NP-Hard and NP-Complete problems

Basic Concepts:

For many of the problems we know and study, the best algorithms for their solution have computing times can be clustered into two groups;

1. Solutions are bounded by the **polynomial**- Examples include Binary search O(log n), Linear search O(n), sorting algorithms like merge sort O(n log n), Bubble sort $O(n^2)$ &matrix multiplication $O(n^3)$ or in general $O(n^k)$ where k is a constant.
2. Solutions are bounded by a non-polynomial-Examples include travelling salesman problem $O(n^2 2^n)$ & knapsack problem $O(2^{n/2})$. As the time increases exponentially, even moderate size problems cannot be solved.

So far, no one has been able to device an algorithm which is bounded by the polynomial for the problems belonging to the non-polynomial. However impossibility of such an algorithm is not proved.

Deterministic and non-deterministic algorithms

Deterministic: The algorithm in which every operation is uniquely defined is called deterministic algorithm.

Non-Deterministic: The algorithm in which the operations are not uniquely defined but are limited to specific set of possibilities for every operation, such an algorithm is called non-deterministic algorithm.

The non-deterministic algorithms use the following functions:

1. Choice: Arbitrarily chooses one of the element from given set.
2. Failure: Indicates an un successful completion
3. Success: Indicates a successful completion

A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal. Whenever, there is a set of choices that leads to a successful completion, then one such set of choices is selected and the algorithm terminates successfully.

In case the successful completion is not possible, then the complexity is O(1). In case of successful signal completion then the time required is the minimum number of steps needed to reach a successful completion of O(n) where n is the number of inputs.

The problems that are solved in polynomial time are called tractable problems and the

problems that require super polynomial time are called non-tractable problems. All deterministic polynomial time algorithms are tractable and the non-deterministic polynomials are intractable.

```
1   Algorithm NSort(A, n)
2   // Sort n positive integers.
3   {
4       for i := 1 to n do B[i] := 0; // Initialize B[ ].
5       for i := 1 to n do
6       {
7           j := Choice(1, n);
8           if B[j] ≠ 0 then Failure();
9           B[j] := A[i];
10      }
11      for i := 1 to n − 1 do  // Verify order.
12          if B[i] > B[i + 1] then Failure();
13      write (B[1 : n]);
14      Success();
15  }
```

Nondeterministic sorting

```
1   Algorithm DKP(p, w, n, m, r, x)
2   {
3       W := 0; P := 0;
4       for i := 1 to n do
5       {
6           x[i] := Choice(0, 1);
7           W := W + x[i] * w[i]; P := P + x[i] * p[i];
8       }
9       if ((W > m) or (P < r)) then Failure();
10      else Success();
11  }
```

Nondeterministic knapsack algorithm

Satisfiability Problem:

The satisfiability is a boolean formula that can be constructed using the following literals and operations.

1. A literal is either a variable or its negation of the variable.
2. The literals are connected with operators $\vee, \wedge, \Rightarrow, \Leftrightarrow$
3. Parenthesis

The satisfiability problem is to determine whether a Boolean formula is true for some assignment of truth values to the variables. In general, formulas are expressed in Conjunctive Normal Form (CNF).

A Boolean formula is in conjunctive normal form iff it is represented by

$(x_i \vee x_j \vee x_k^1) \wedge (x_i \vee x^1_j \vee x_k)$

A Boolean formula is in 3CNF if each clause has exactly 3 distinct literals. Example:

The non-deterministic algorithm that terminates successfully iff a given formula E(x1,x2,x3) is satisfiable.

```
1   Algorithm Eval(E, n)
2   // Determine whether the propositional formula E is
3   // satisfiable. The variables are x_1, x_2, ..., x_n.
4   {
5       for i := 1 to n do  // Choose a truth value assignment.
6           x_i := Choice(false, true);
7       if E(x_1, ..., x_n) then Success();
8       else Failure();
9   }
```

Nondeterministic satisfiability

Reducability:
A problem Q1 can be reduced to Q2 if any instance of Q1 can be easily rephrased as an instance of Q2. If the solution to the problem Q2 provides a solution to the problem Q1, then these are said to be reducible problems.

Let L1 and L2 are the two problems. L1 is reduced to L2 iff there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time and is denoted by L1α L2.
If we have a polynomial time algorithm for L2 then we can solve L1 in polynomial time.
Two problems L1 and L2 are said to be polynomially equivalent iff L1α L2 and L2 α L1.

Example: Let P1 be the problem of selection and P2 be the problem of sorting. Let the input have n numbers. If the numbers are sorted in array A[ ] the ith smallest element of the input can be obtained as A[i]. Thus P1 reduces to P2 in O(1) time.

Decision Problem:
Any problem for which the answer is either yes or no is called decision problem. The algorithm for decision problem is called decision algorithm.
Example: Max clique problem, sum of subsets problem.

Optimization Problem: Any problem that involves the identification of an optimal value (maximum or minimum) is called optimization problem.
Example: Knapsack problem, travelling salesperson problem.
In decision problem, the output statement is implicit and no explicit statements are permitted. The output from a decision problem is uniquely defined by the input

parameters and algorithm specification.

Many optimization problems can be reduced by decision problems with the property that a decision problem can be solved in polynomial time if the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time then the optimization problem cannot be solved in polynomial time.

Class *P:*

P: the class of decision problems that are solvable in O($p(n)$) time, where $p(n)$ is a polynomial of problem‟s input size n

Examples:

- searching
- element uniqueness
- graph connectivity
- graph acyclic
- primality testing

Class *NP*

NP(*nondeterministic polynomial*): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:

- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

By definition, it solves the problem if it‟s capable of generating and verifying a solution on one of its tries

Example: CNF satisfiability

Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true? This problem is in *NP*.

Nondeterministic algorithm:

- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

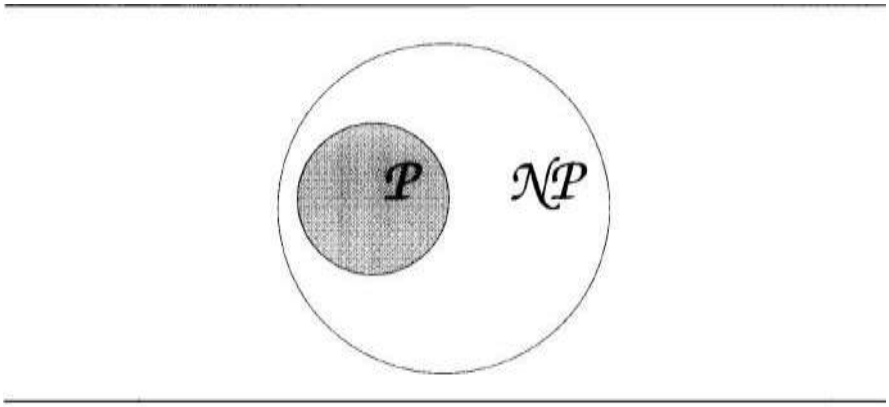
What problems are in *NP*?

- Hamiltonian circuit existence
- Partition problem: Is it possible to partition a set of n integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)
- All the problems in *P* can also be solved in this manner (but no guessing is

necessary), so we have:

$$P \subseteq NP$$

- Big question: $P = NP$?



Commonly believed relationship between $\mathcal{P}$ and $\mathcal{NP}$

NP HARD AND NP COMPLETE CLASSES

Polynomial Time algorithms

Problems whose solutions times are bounded by polynomials of small degree are called polynomial time algorithms

Example: Linear search, quick sort, all pairs shortest path etc.

Non- Polynomial time algorithms

Problems whose solutions times are bounded by non-polynomials are called non-polynomial time algorithms

Examples: Travelling salesman problem, 0/1 knapsack problem etc

It is impossible to develop the algorithms whose time complexity is polynomial for non-polynomial time problems, because the computing times of non-polynomial are greater than polynomial. A problem that can be solved in polynomial time in one model can also be solved in polynomial time.

NP-Hard and NP-Complete Problem:

Let P denote the set of all decision problems solvable by deterministic algorithm in polynomial time. NP denotes set of decision problems solvable by nondeterministic algorithms in polynomial time. Since, deterministic algorithms are a special case of nondeterministic algorithms, P ⊆ NP. The nondeterministic polynomial time problems can be classified into two classes. They are

1. NP Hard and
2. NP Complete

NP-Hard: A problem L is NP-Hard iff satisfiability reduces to L i.e., any

nondeterministic polynomial time problem is satisfiable and reducible then the problem is said to be NP-Hard.

Example: Halting Problem, Flow shop scheduling problem

NP-Complete: A problem L is NP-Complete iff L is NP-Hard and L belongs to NP (nondeterministic polynomial).

A problem that is NP-Complete has the property that it can be solved in polynomial time iff all other NP-Complete problems can also be solved in polynomial time. (NP=P)

If an NP-hard problem can be solved in polynomial time, then all NP- complete problems can be solved in polynomial time. All NP-Complete problems are NP-hard, but some NP-hard problems are not known to be NP- Complete.

Normally the decision problems are NP-complete but the optimization problems are NP-Hard.

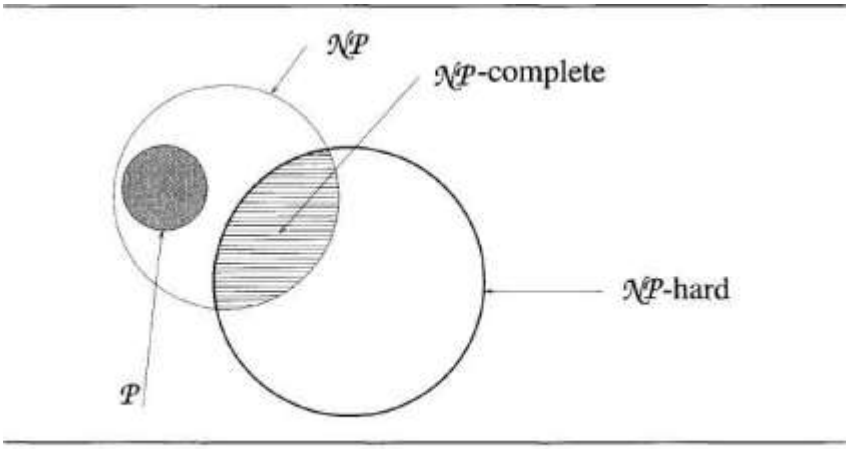
However if problem L1 is a decision problem and L2 is an optimization problem, then it is possible that L1α L2.

Example: Knapsack decision problem can be reduced to knapsack optimization problem.

There are some NP-hard problems that are not NP-Complete.

Relationship between P,NP, NP-hard, NP-Complete

Let P, NP, NP-hard, NP-Complete are the sets of all possible decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic



Commonly believed relationship among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard problems

algorithms, NP-Hard and NP-complete respectively. Then the relationship between P, NP, NP-hard, NP-Complete can be expressed using Venn diagram as:

Problem conversion

A decision problem D1 can be converted into a decision problem D2 if there is an algorithm which takes as input an arbitrary instance I1 of D1 and delivers as output an instance I2 of D2such that I2 is a positive instance of D2 if and only if I1 is a positive instance of D1. If D1 can be converted into D2, and we have an algorithm which solves D2, then we thereby have an algorithm which solves D1. To solve an instance I of D1, we first use the conversion algorithm to generate an instance I0 of D2, and then use the algorithm for solving D2 to determine whether or not I0 is a positive instance of D2. If it is, then we know that I is a positive instance of D1, and if it is not, then we know that I is a negative instance of D1. Either way, we have solved D1 for that instance. Moreover, in this case, we can say that the computational complexity of D1 is at most the sum of the computational complexities of D2 and the conversion algorithm. If the conversion algorithm has polynomial complexity, we say that D1 is at most polynomially harder than D2.ItmeansthattheamountofcomputationalworkwehavetodotosolveD1,over and

above whatever is required to solve D2, is polynomial in the size of the problem instance. In such a case the conversion algorithm provides us with a feasible way of solving D1, given that we know how to solve D2.

Given a problem X, prove it is in NP-Complete.

1. Prove X is in NP.
2. Select problem Y that is known to be in NP-Complete.
3. Define a polynomial time reduction from Y toX.
4. Prove that given an instance of Y, Y has a solution iff X has a solution.

Cook's theorem:

Cook"s Theorem implies that any NP problem is at most polynomially harder than SAT. This means that if we find a way of solving SAT in polynomial time, we will then be in a position to solve any NP problem in polynomial time. This would have huge practical repercussions, since many frequently encountered problems which are so far believed to

be intractable are NP. This special property of SAT is called NP-completeness. A decision problem is NP-complete if it has the property that any NP problem can be converted into it in polynomial time. SAT was the first NP-complete problem to be recognized as such (the theory of NP-completeness having come into existence with the proof of Cook‟s Theorem), but it is by no means the only one. There are now literally thousands of problems, cropping up in many different areas of computing, which have been proved to be NP-complete.

In order to prove that an NP problem is NP-complete, all that is needed is to show that SAT can be converted into it in polynomial time. The reason for this is that the sequential composition of two polynomial-time algorithms is itself a polynomial-time algorithm, since the sum of two polynomials is itself a polynomial.

Suppose SAT can be converted to problem D in polynomial time. Now take any NP problem D0. We know we can convert it into SAT in polynomial time, and we know we can convert SAT into D in polynomial time. The result of these two conversions is a polynomial-time conversion of D0 into

D. since D0 was an arbitrary NP problem, it follows that D is NP-complete