Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture 8– P, NP- Complete and NP Hard Problems

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Introduction

- An algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to *O(p(n))* where *p(n)* is a polynomial of the problem's input size *n.*
- Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.
- On an input of size n the worst-case running time is O(n k) for some constant k – $O(n^2)$, $O(n^3)$, O(1), O(n lg n), $O(2^n)$, $O(n^n)$, O(n!) –
  - 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!)

The Class P

- P: the class of problems that have polynomial-time deterministic algorithms.
  - That is, they are solvable in O(p(n)), where p(n) is a polynomial on n
  - A deterministic algorithm is (essentially) one that always computes the correct answer.
- Examples in P
  - Sorting
  - Searching
  - Fractional Knapsack
  - Minimum Spanning Tree

Nondeterministic Algorithm

- Deterministic Algorithm.
- Nondeterministic Algorithm.
  - Can make a choice randomly according to a distribution.
  - Guessing stage and verification stage.
  - Three new functions

1. Choice(S) arbitrarily chooses one of the elements of set S.
2. Failure() signals an unsuccessful completion.
3. Success() signals a successful completion.

Nondeterministic Searching Algorithm

Consider the problem of searching for an element x in a given set of elements $A[1:n]$, $n \geq 1$. We are required to determine an index j such that $A[j] = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is

---

```
1    j := Choice(1, n);
2    if A[j] = x then {write (j); Success();}
3    write (0); Failure();
```

Nondeterministic Sorting Algorithm

```
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 Algorithm

[Sorting] Let $A[i]$, $1 \leq i \leq n$, be an unsorted array of positive integers. The nondeterministic algorithm NSort(A, n) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B[1 : n]$ is used for convenience. Line 4 initializes B to zero though any value different from all the $A[i]$ will do. In the **for** loop of lines 5 to 10, each $A[i]$ is assigned to a position in B. Line 7 nondeterministically determines this position. Line 8 ascertains that $B[j]$ has not already been used. Thus, the order of the numbers in B is some permutation of the initial order in A. The **for** loop of lines 11 and 12 verifies that B is sorted in nondecreasing order. A successful completion is achieved if and only if the numbers are output in nondecreasing order. Since there is always a set of choices at line 7 for such an output order, algorithm NSort is a sorting algorithm. Its complexity is $O(n)$. Recall that all deterministic sorting algorithms must have a complexity $\Omega(n \log n)$. □

Verification algorithms

- ***Verification algorithm*** as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***.
- A two-argument algorithm A ***verifies*** an input string x if there exists a certificate y such that A(x, y)= 1.
- The ***language verified*** by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}$$

The Class NP

- Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.
- Most decision problems are in *NP*.
- First of all, this class includes all the problems in *P* :

$$P \subseteq NP.$$

- A decision problem *D* is said to be ***NP-complete*** if:
  - it belongs to class *NP*
  - every problem in *NP* is polynomially reducible to *D*.

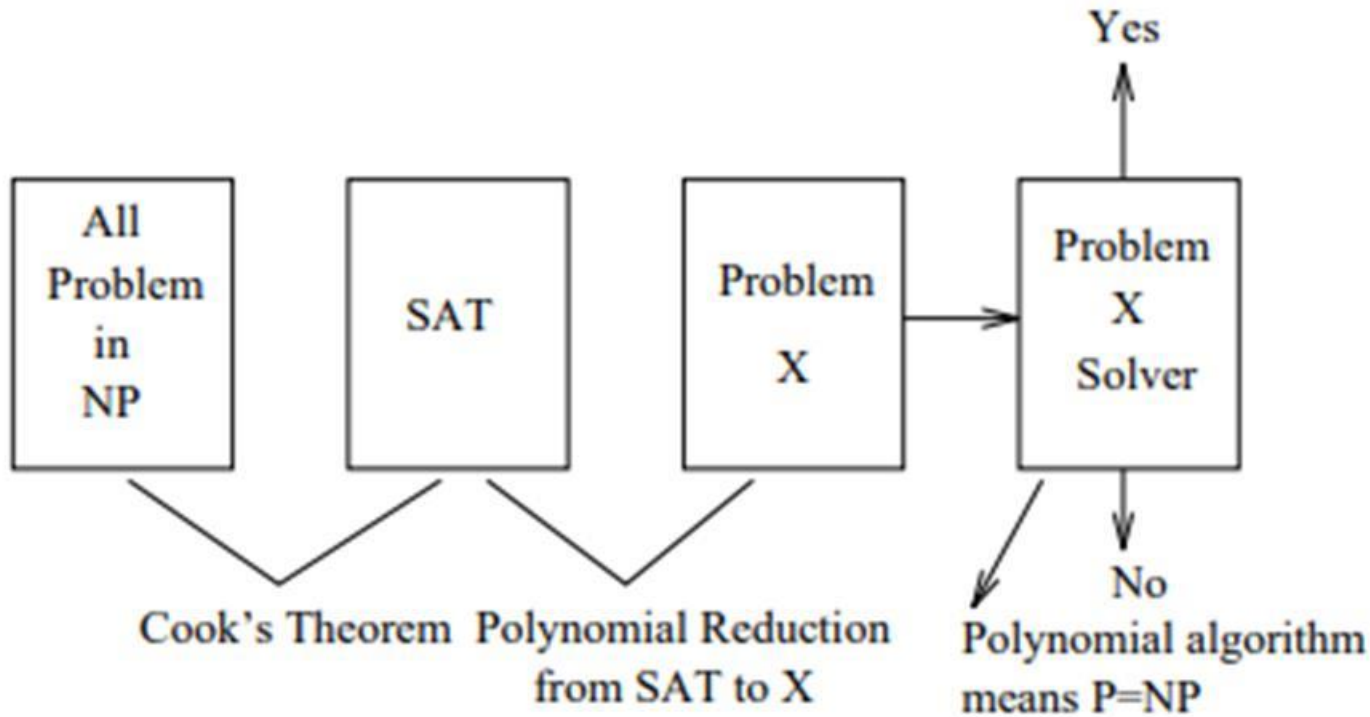
Reduciblilty

- A decision problem *D*1 is said to be ***polynomially reducible*** to a decision problem *D*2, if there exists a function *t* that transforms instances of *D*1 to instances of *D*2 such that:
  - *t* maps all yes instances of *D*1 to yes instances of *D*2 and all no instances of *D*1 to no instances of *D*2
  - *t* is computable by a polynomial time algorithm
- This definition immediately implies that if a problem *D*1 is polynomially reducible to some problem *D*2 that can be solved in polynomial time, then problem*D*1 can also be solved in polynomial time.

Cook's theorem

- Cook's Theorem states that
  - Any NP problem can be converted to SAT in polynomial time.
- 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.
- 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 recognised as such (the theory of NP-completeness having come into existence with the proof of Cook's Theorem)

Cook's theorem



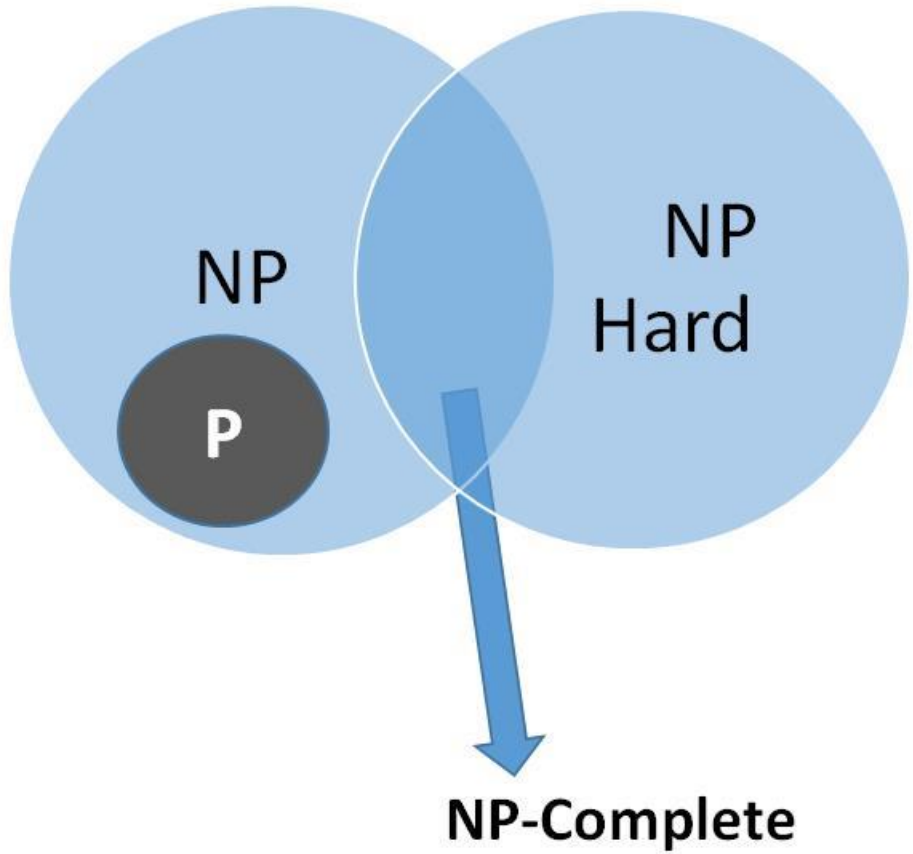
NP-complete

- NP-Complete problems can be solved by a non-deterministic Algorithm or Turing Machine in polynomial time.
- It is exclusively a Decision problem
- Example:
  - Determine whether a graph has a Hamiltonian cycle,
  - Determine whether a Boolean formula is satisfiable or not,
  - Circuit-satisfiability problem, etc.

NP-Hard

- NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.
- Do not have to be a Decision problem
- Example: Halting problem, Vertex cover problem, etc.

P, NP- Complete and NP Hard Problems

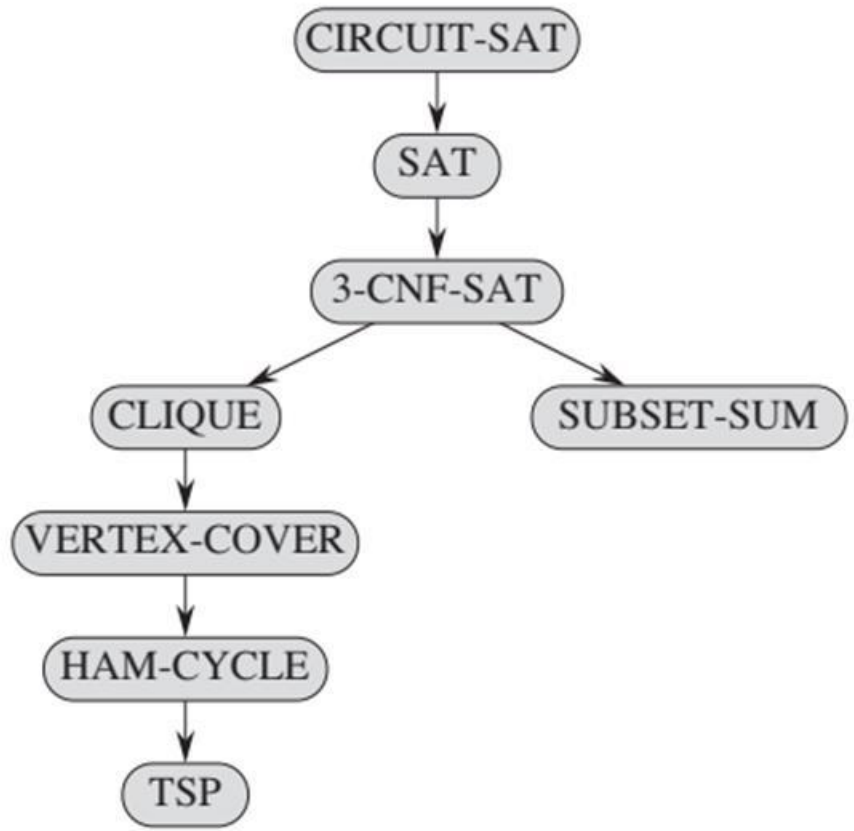


Difference between NP Complete and Np Hard

- A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time.
- NP-Complete problems are as hard as NP problems.
- A problem is NP-Complete if it is a part of both NP and NP-Hard Problem.

- A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time.
- NP-Hard problems are as hard as NP-Complete problems.
- NP-Hard Problem need not be in NP class.

NP-complete problems



Satisfiability Problem

- Let $x_1, x_2, \ldots$ denote a set of boolean variables
- Let $\bar{x_i}$ denote the complement of x_i
- Examples of formulas in propositional calculus
  * $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x_4})$
  * $(x_3 \vee \bar{x_4}) \wedge (x_1 \vee \bar{x_2})$
- Satisfiability problem is to determine whether a formula is true for some assignment of truth values to the variables
- CNF-satisfiability is the satisfiability problem for CNF (conjunctive normal form) formulas
- Non-deterministically choose one of the 2^n possible assignments of truth values to $(x_1 \ldots x_n)$
- Verify that $E(x_1 \ldots x_n)$ is true for that assignment

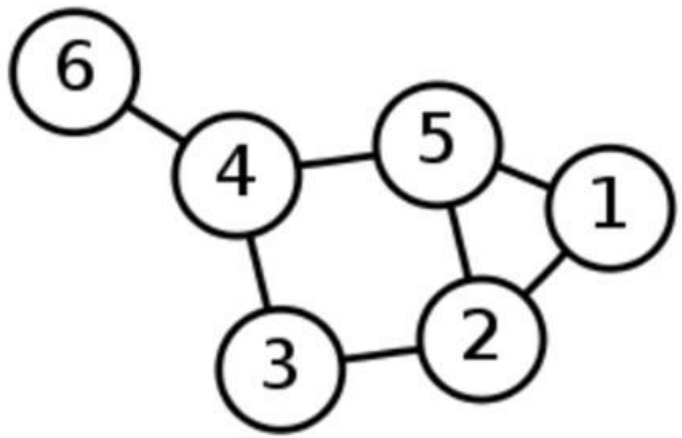
Satisfiability Problem

```
algorithm eval ( E, n )
{
    // Determine whether the propositional formula E is satisfiable.
    // Variable are x1, x2, ..., xn

    // Choose a truth value assignment

    for ( i = 1; i <= n; i++ )
        x_i = choice ( true, false );

    if ( E ( x1, ..., xn ) )
        success();
    else
        failure();
}
```

* The nondeterministic time to choose the truth value is $O(n)$
* The deterministic evaluation of the assignment is also done in $O(n)$ time

Vertex Cover Problem

- The vertex cover problem is to find a vertex cover of minimum size in a given undirected graph.
- Such a vertex cover is called an optimal vertex cover.
- This problem is the optimization version of an NP-complete decision problem.
- Definition: Consider a graph G= (V, E) where V and E are accordingly vertex and edges. A vertex cover of an undirected graph is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, Then either $u \in V'$ or $v \in V'$or both.



vertex covers of size 3, such as {2, 4, 5} and {1, 2, 4}.

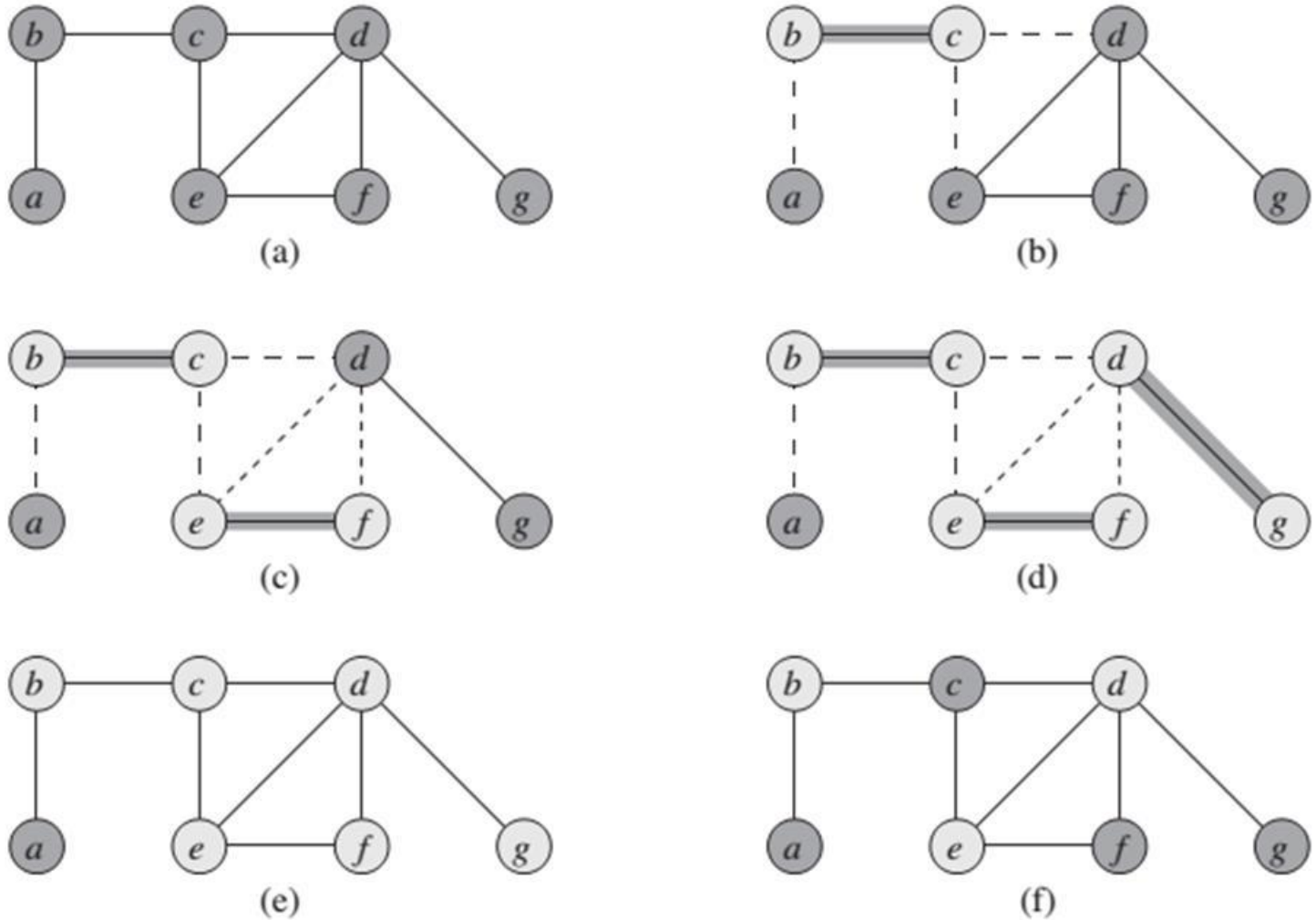
Applications

- Performance of a computer network, for collecting statistics on packets being transmitted. Draw a graph where the vertices are computers/routers, and edges are communication links.
- Installing security cameras covering all the places.

Algorithms for Vertex Cover Problem

- Brute Force
  - All possible subsets – N nodes => O(2^N)
- Approximation Algorithm
- Greedy algorithm
- Dynamic Programming algorithm

Approximation Algorithm



Approximation Algorithm

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4       let (u, v) be an arbitrary edge of E'
5       C = C ∪ {u, v}
6       remove from E' every edge incident on either u or v
7  return C
```

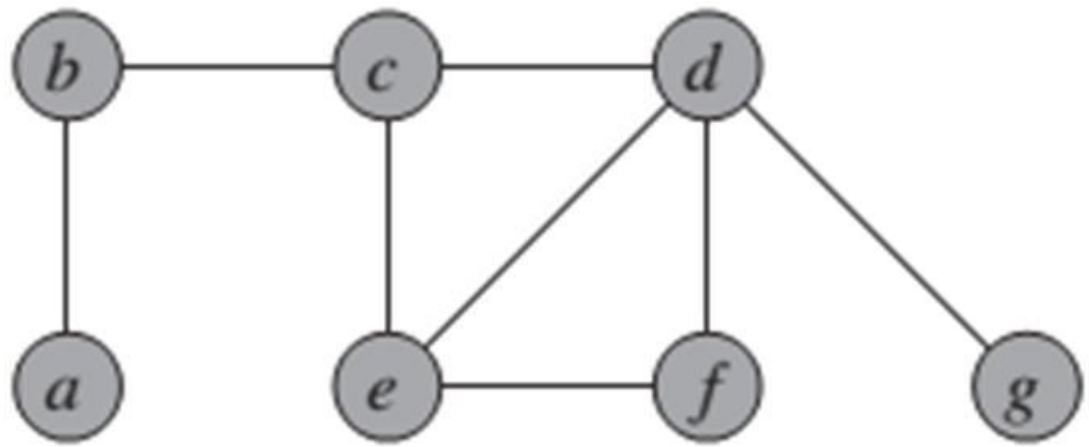
Approximation Algorithm

- The variable C contains the vertex cover being constructed.
- Line 1 initializes C to the empty set. Line 2 sets E’ to be a copy of the edge set G.E of the graph.
- The loop of lines 3–6 repeatedly picks an edge (u, v) from E’, adds its endpoints u and v to C, and deletes all edges in E’ that are covered by either u or v.
- Finally, line 7 returns the vertex cover C.
- The running time of this algorithm O(V + E), using adjacency lists to represent E’.

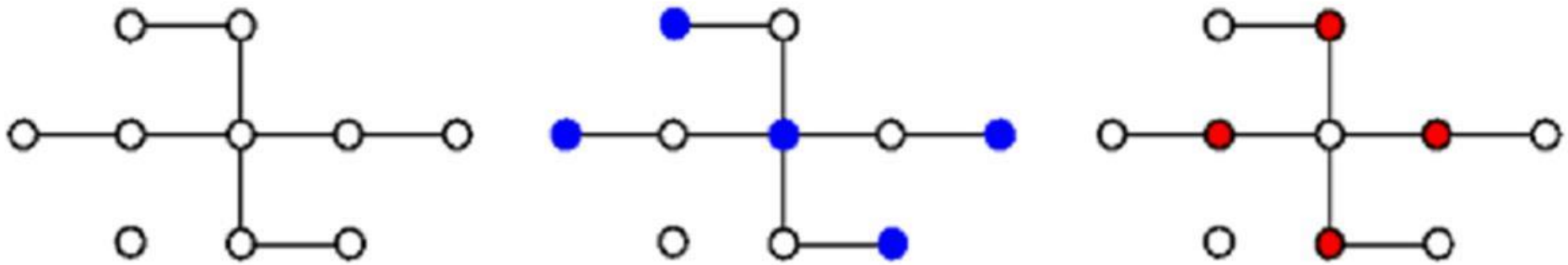
Greedy algorithm

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$
3. Pick a vertex $v \in V$ of maximum degree in the *current* graph
4. $C \leftarrow C \cup \{v\}$
5. $E \leftarrow E \setminus \{e \in E : v \in e\}$
6. **return** C

Greedy algorithm



Greedy algorithm



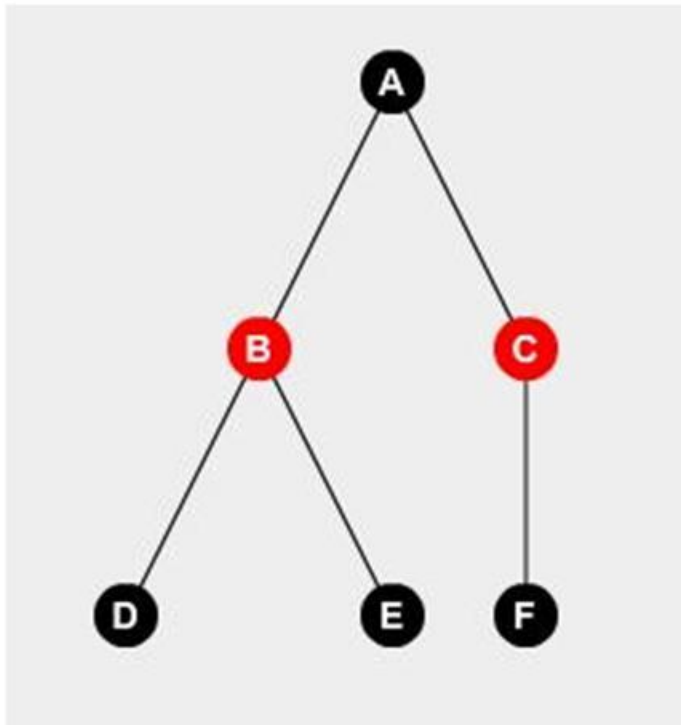
(a) A graph instance

(b) a vertex cover of size 5 obtained by the greedy algorithm

(c) a vertex cover of size 4 optimal solution

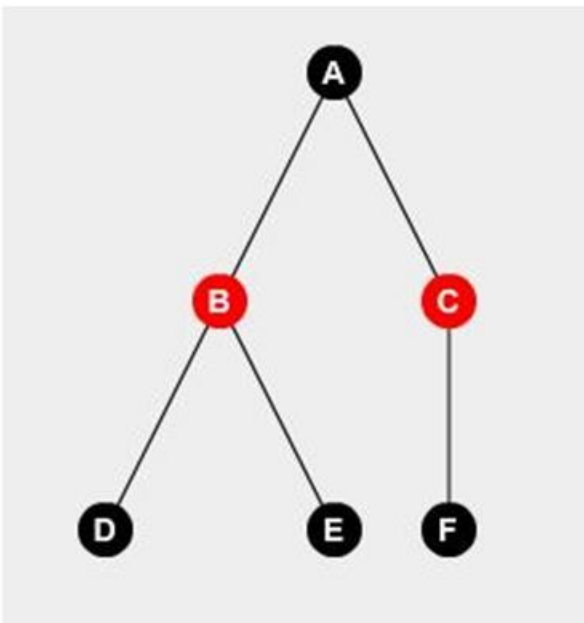
Dynamic Programming algorithm

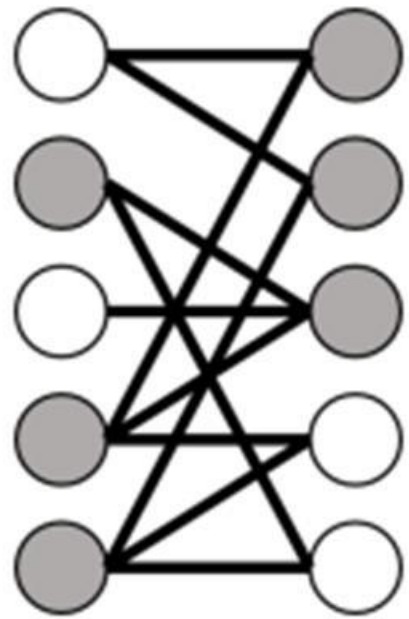
- If the graph was a **Tree**, that means if it had **(n-1)** nodes where **n** is the number of edges and there are no cycle in the graph, we can solve it using dynamic programming.



Dynamic Programming algorithm

- If the graph was a Bipartite Graph or tree, we can solve it using dynamic programming.

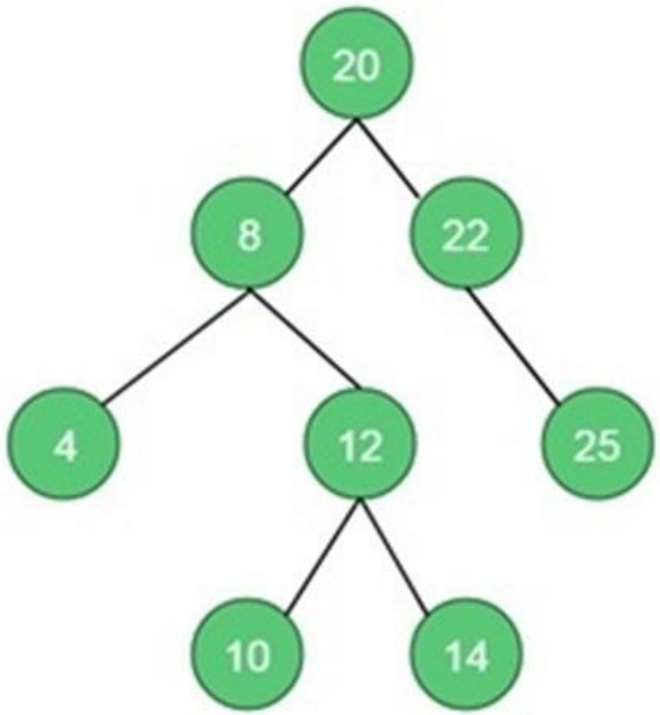


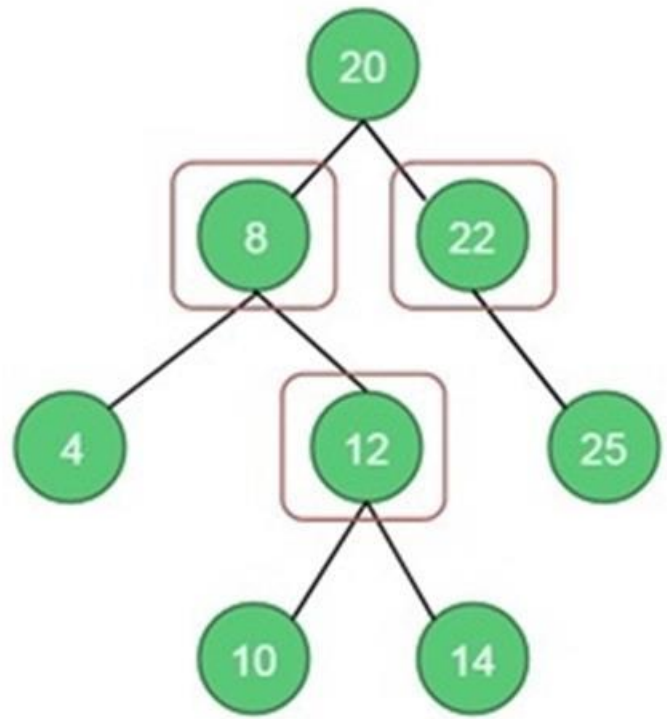


Dynamic Programming algorithm

- The idea of the approach to find minimum vertex cover in a tree is to consider following two possibilities for root and recursively for all the nodes down the root.
  - Root is a part of vertex cover:
    - In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).
  - Root is not a part of vertex cover:
    - In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).
- Finaly we take the minimum of the two, and return the result.
- The brute force time complexity of this approach is exponential i.e. $O(2^n)$ because we recursively solve the same subproblems many times.

Dynamic Programming algorithm





Dynamic Programming algorithm

- Solve the vertex cover problem in bottom up manner and store the solutions of the sub-problems to calculate the minimum vertex cover of the problem.
- In bottom up manner we start from the leaf nodes, compute its minimum vertex cover and store it to that node.
- Now for the parent node we recursively compute the vertex cover from its children nodes and store it to that node.
- In this approach we calculate the vertex cover of any node only once and reuse the stored value if required.

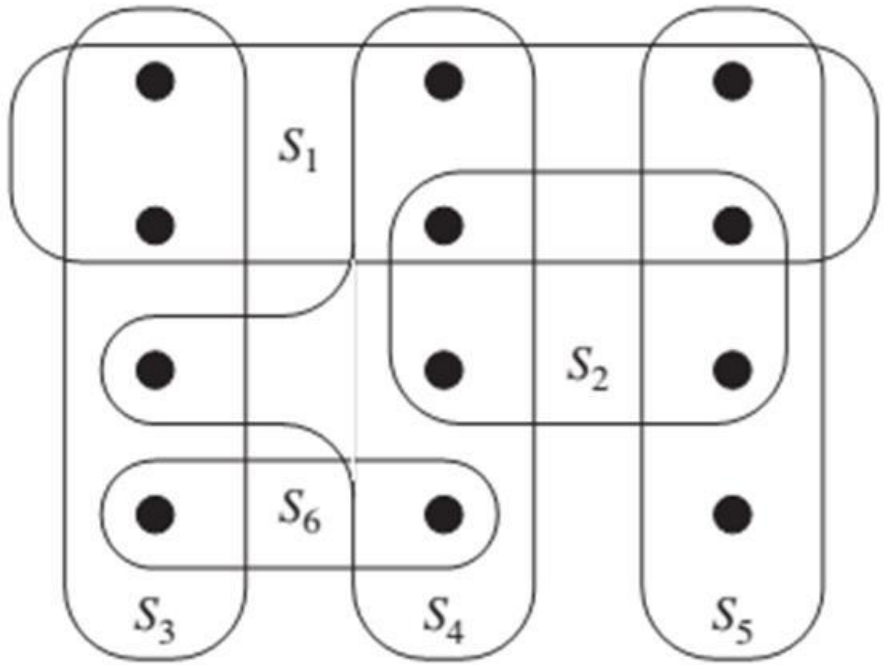
Summary

- The vertex cover (VC) problem belongs to the class of NP-complete graph theoretical problems.
- We are unlikely to find a polynomial-time algorithm for solving vertex-cover problem exactly.
- Minimum vertex cover is one of the Karp‘s 21 diverse combinatorial and graph theoretical problems (Karp, 1972), which were proved to be NP-complete.

The set-covering problem

- The set-covering problem is an optimization problem that models many problems that require resources to be allocated.
- Its corresponding decision problem generalizes the NP-complete vertex-cover problem and is therefore also NP-hard.
- Examine a simple greedy heuristic approach.

The set-covering problem



An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 black points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S_1, S_4, S_5, and S_3 or the sets S_1, S_4, S_5, and S_6, in order.

The set-covering problem

An instance $(X, \mathcal{F})$ of the ***set-covering problem*** consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S \, .$$

We say that a subset $S \in \mathcal{F}$ ***covers*** its elements. The problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S \, . \tag{35.8}$$

A greedy approximation algorithm

```
GREEDY-SET-COVER(X, F)
1  U = X
2  C = ∅
3  while U ≠ ∅
4      select an S ∈ F that maximizes |S ∩ U|
5      U = U − S
6      C = C ∪ {S}
7  return C
```

We can easily implement GREEDY-SET-COVER to run in time polynomial in $|X|$ and $|\mathcal{F}|$. Since the number of iterations of the loop on lines 3–6 is bounded from above by $\min(|X|, |\mathcal{F}|)$, and we can implement the loop body to run in time $O(|X||\mathcal{F}|)$, a simple implementation runs in time $O(|X||\mathcal{F}|\min(|X|, |\mathcal{F}|))$.