UNIT-8

NP Completeness

⊛. Tractable and Intractable Problems: [Imp]

If a problem can be solved using polynomial time algorithms then, we call it as tractable problem. Following are the examples of tractable problems:

→ Searching an unordered list
→ Searching an ordered list
→ Sorting a list
→ Multiplication of integers etc.

The problems that cannot be solved in polynomial time but requires super-polynomial time algorithm are called intractable or hard problems. Following are some examples of intractable problems:

→ Towers of Hanoi.
→ List of all permutations of n numbers etc.

⊛. Polynomial Time and Super Polynomial Time Complexity:

Following are some common functions, ordered by how fast they grow:

Constant $O(1)$
Logarithmic $O(\log n)$
Linear $O(n)$
n-log-n $O(n \log n)$
Quadratic $O(n^2)$
Cubic $O(n^3)$
Exponential $O(k^n)$, e.g. $O(2^n)$
Factorial $O(n!)$
Super-exponential $O(n^n)$.

Scientists divide these functions into two classes as: polynomial running time and Super Polynomial time complexity.

Polynomial Running Time:

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the complexity of the input. Polynomial time algorithms are said to be "fast". Addition, Multiplication, Subtraction, Division etc. and mathematical constants pi, e etc. can be done in polynomial time. Example: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ etc. belongs to polynomial time complexity.

Super Polynomial Time Complexity:

An algorithm is said to be solvable in super polynomial time if it exceeds the number of steps than that of polynomial time (i.e, not bounded by input $O(n^k)$) is called super polynomial time complexity. Super Polynomial time algorithms are said to be "slow". $O(n^n)$ is an example of super polynomial time complexity.

⊛. Complexity Classes: [Imp]

i) Class-P:

The class P consists of those problems that can be solved by a deterministic Turing machine in Polynomial time. P problems are tractable.

Example: Addition, Subtraction etc. of two numbers.

ii) NP-Class:

NP is a set of decision problems that can be solved by a Non-deterministic machine in Polynomial time. P is subset of NP. The class NP consists of those problems that are verifiable in polynomial time.

Example: Factorization is an example of NP-class problems since it is not solvable in polynomial time but the solution is verifiable in polynomial time.

iii) NP-Comple:

This represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time. NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

→ L is in NP
→ Every problem in NP is reducible to L in polynomial time.

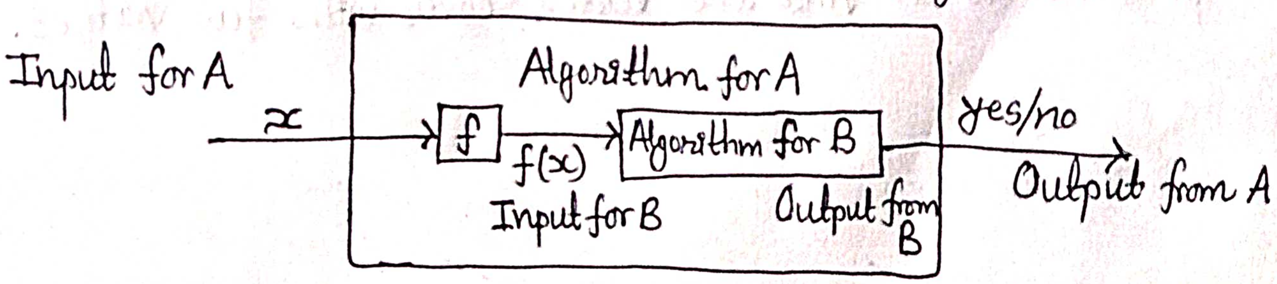
Example: Graph Isomorphism.

iv) NP-hard: A problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time. Since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

Example: The halting problem.

Polynomial Time Reduction:

Given two decision problems A and B, a polynomial time reduction from A to B is a polynomial time function f that transforms the instances of A into instances of B such that the output of algorithm for the problem A on input instance x must be same as the output of the algorithm for the problem B on input instance $f(x)$. If there is polynomial time computable function f such that it is possible to reduce A to B, then it is denoted as $A \le_p B$. The function f described above is called reduction function and the algorithm for computing f is called reduction algorithm.



Cooks Theorem: Cook's theorem states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

An important consequence of this theorem is that if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm. The question of whether such an algorithm for Boolean satisfiability exists is thus equivalent to the P versus NP problem, which is widely considered the most unsolved problem in theoretical computer science.

Approximation Algorithms:

An approximate algorithm is a way of approach NP-completeness for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

Applications:

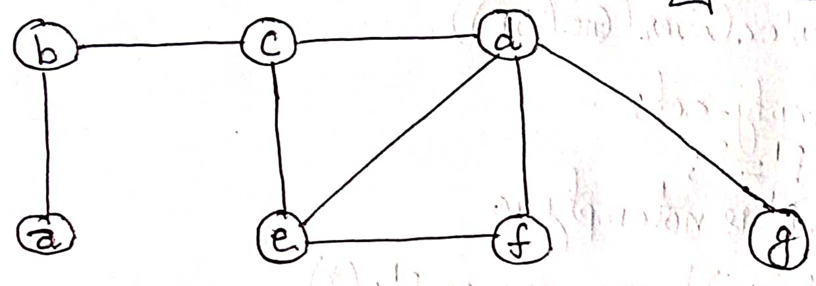
i) For the travelling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.

ii) For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Vertex Cover Problem: [Imp]

A Vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices. Now we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C^*. The idea is to take an edge (u,v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed.

Example: Find minimum vertices covered by using vertex cover problem.



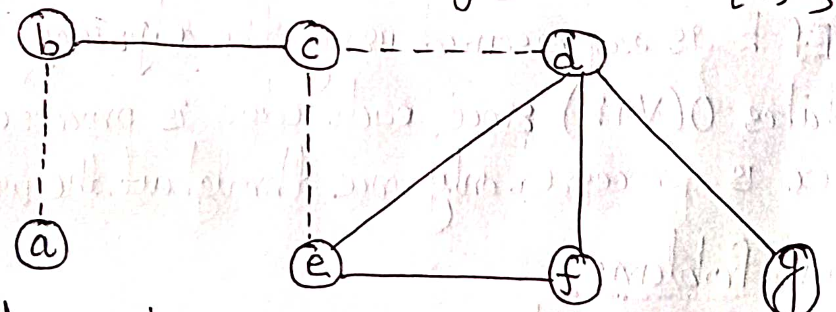
Solution:

$E = \{(a,b), (b,c), (c,d), (c,e), (d,e), (d,f), (e,f), (d,g)\}$

Step1: Lets choose edge (b,c).

Now eliminate edges incident to vertex b and c.

$E = \{(d,e), (d,f), (e,f), (d,g)\}$ $C = \{b,c\}$

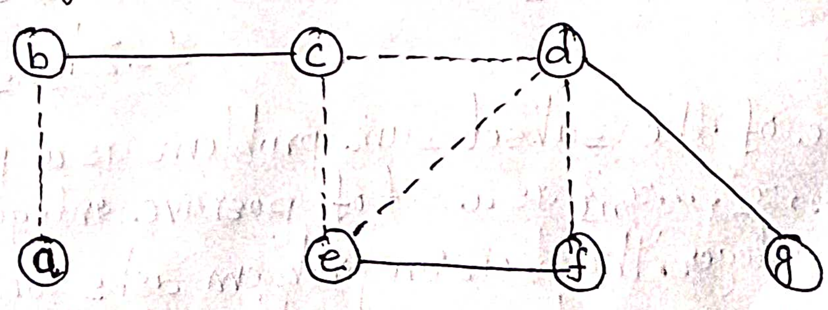


Step2: Lets choose edge (e,f).

$C = \{b,c,e,f\}$

Eliminate edges incident on vertex e and f.

$E = \{(d,g)\}$

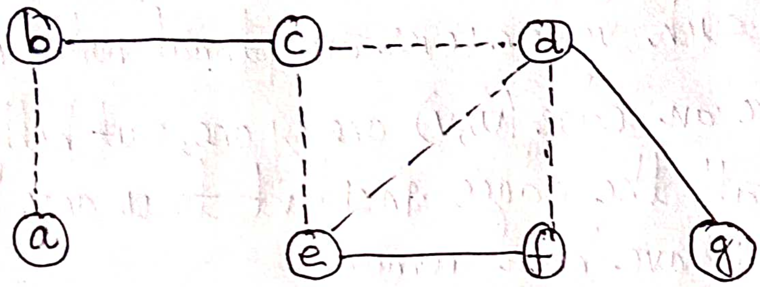


Step3: Let's choose edge (d,g)

$C=\{b,c,e,f,d,g\}$

Eliminate edges incident to vertex d and g.

$E=\{\Phi\}$



Algorithm:

```
ApproxVertexCover(G=(V,E))
{   C = empty-set;
        E' = E;
    while E' is notempty do
    {  Let (u,v) be any edge in E': (*)
       Add u and v to C;
       Remove from E' all edges incident to u or v;
    }
     Return C;
}
```

Analysis: If E is represented using the adjacency lists the above algorithm takes $O(V+E)$ since each edge is processed only once and every vertex is processed only once throughout the whole operation.

⊛. Subset Sum Problem:

In the subset sum problem, we are given a finite set $S \subseteq N$ and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.

$$\text{Sub-set sum} = \{(S,t) : \text{there exists a subset } S' \subseteq S \text{ such that } t = \textstyle\sum_{s \in S'} s\}$$

An instance of the subset sum problem is a pair (S,t), where $S=\{x_1, x_2, \ldots x_n\}$ is a set of positive integers and t is a positive integer. The decision problem asks for a subset of S whose sum is as large as possible, but not larger than t.