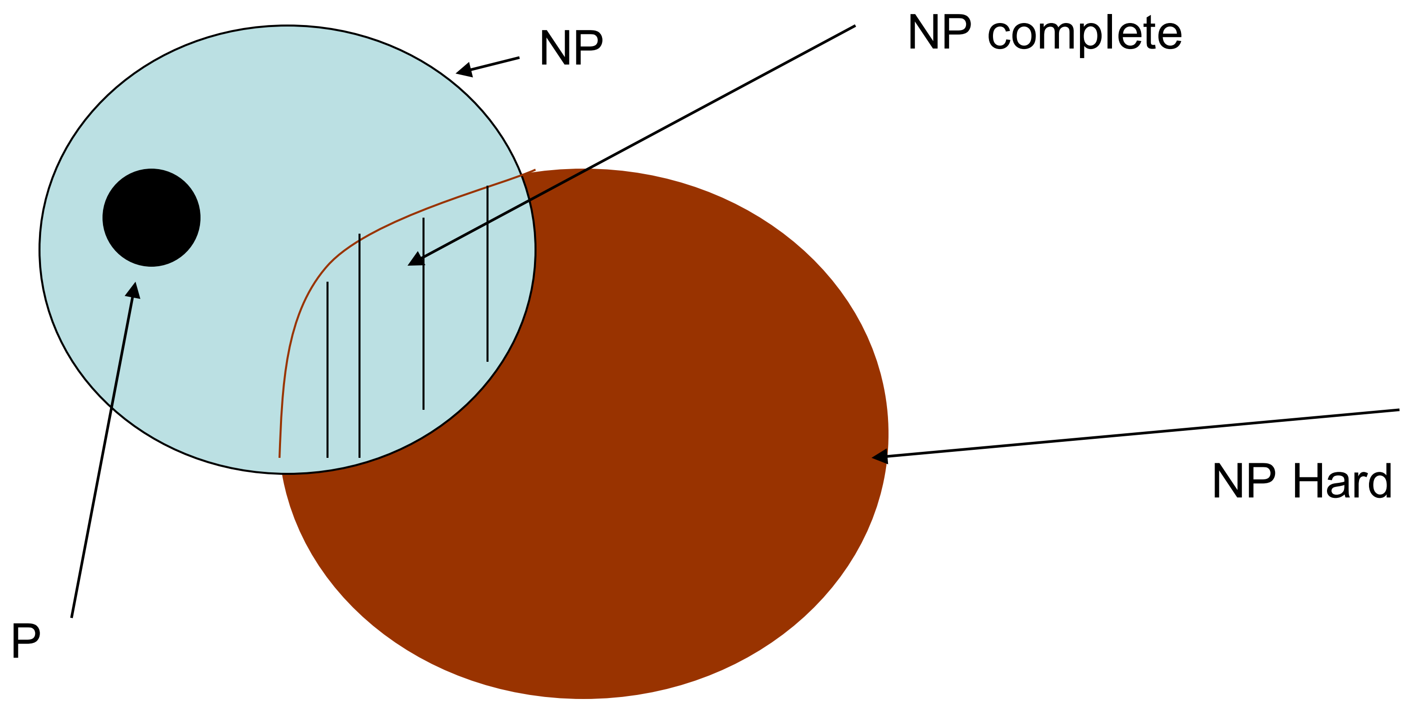
A problem L is NP hard if and only if satisfiability reduces to L.

Hard ness of the NP hard problem can be thought of as

If one can generate polynomial time algorithm for any NP hard problem then by the definition of reducibilty it means that satisfiablity problem can also be solved by a polynomial time algorithm, which we know how important it is(**remember "Cook's theorem")**

NP complete Problem

A problem is NP complete if and only if L is NP hard and L is in NP.



Commonly believed relation ship among P,NP, NP hard and NP complete problems

$P \neq NP$ question has been one of the deepest , most perplexing open research problems in theoretical computer science.

In other words no problem has been found for which it can be proved that it is in NP but not in P.

NP complete problems are hardest among NP problems

Reason is that

A problem that is NP complete can essentially be used as a subroutine for any problem in NP, with only a polynomial amount of overhead.

Thus if a problem has a polynomial time algorithm then every problem in NP must also have .

Polynomially equivalent problems

Two problems L_1 and L_2 are said to be polynomially equivalent if and only if

$L_1 \alpha L_2$ and $L_2 \alpha L_1$

Some known NP complete problems

Vertex cover problem:

Optimization problem: Given an undirected graph G find a vertex cover for G with as few vertices as possible.

Decision problem: Given an undirected graph G and an integer k , does G have a vertex cover consisting of k vertices.

Example of NP complete problem

Graph coloring , Hamiltonian cycle Hamiltonian path job scheduling with penalities, bin packing ,the subset sum ,the knapsack problem

To prove that problem Q in NP is NP complete

If it is known that problem Q is NP

Then some other NP complete problem is to shown reducible to Q.

Since reduciblity is transitive

Siatisfiability is reducible to Q

So Q is NP complete.

Example of an NP hard problem that is not NP complete

The Halting Problem:

For an arbitrary deterministic algorithm A and input I whether algorithm A with input I ever terminates(or enters an infinite loop).

In simple words halting problem is to determine whether an arbitrary given algorithm (or computer program) will eventually halt (rather than say, get into infinite loop) while working on a given input.

It is well known that it is undecidable so there exists no algorithm of any complexity to solve it.

So it can not be in NP.

Is it possible to have a compiler having extra feature that not only detects syntax errors but also all infinite loops?

Example of reduction

Ex:Traveling salesman decision problem is NP complete?

Suppose we already know that Hamiltonian cycle problem is NP complete.

“Given a complete graph G=(V.E) with edge costs and an integer K, does there exists a simple cycle that visits all vertices and has total cost <= K”

This problem is different than hamiltonian cycle problem because all possible edges are present in G and graph is weighted.

It is easy to see that a solution can be checked in polynomial time so it is certainly in NP.

To show that it is NP complete we polynomially reduce the hamiltonian cycle problem to it.

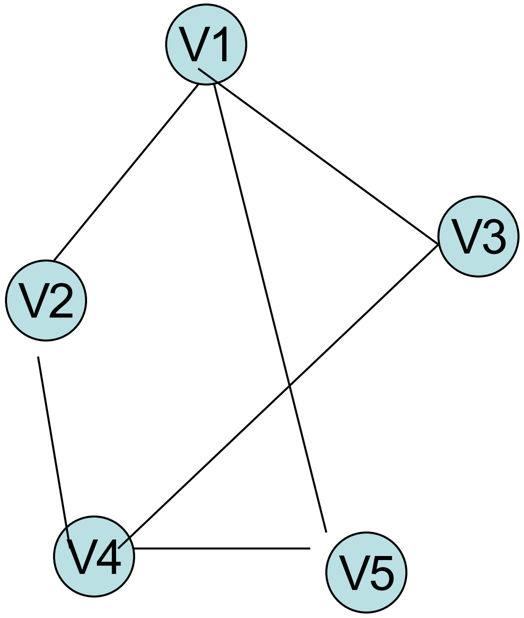
Construct a new graph G’ as:

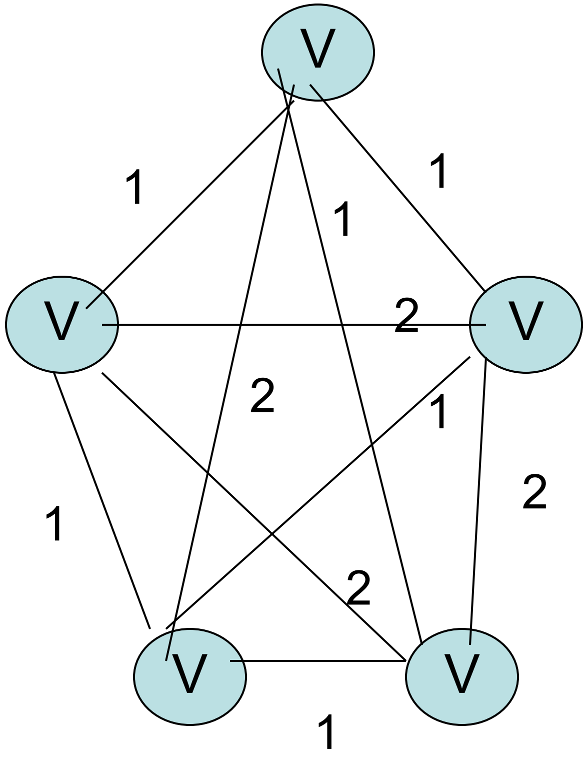
G’ has the same set of vertices as G.

For G’ each edge (v , w) has a weight of 1 if (v, w) is in G and 2 otherwise.

Choose K = |V|.

It is eay to verify that G has a Hamiltonian cycle if and only if G’ has a traveling salesman tour of total weight |V|


V1
V2
V3
V4
V5


V
V
V
V
V
1
1
1
2
2
1
2
1
2
1
1