# BACKTRACKING

# Introduction

- Some problems can be solved, by exhaustive search. The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property
- Backtracking is a more intelligent variation of this approach.

# Backtracking

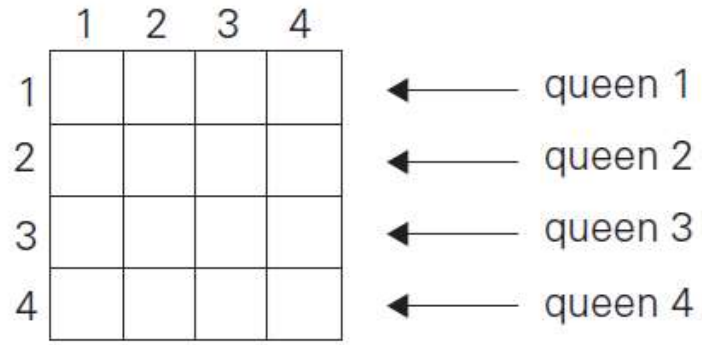
- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
  - If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
  - If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered.
  - In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

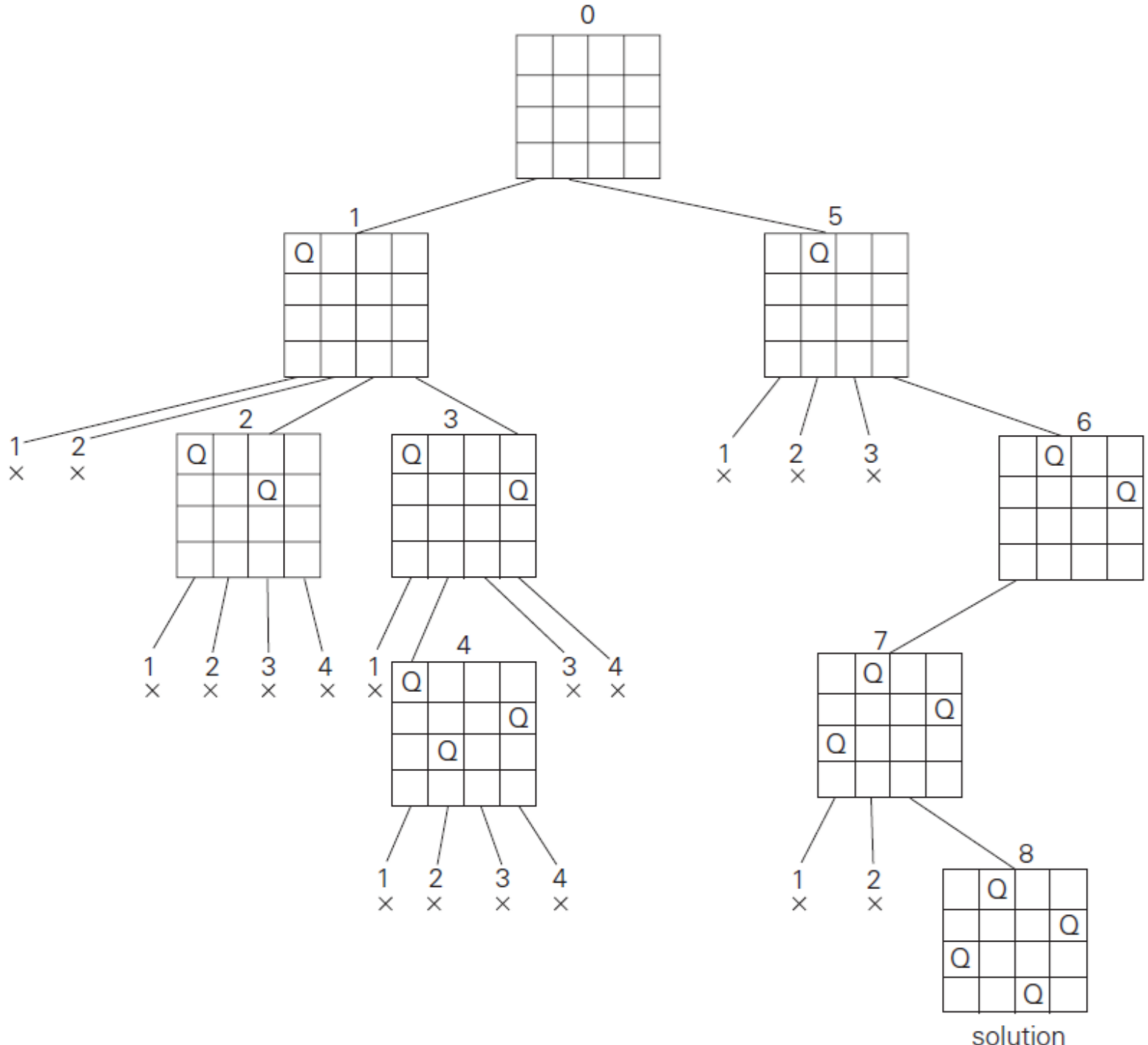
- It is convenient to implement this kind of processing by constructing a tree of choices being made, called the state-space tree.
- Its root represents an initial state before the search for a solution begins.
- The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second component, and soon.
- A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called non-promising.
- Leaves represent either non-promising dead ends or complete solutions found by the algorithm.

- In the majority of cases, a state space tree for a backtracking algorithm is constructed in the manner of depth-first search.
- If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child.
- If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on.
- Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

# N-Queens Problem

- The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
- Consider 4 –Queens Problem
  - Each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board



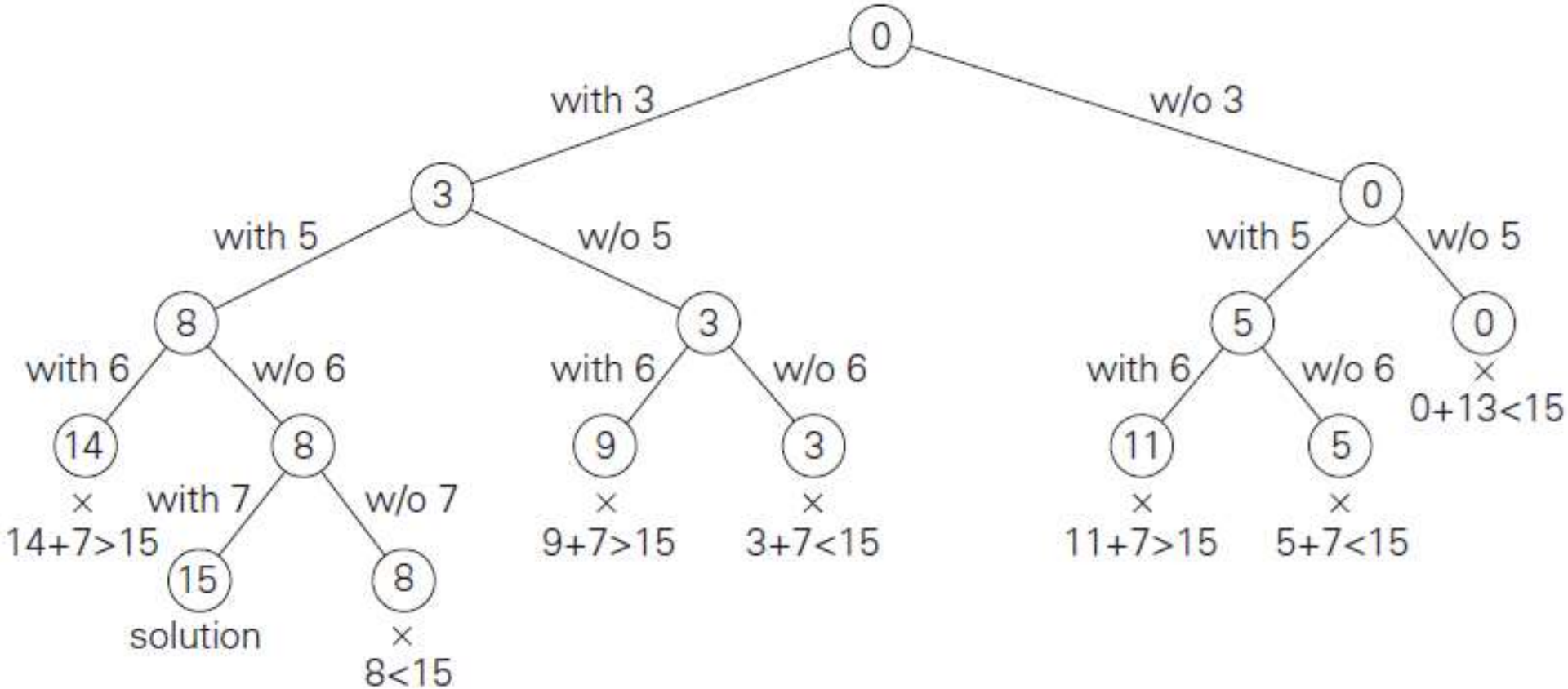

0
1
Q
5
Q
1
2
2
Q
Q
3
Q
Q
1
2
3
6
Q
Q
1
2
3
4
1
4
Q
Q
Q
3
4
1
2
3
4
7
Q
Q
Q
1
2
8
Q
Q
Q
Q
solution

- We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1.
- Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3.
- This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4).
- Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2).
- Queen 2 then goes to (2, 4), queen 3 to(3, 1), and queen 4 to (4, 3), which is a solution to the problem.

# Subset-Sum problem

Find a subset of a given set $A = \{a1, \ldots, an\}$ of $n$ positive integers whose sum is equal to a given positive integer $d$.

For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1, 2, 6} and {1, 8}.




Complete state-space tree of the backtracking algorithm applied to the instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

- The state-space tree can be constructed as a binary tree for the instance $A = \{3, 5, 6, 7\}$ and $d = 15$.
- The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of $a1$ in a set being sought.
- Similarly, going to the left from a node of the first level corresponds to inclusion of $a2$ while going to the right corresponds to its exclusion, and so on.
- Thus, a path from the root to a node on the $i$th level of the tree indicates which of the first $I$ numbers have been included in the subsets represented by that node.
- We record the value of $s$, the sum of these numbers, in the node. If $s$ is equal to $d$, we have a solution to the problem.
- We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If $s$ is not equal to $d$,

We can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

# Branch and Bound Techniques.

* Branch and Bound is an algorithic problem Solving Techniques.

* It is used to solve the optimization problem.

* An optimization problem seeks to minimize or maximize some objective function.

  eg a tour length, the value of items selected

Compared to backtracking, branch and bound requires two additional items.

↳ For every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node.

↳ the value of the best solution seen so far.

we terminate a search path at the current node in a state-space tree of a branch-bound algorithm for any one of the following three reasons.

i) The value of the node's bound is not better than the value of the best solution seen so far.

ii) The node represents no feasible solutions because the constraints of the problem are already violated.

iii) The subset of feasible solutions represented by the node consists of a single point. In this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

# Traveling Salesman Problem

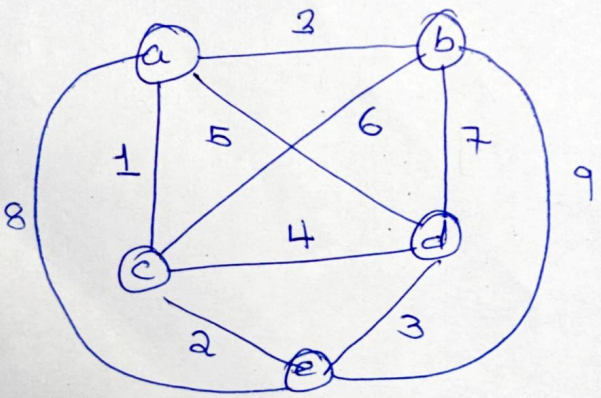
The branch and bound technique to instances of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths.

## Lower bound is calculated as

Find the sum $s_i$ of the distances from city i to the two nearest cities; compute the sum s of these n numbers, divide the result by 2.

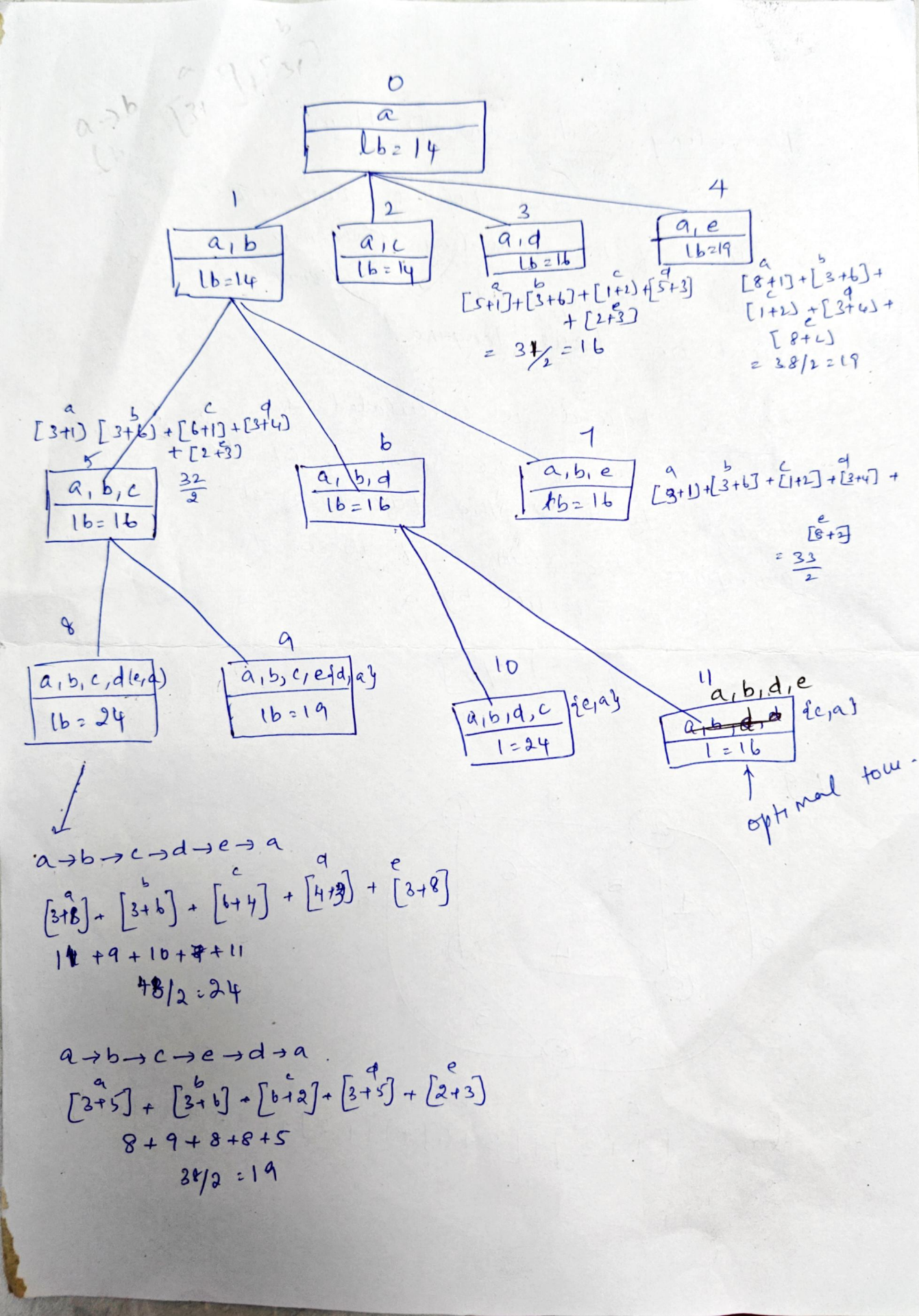
$$lb = \lceil s/2 \rceil$$

Eg:




$$lb = \underset{a}{[1+3]} + \underset{b}{[3+6]} + \underset{c}{[1+2]} + \underset{d}{[3+4]} + \underset{e}{[2+3]}/2$$

$$= 14$$


0
a
lb = 14
1
a, b
lb = 14
2
a, c
lb = 14
3
a, d
lb = 16
4
a, e
lb = 19
[5+1] + [3+6] + [1+2] + [5+3] + [2+3]
= 31/2 = 16
[8+1] + [3+6] + [1+2] + [3+6] + [8+2]
= 38/2 = 19
[3+1] [3+6] + [6+1] + [3+4] + [2+3]
32/2
5
a, b, c
lb = 16
b
a, b, d
lb = 16
7
a, b, e
lb = 16
[3+1] + [3+6] + [1+2] + [3+4] + [8+2]
= 33/2
8
a, b, c, d (e, a)
lb = 24
9
a, b, c, e (d, a)
lb = 19
10
a, b, d, c {e, a}
l = 24
11
a, b, d, e
a, b, d, d {c, a}
l = 16
optimal tour.
a → b → c → d → e → a
[3+8] + [3+6] + [6+4] + [4+3] + [8+8]
14 + 9 + 10 + 7 + 11
48/2 = 24
a → b → c → e → d → a
[3+5] + [3+6] + [6+2] + [3+5] + [2+3]
8 + 9 + 8 + 8 + 5
38/2 = 19

# Knapsack problem using Branch and Bound.

Solve the following Knapsack problem using branch and bound given the following data.

Capacity of Knapsack M = 10

| Item | weight | value | v/w |
|---|---|---|---|
| 1 | 4 | 40 | 10 |
| 2 | 7 | 42 | 6 |
| 3 | 5 | 25 | 5 |
| 4 | 3 | 12 | 4 |

Note: Arrange the item by decreasing order of value/weight ratio.

## The foomula for calculating upper bound

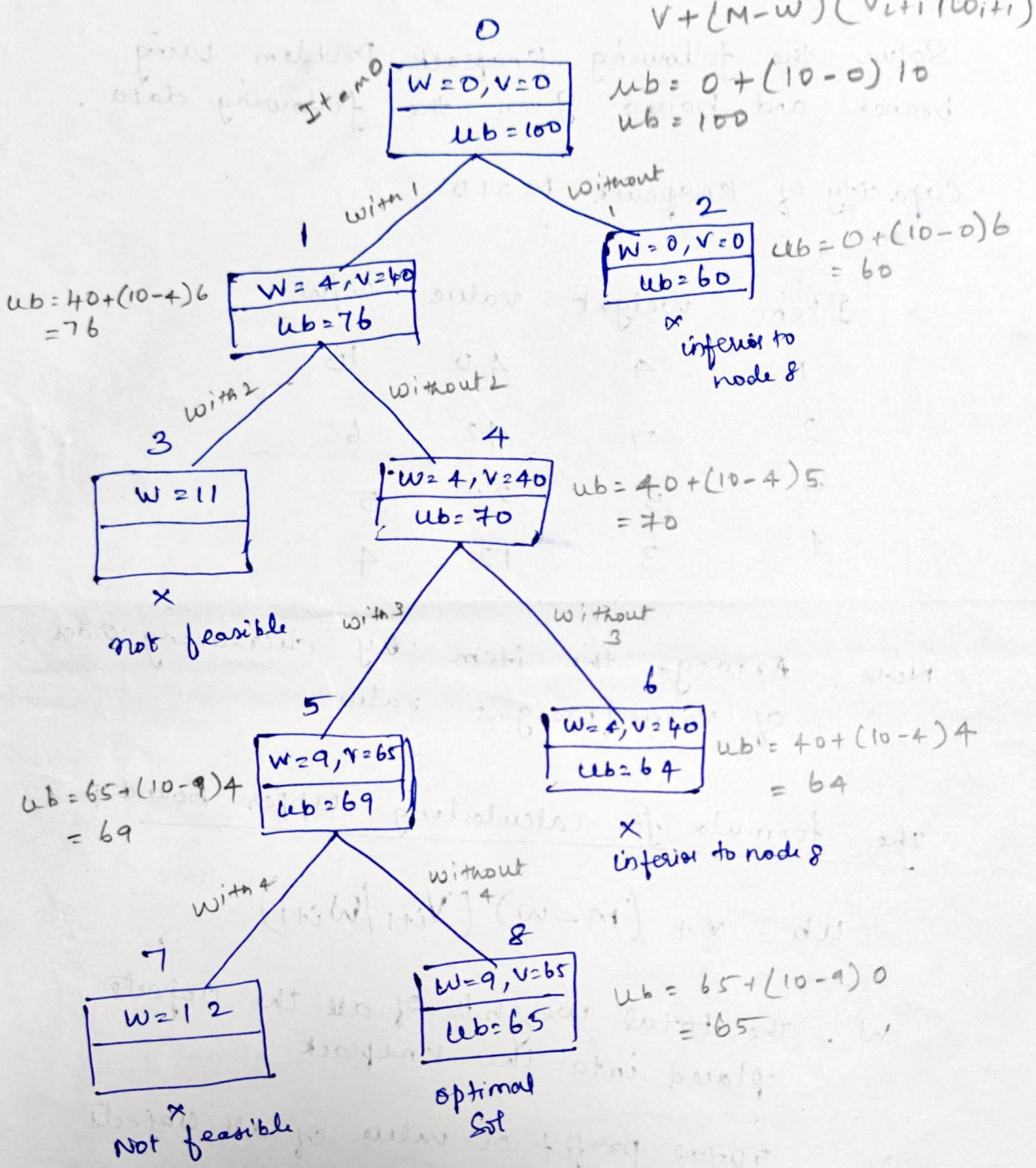
$$ub = v + (M - w)(V_{i+1}/W_{i+1})$$

w - The total weights of all the objects placed into the knapsack.

v - Total profit or value of all objects placed into the knapsack.

# STATE SPACE TREE

$V + (M-W)(V_{i+1}/W_{i+1})$



the board. Needs to collect as many coins as possible and bring them to lower right cell

- A Robot can move one cell down and one cell right.
- When the ~~coin~~ robot visits a cell with a coin it always picks up the coin.

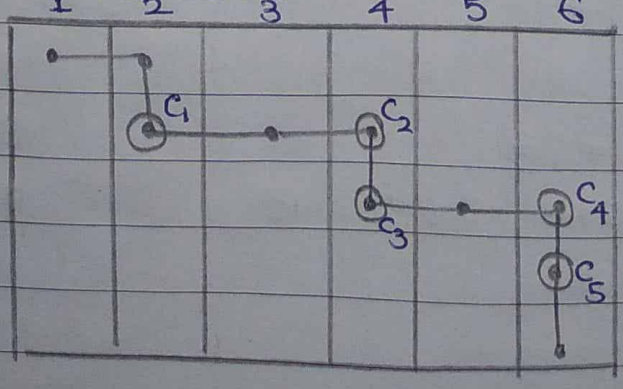
15/07/2024

Coins are represented with mxn matrix or arra

mxn 5X6


| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 1 | 2 | 2 | 2 |
| 3 | 0 | 1 | 1 | 3 | 3 | 4 |
| 4 | 0 | 1 | 2 | 3 | 3 | 5 |
| 5 | 1 | 1 | 2 | 3 | 4 | 5 |

Path should choosen in such a way that maximum coins are collected



Time Complexity:
At max. robot can travel m rows and n columns.

$T(n) = O(mn) = \Omega(mn) = \Theta(mn)$

## Branch and Bound:

- This is a problem solving technique, uses upper bound and least cost in each step.
- Every problem of branch and bound is solved with state space tree preferably.
- Every node of a state space tree is represent vertex of the given graph (No. of weights with respect to knapsack).
- Branch is preferably based on cost at every node.

Branch and Bound mainly consist of two main problems

1) Knapsack Problem
2) Travelling Salesman Problem

## Knapsack Problem:

- Knapsack is a maximization problem where knapsack is fit with max profit.
- Knapsack is solved using least cost method.

| | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|
| Profit | 10 | 10 | 12 | 18 | m = 15 |
| weight | 2 | 4 | 6 | 9 | n = 4 |

LC → Least Cost
BB → Branch Bound
u/UB → Upper Bound

$$u = \sum_{i=1}^{n} P_i x_i \leq m \text{ (capacity of knapsack)}$$

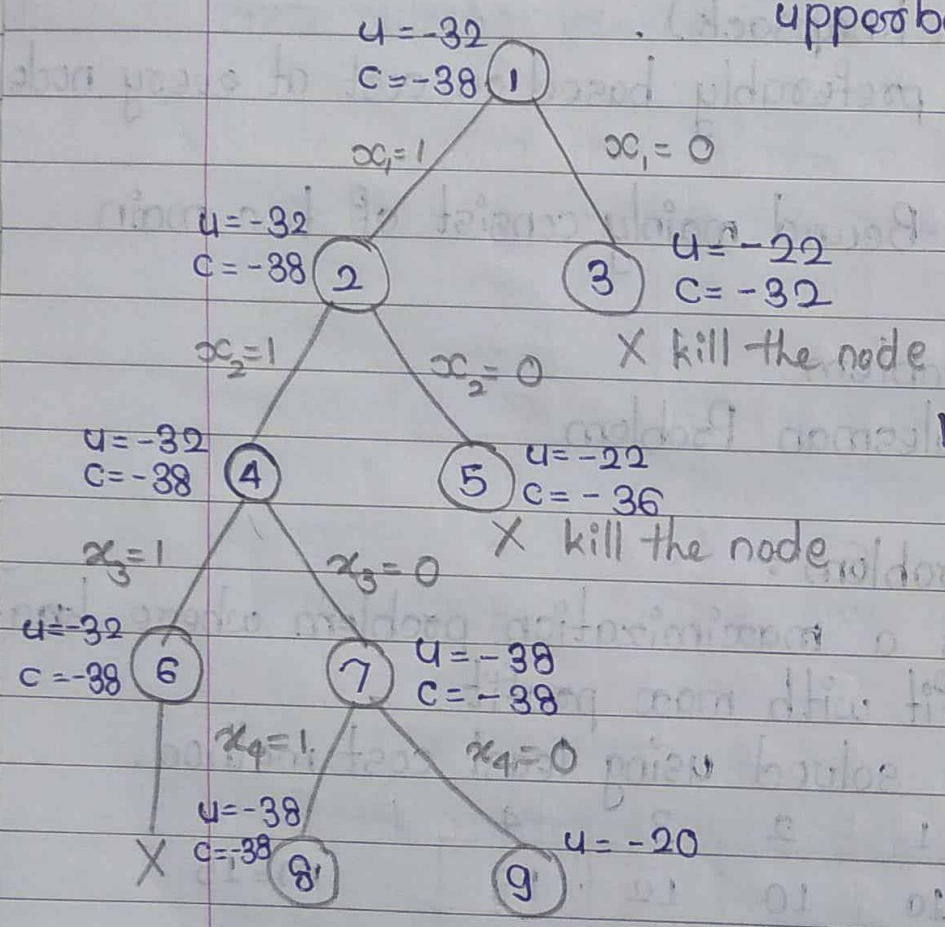
$c = \sum_{i=1}^{n} P_i x_i$ (with fraction)

$S = \{1, 0, 1, 0\}$ Format of the solution.

1 // Included 0 // Not included

Initially upperbound is assume as infinity.

The states space tree for knapsack problem is given below:




upperbound = ∞ -32 -38

$C = 10 + 10 + 12 + \frac{18}{9} \times 3$

$= 38$

$= 2 + 4 + 6$

$= 15 - 12 = 3$

Node: 3

$C = 10 + 10 + 12 + \frac{18}{9} \times 5$

$7 + 4 + 6$

$C = 32$

Node: 5

$C = 10 + 10 + 12 + \frac{18}{9} \times 7$

$2 + 4 + 6$

$C = 36$

$x = \{1, 1, 0, 1\}$

$x = 10 + 10 + 0 + 18$

$= 38$

Propit = $x = 38$

Weight = $2 + 4 + 0 + 9 = 15$

Node: 7

$C = 10 + 10 + 12 + 18$

$2 + 4 + 6 + 9$

Algorithm : Knapsack

Step 1 : Least count and upper bound is calculated for node 1.

Step 2 : Selection of the nodes based on least count upper bound.

Step 3 : Based on step 2 we complete the states space tree for all the nodes.

Step 4 : Where least adjacent node least count and UB is low we delete that node.

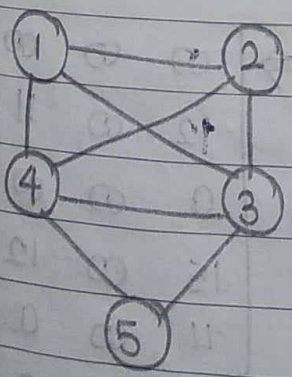
Time Complexity:

For every node we are considering weight selection or weight rejection

$T(n) = O(2^n) = \Omega(2^n) = \Theta(2^n)$

Travelling Salesman Problem:

- Travelling salesman need to find out the shortest route from source vertex to all other vertices. (Visited only once and return back to source city)
- It is a graph based problem. Consider the graph given below.




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | ∞ | 20 | 30 | 10 | 11 |
| 2 | 15 | ∞ | 16 | 4 | 2 |
| 3 | 3 | 5 | ∞ | 2 | 4 |
| 4 | 19 | 6 | 18 | ∞ | 3 |
| 5 | 16 | 4 | 7 | 16 | ∞ |

- Solving TSP using branch and bound, the first

step is finding reduced matrix, based on the reduced matrix we construct state space tree for all the vertices to find the shortest journey.

First we calculate reduced matrix:
In calculating reduced matrix, first we do row wise reduction, second one is column wise reduction.

* While finding the routes make 1st row full infinity and second column be infinity.

| | | | | | |
|---|---|---|---|---|---|
| ∞ | 20 10 | 30 20 17 | 10 0 | 11 1 | 10 |
| 15 13 12 | ∞ | 16 14 11 | 4 2 | 2 0 | 2 |
| 3 1 0 | 5 3 | ∞ | 2 0 | 4 2 | 2 |
| 19 16 15 | 6 3 | 18 15 12 | ∞ | 3 0 | 3 |
| 16 12 11 | 4 0 | 7 3 0 | 16 12 | ∞ | 4 |
| 1 | 0 | 3 | 0 | 0 \| 4 | 21 |

4+21 = 25

Reduced matrix = 25 (cost)

Reduced matrix:

| | | | | |
|---|---|---|---|---|
| ∞ | 10 | 17 | 0 | 1 |
| 12 | ∞ | 11 | 2 | 0 |
| 0 | 3 | ∞ | 0 | 2 |
| 15 | 3 | 12 | ∞ | 0 |
| 11 | 0 | 0 | 12 | ∞ |

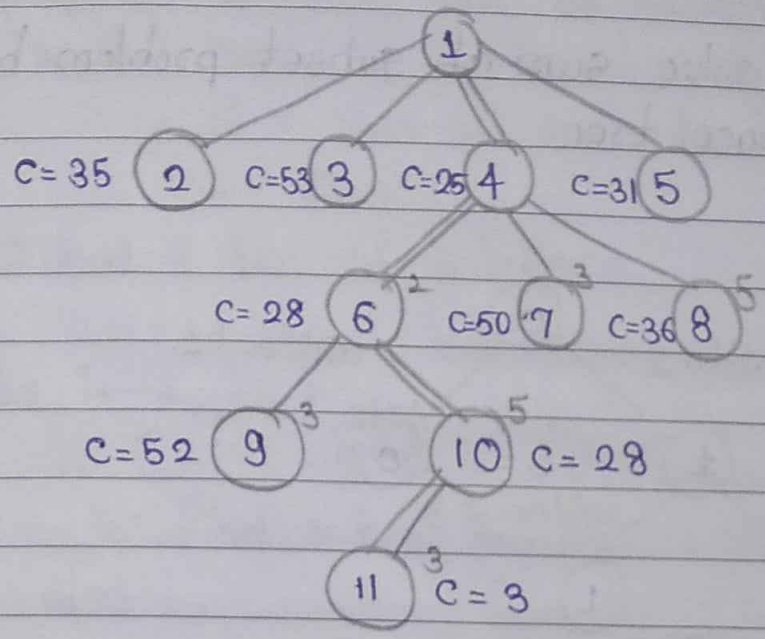
$\gamma = 25$ (rc)
↳ Notation

⇒

| | | | | | |
|---|---|---|---|---|---|
| ∞ | ∞ | ∞ | ∞ | ∞ | |
| 12 | ∞ | 11 | 2 | 0 | 0 |
| 0 | ∞ | ∞ | 0 | 2 | 0 |
| 15 | ∞ | 12 | ∞ | 0 | 0 |
| 11 | ∞ | 0 | 12 | ∞ | 0 |
| | 0 | 0 | 0 | 0 | |

$\hat{\gamma} = 0$
↳ New reduced cost

$c(1,2)+rc+\hat{r}$




1-4-2-5-3-1 as shortest route

Time Complexity:
$T(n) = O(2^n)$

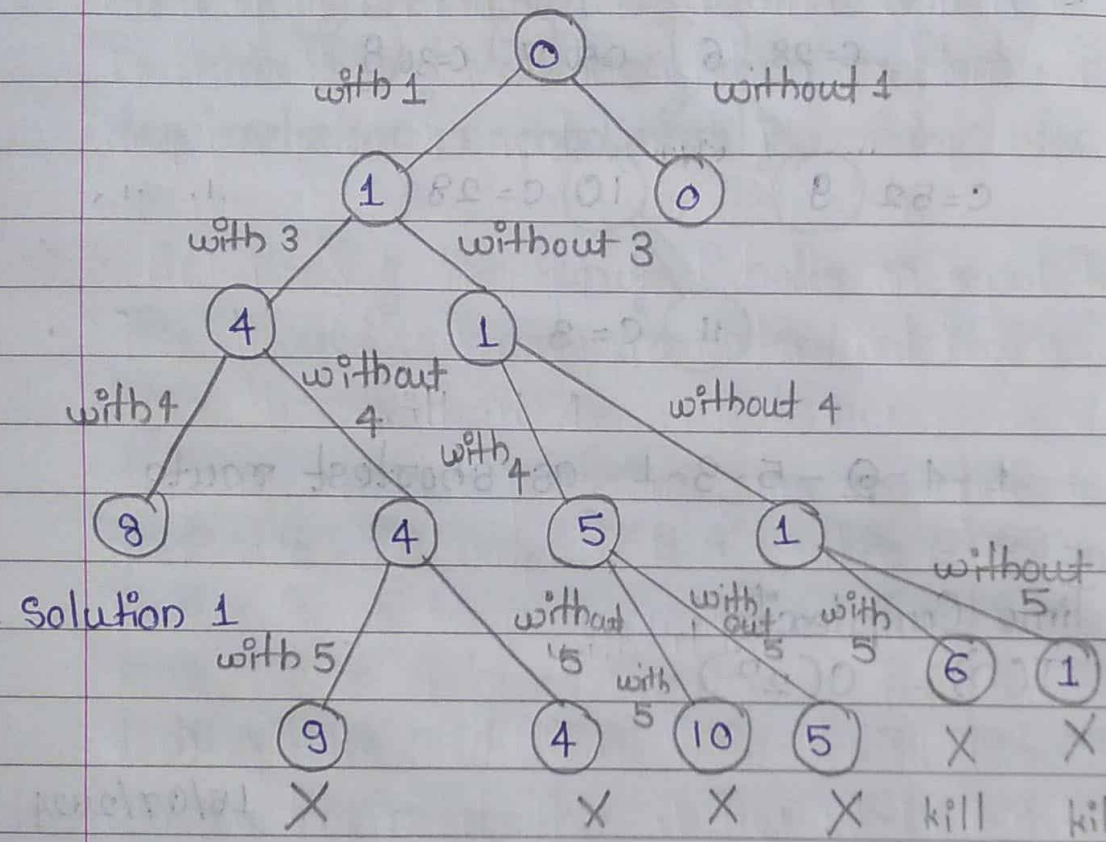
15/07/2024

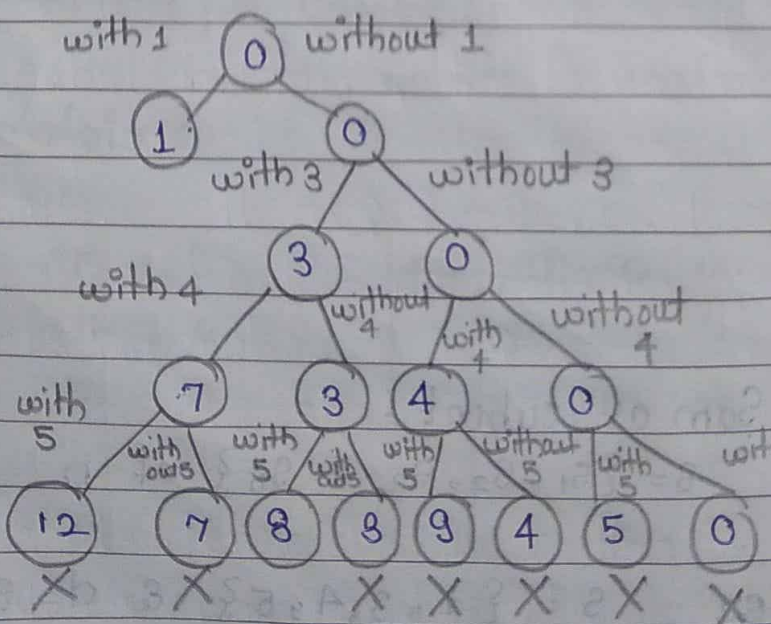
Sum of subset:
$S = \{s_1, s_2, s_3 \cdots s_n\}$ of n 'd'

eg: $S = \{1, 3, 4, 5\}$ & $d = 8$
subset 1 $\{1, 3, 4\}$ → solution '1' 1+3+4 → 8

solution 2 {5, 3} → solution '2' 3+5 → 8 d

We can solve sum of subset problem by using state space tree



Algorithm : Sum of Subsets ( s[], d)
// purpose : To find sum of subsets
// Input : set with different weights,
d → sum of subsets algorithm needs to find
// Output : No. of solutions.

```
for i ← 1 to k
    if ( wi*xi + Wk+1 <= d )
        find subset solution
    else
        kill the nodes
```

CW → Current Weight
NW → Next Weight

Time Complexity :
Sum of Subset problem is solved using state space tree.
$T(n) = 2^n$

16/07/2024

P, NP, NP- Completeness

Class P is a class of decision problem that can be solved in polynomial time. This class of problem is called as polynomial.

Examples for P class hamiltonian circuit problem

Some decision problems cannot be solved at all by any algorithm. Such problems are called undecidable as opposed to decidable problems that can be solved by an algorithm.

- Halting problem : given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

By way of contradiction, assume that A is an algo-rithm that solves the halting problem. That is, for any program P and input I,

$$A(P,I) = \begin{cases} 1, & \text{if program P halts on input I;} \\ 0, & \text{if program P does not halt on input I.} \end{cases}$$

we can consider program p as an input to itself and use the output of algorithm A for pair (P,P) to construct a program Q as follows:

$$Q(P) = \begin{cases} \text{halts,} & \text{if } A(P,P)=0, \text{ i.e., if program P does not halt} \\ \text{does not halts} & \text{if } A(P,P)=1, \text{ i.e, if program P halts on input p} \end{cases}$$

Then on substituting Q for P, we obtain

$$Q(Q) = \begin{cases} \text{halts,} & \text{if } A(Q,Q)=0, \text{ i.e., if program Q does not halts on input Q;} \\ \text{does not halts,} & \text{if } A(Q,Q)=1, \text{ i.e., if program Q halts on input Q.} \end{cases}$$

This is a contradiction because neither of the two outcomes for program Q is possible, which comple-tes the proof.

Class NP the class of decision problems that can be solved by nondeterministic polynomial algorithm

A decision problem D is said to be NP-Complete if
1) If belongs to class NP
2) Every problem in NP is polynomially reduced to D.

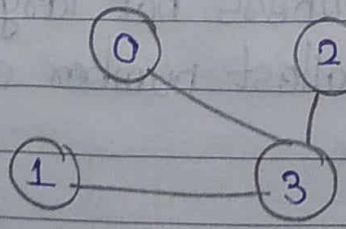
Explain Notation of NP-Complete Problem with the neat diagram.

Explain proving NP-completeness by reduction with neat diagram.

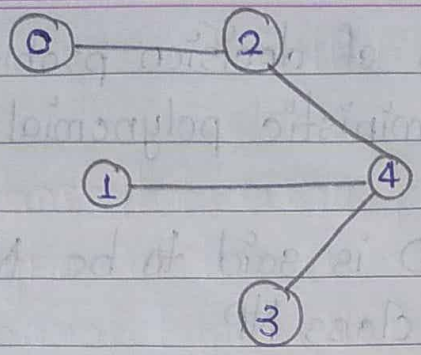
Vertex Cover Problem
- It is also called as NP-Complete problem.
- It is defined as vertex cover of undirected gra
-ph G(V, E)
V → vertices
E → Edges
- Importance is given for edges (all the edges to b covered with the minimum vertices)

eg: 

{ }



{1, 2, 3} Minimum Vertice




{0, 2, 4} Minimum Vertic

- Hamiltonian circuit problem: a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

- Travelling salesman problem: Find the shortest tour through n cities with known positive integer distances between them.

- Knapsack problem: Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

- Partition problem: Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.

- Bin-packing problem: Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.

- Graph-coloring problem: For a given graph, find its chromatic number, which is the smallest

number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.

- Integer linear programming problem: Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

Time Complexity:
The problem of the existence of a cycle that tra-verses all the edges of the graph exactly once

$T(n) = O(n^2)$

- A nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution. (In other words, we require a nondeterministic algorithm to be capa-ble of "guessing" a solution at least once and to be able to verify its validity. And, of course, we do not want it to ever output a yes answer on an instance for which ther answer should be no.). Finally, a nondeterministic algorithm is said to be nondeterministic polynomial if the time effi-ciency of its verification stage is polynomial.

Most decision problem are in NP. First of all, this class includes all the problems in P:

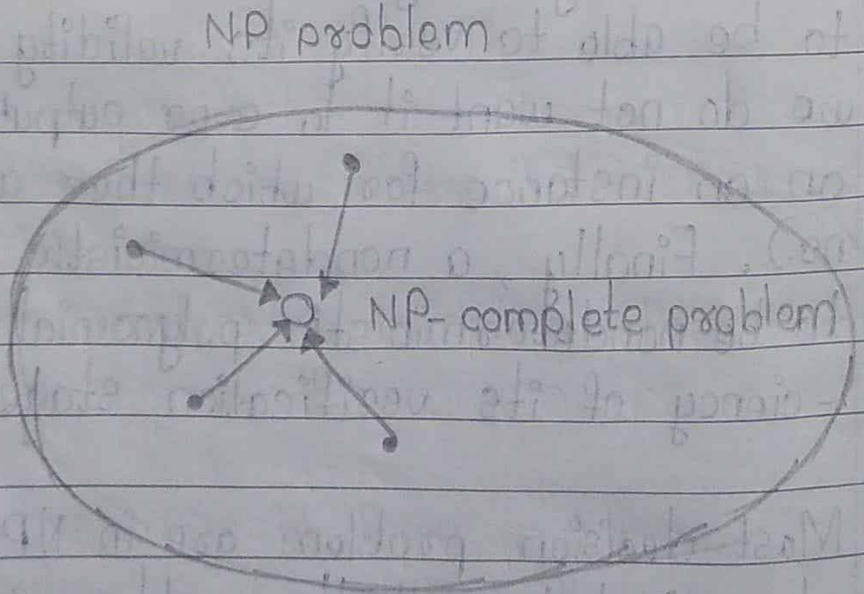
$P \subseteq NP$

It means that every problem that can be solved in polynomial time (P) can also be verified in polynomial time (NP). This inclusion shows that all problems solvable by deterministic algorithms (P) are also within the set of problems solvable by nondeterministic polynomial time algorithm (NP)

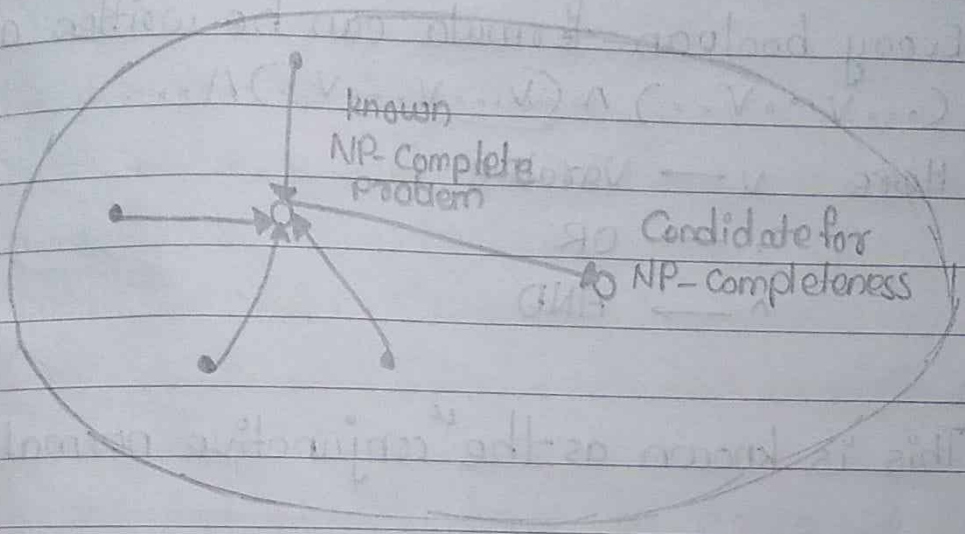
$P \stackrel{?}{=} NP$

The P?= NP refers to the open question in computer science about whether every problem that can be ~~so~~ verified in polynomial time (NP) can also be solved in polynomial time (P). This question, known as the P vs. NP problem, remains one of the most important and unresolved questions in theoretical computer science.

Notation of an NP-Complete problem.



# Proving NP-completeness by reduction



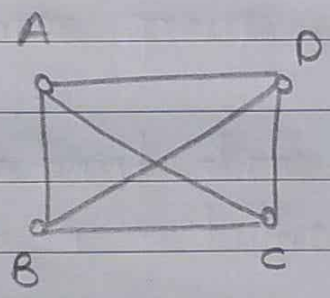
# Reducing SAT to Clique

SAT ⟶ CLIQUE

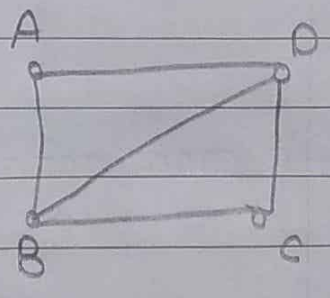
Every boolean formula can be written as
$(\ldots \vee \ldots \vee \ldots) \wedge (\ldots \vee \ldots \vee \ldots) \wedge \ldots$
Here $\vee$ ⟶ Varable (clause)
. ⟶ OR
$\wedge$ ⟶ AND

This is known as the "conjunctive normal form"

CLIQUE: A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. The size of clique is the number of vertices in it




This is a clique




This is not a clique
A & C are not connected

The clique problem is an optimization problem of finding a clique of maximum size in a graph.

Decision version of clique problem:
The formal way of defining it

CLIQUE = {(G, k): G is a graph containing a clique of size k}

## 3CNF to CLIQUE

The clique problem is NP-Complete

Step 1: Clique ∈ NP

If we are given a certificate, we need to able to verify it in polynomial time. Given a certificate, we can check if it correct by verifying it for each pair (u, v) in the certificate, there is an edge.

Step 2: We next have to prove 3-CNF-SAT ≤p Clique

The reduction a algorithm begins with an instance of 3-CNF-SAT

let $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a boolean formula in 3-CNF with k clauses.

For $C_r \rightarrow L_1^r, L_2^r, L_3^r$ are its three distinct literals.

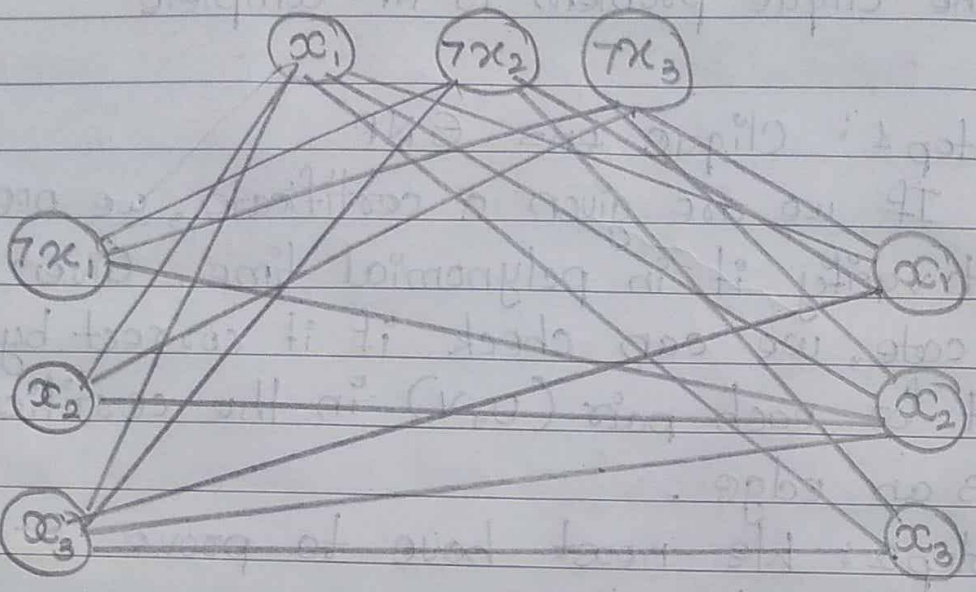
We will construct a graph G such that $\phi$ is satisfiable if and only if G has a clique of size k. If and only if implies we have to show the reduction both ways.

First we need to construct the graph. G=(V, E) as follows. For each clause $C_r = (L_1^r \vee L_2^r \vee L_3^r)$ in $\phi$ we place a triple of vertices $v_1^r, v_2^r, v_3^r$ as follows.

Let us consider an example

$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

$c_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$




$c_2 = \neg x_1 \vee x_2 \vee x_3$

$c_3 = x_1 \vee x_2 \vee x_3$

Note that there is no edge between $x_i$ and $\neg x_i$ if they are in separate groups.

There are no edges between vertices in the same group.

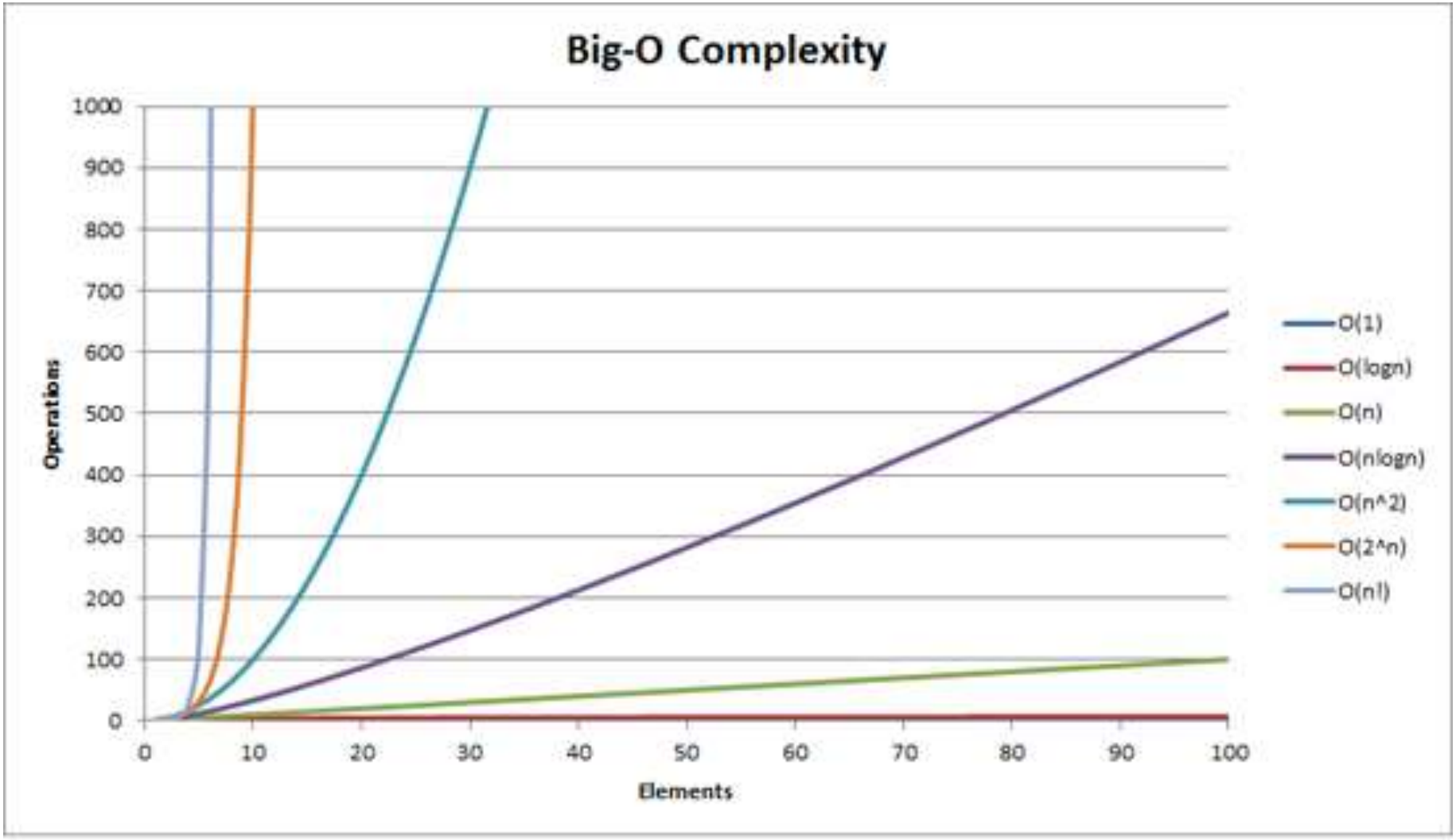
We must show this transformation from $\phi$ to $G$ is a reduction.

1. First suppose $\phi$ has a satisfying assignment.
   → Each clause $C_r$ contains atleast one literal $L_i^r = 1$
   → Picking one "true" literal from each clause yields a set $V'$ of $k$ vertices. We claim that $V'$ is a clique.
   → For any two vertices $v_i^r, v_j^s \in V'$ $r \neq s$ because we only pick one vertex from each

group
→ Literals $L_i^r$ and $L_j^s$ corresponding $V_i^r$ and $V_j^s$ both map to 1 and thus $L_i^r$ and $L_j^s$ cannot be complements.
→ Thus by construction of G an edge exists between $V_i^r$ and $V_j^s$ in V'. Thus the vertices in V' form a clique of size $|V'| = k$

2. Suppose G has a clique V' of size k
→ No edges in G connect vertices in the same triple.
→ So V' contains exactly on vertex per triple.
→ We can assign 1 to each literal $L_i^r$ such that $V_i^r \in V'$. This is possible because G contains no edges between a literal and its complement, and so we can assign $L_i^r = 1$
→ Each clause $C_r$ is satisfied and hence $\phi$ is satisfied.

This completes the reduction an we have shown the transformation holds in both direction.


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

# Polynomial (P) Problems

- Are solvable in polynomial time
  - Polynomial time: O(n2), O(n3), O(1), O(n lg n)
- Are solvable in $O(n^k)$, where k is some constant.

- Most of the algorithms we have covered are in P

# 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

# Polynomial time solvable problems

- **Sorting Algorithms**
  - Merge Sort: O(n log n)
  - Quick Sort: Average-case O(n log n)
  - Heap Sort: O(n log n)
  - Bubble Sort: O(n^2)
- **Graph Algorithms**
  - Dijkstra's Algorithm: O(V^2) (using an adjacency matrix), O((V + E) log V) (using a priority queue and adjacency list)
  - Floyd-Warshall Algorithm: O(V^3)
  - Kruskal's Algorithm for Minimum Spanning Tree: O(E log E)
  - Prim's Algorithm for Minimum Spanning Tree: O(V^2) or O((V + E) log V) with a priority queue
- **Matrix Operations**

# Polynomial Time Verification - Hamiltonian cycles

- Determining whether a directed graph has a Hamiltonian cycle does not have a polynomial time algorithm (yet!)
- However if someone was to give you a sequence of vertices, determining whether or not that sequence forms a Hamiltonian cycle can be done in polynomial time
- Therefore Hamiltonian cycles are in NP

# NP

- **NP is not the same as non-polynomial complexity/running time. NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**
- NP means verifiable in polynomial time

# Nondeterministic Polynomial (NP) Problems

- NP is the set of decision problems *solvable* in polynomial time by a nondeterministic Turing machine.
- NP is the set of decision problems *verifiable* in polynomial time by a deterministic Turing machine.

# NP problems

- Graph theory has these fascinating(annoying?) pairs of problems
  - Shortest path algorithms?
  - Longest path is NP complete
  - Eulerian tours (visit every vertex but cover every edge only once, even degree etc). Solvable in polynomial time!
  - Hamiltonian tours (visit every vertex, no vertices can be repeated). NP complete

# P and NP Summary

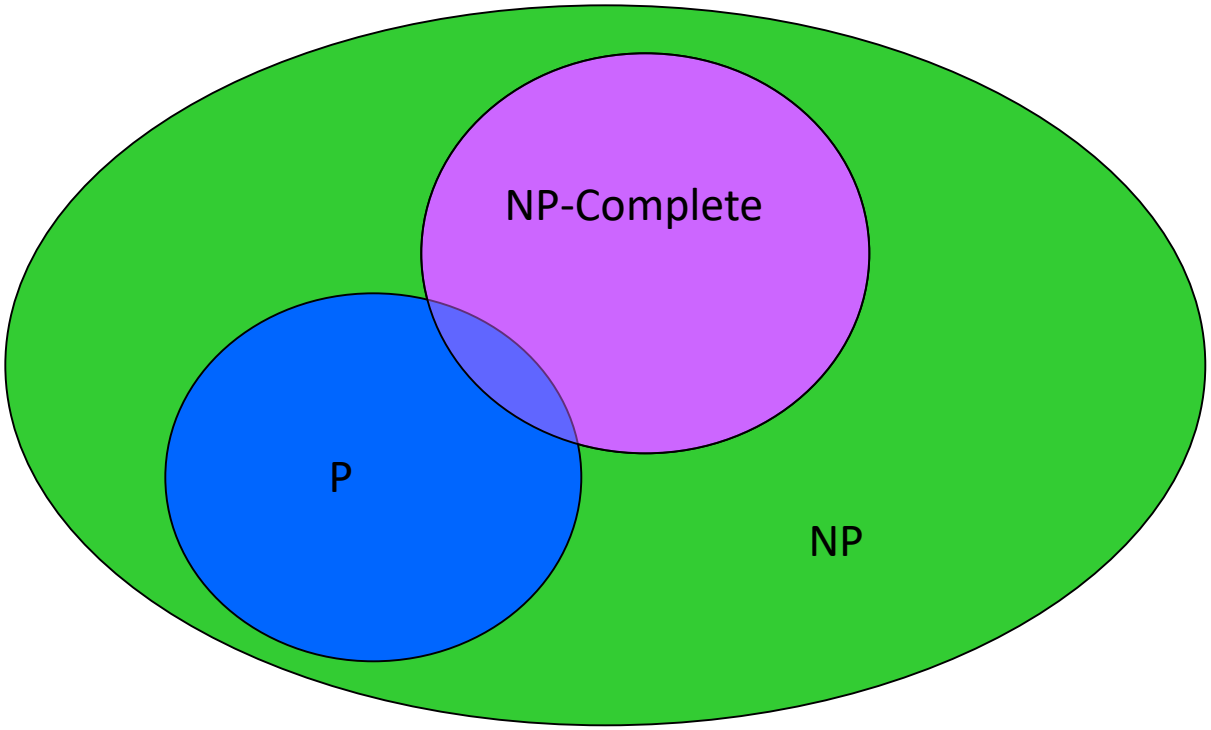
- **P** = set of problems that can be solved in polynomial time
  - Examples: Fractional Knapsack, …
- **NP** = set of problems for which a solution can be verified in polynomial time
  - Examples: Fractional Knapsack,…, TSP, CNF SAT, 3-CNF SAT
- Clearly $\mathbf{P} \subseteq \mathbf{NP}$
- Open question: Does $\mathbf{P} = \mathbf{NP}$?
  - $P \neq NP$

| Sr. No | P class Problem | NP class Problem |
|---|---|---|
| 1. | P problems are a set of problems that can be solved in polynomial time by deterministic algorithms. | NP problems are problems that can be solved in nondeterministic polynomial time. |
| 2. | P Problems can be solved and verified in polynomial time. | The solution to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be verified in polynomial time. |
| 3. | P problems are a subset of NP problems. | NP Problems are a superset of P problems. |
| 4. | All P problems are deterministic in nature. | All the NP problems are non-deterministic in nature. |
| 5. | It takes polynomial time to solve a problem like n, n^2, n*logn, etc. | It takes non-deterministic polynomial time to quickly check a problem. |
| 6. | The solution to P class problems is easy to find. | The solution to NP class problems is hard to find. |
| 7. | **Examples of P problems are:** Selection sort, Linear Search | **Examples of NP problems are** the Travelling salesman problem and the knapsack problem. |

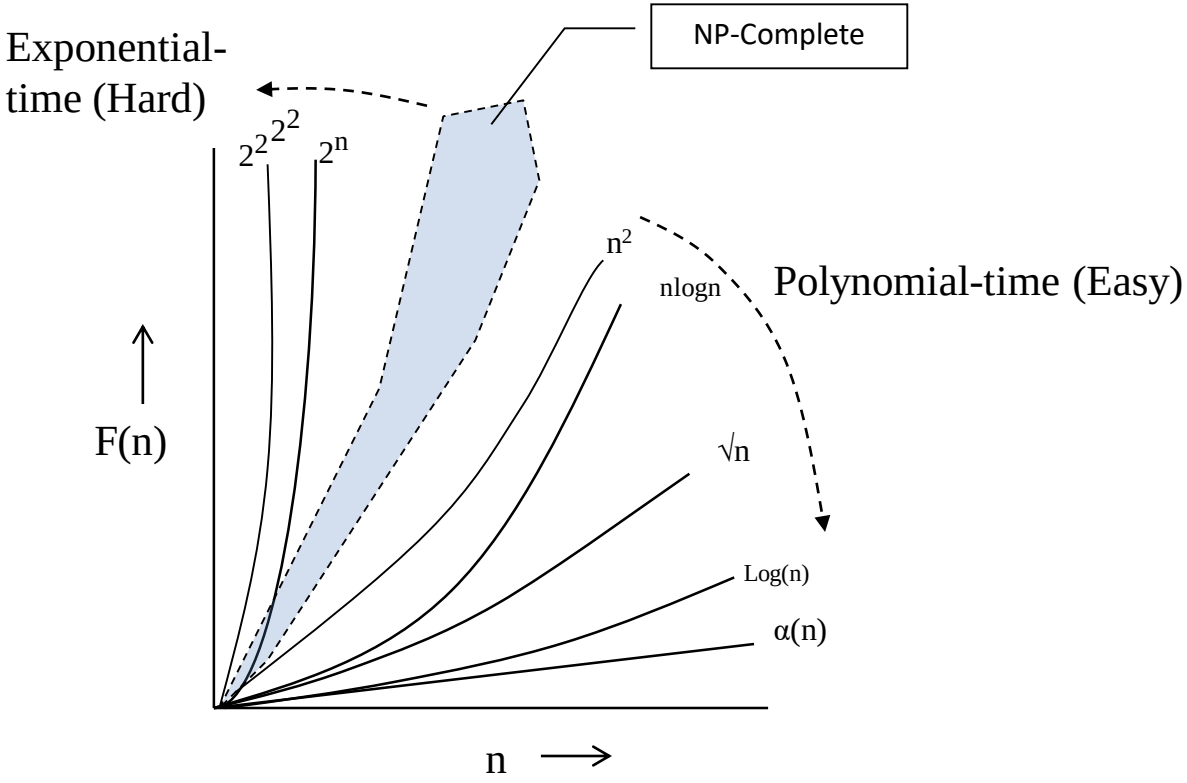
# NP Complete

- Any given problem X acts as NP-Complete when there exists an NP problem Y- so that the problem Y gets reducible to the problem X in a polynomial line.
- For solving an NP-Complete Problem, the given problem must exist in both NP-Hard and NP Problems.
- This type of problem is always a Decision problem (exclusively).
- A few examples of NP-Complete Problems are the determination of the Hamiltonian cycle in a graph, the determination of the satisfaction level of a Boolean formula

# NP-Completeness

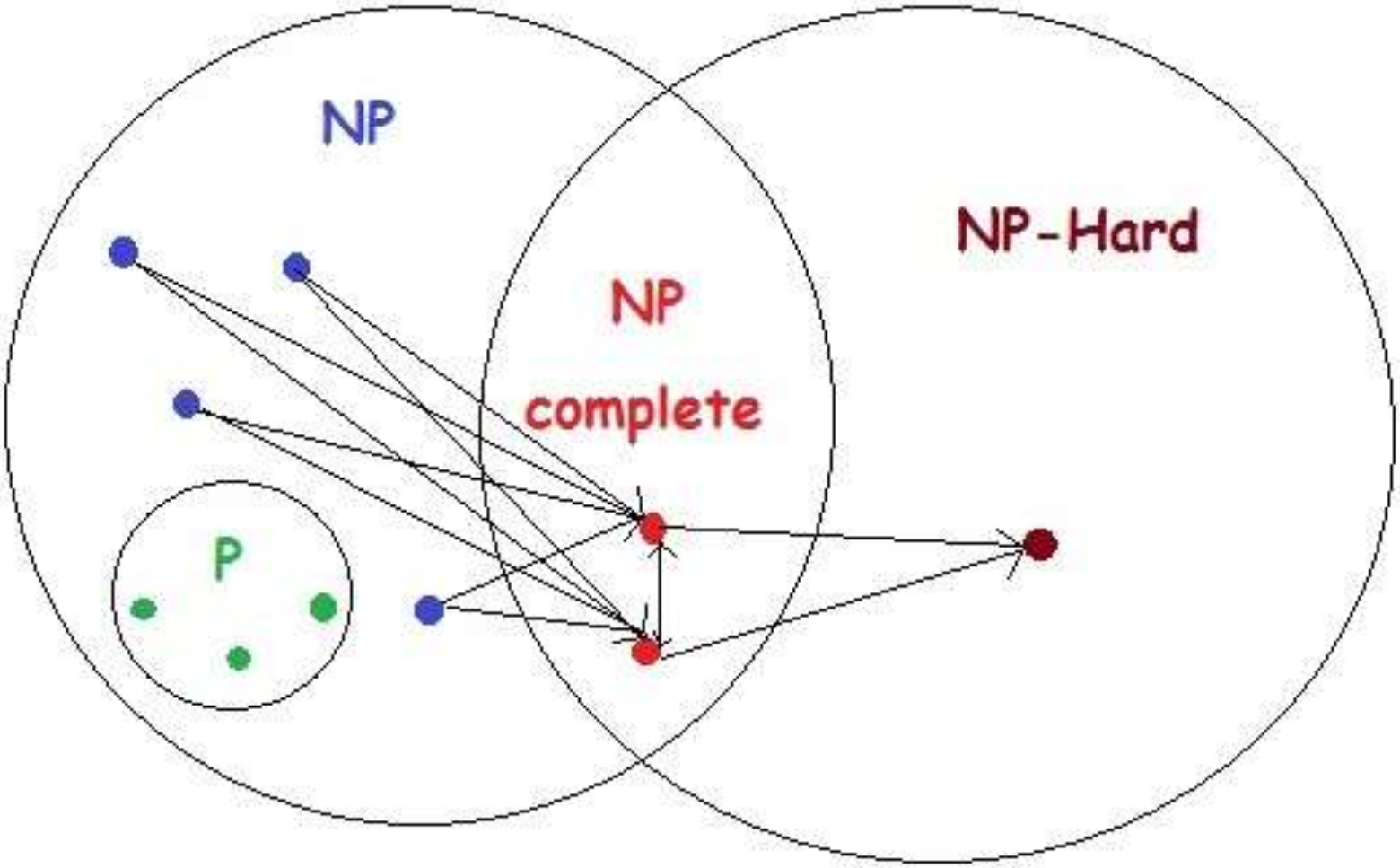


# Exponential Time Algorithms



# NP Hard

- One can only solve an NP-Hard Problem X only if an NP-Complete Problem Y exists. It then becomes reducible to problem X in a polynomial time.
- The NP-Hard Problem does not have to exist in the NP for anyone to solve it.
- This type of problem need not be a Decision problem.
- Circuit-satisfactory, Vertex cover, Halting problems, etc., are a few examples of NP-Hard Problems.


NP
NP-Hard
NP
complete
P

# NP Completeness and Reducibility

- One way that sometimes works for solving a problem is to recast it as a different problem.
- Think of a problem **Q as being reducible to another problem Q'** if any instance of Q can be recast as an instance of Q' , and the solution to the instance of Q' provides a solution to the instance of Q.

# Reducibility

- Is a linear equation reducible to a quadratic equation?
  - Sure! Let coefficient of the square term be 0

# Example

. For example, the problem of solving linear equations in an indeterminate $x$ reduces to the problem of solving quadratic equations. Given a linear-equation instance $ax + b = 0$ (with solution $x = -b/a$), you can transform it to the quadratic equation $ax^2 + bx + 0 = 0$. This quadratic equation has the solutions $x = (-b \pm \sqrt{b^2 - 4ac})/2a$, where $c = 0$, so that $\sqrt{b^2 - 4ac} = b$. The solutions are then $x = (-b + b)/2a = 0$ and $x = (-b - b)/2a = -b/a$, thereby providing a solution to $ax + b = 0$. Thus, if a problem $Q$ reduces to another problem $Q'$, then $Q$ is, in a sense, "no harder to solve" than $Q'$.

Example of a polynomial-time reduction:

We will reduce the

3CNF-satisfiability problem

to the

CLIQUE problem

# SAT

- A boolean formula is ***satisfiable*** if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.
- CNF – Conjunctive Normal Form AND ing of clauses of ORs

$$(x_0 \vee x_2) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee x_1 \vee \neg x_2)$$

# 2-CNF SAT

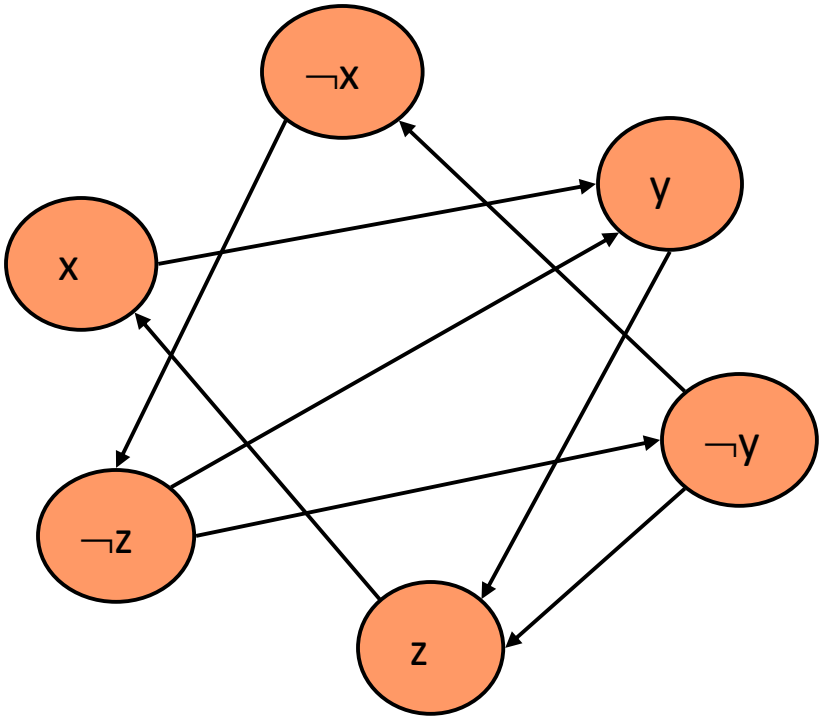
- Each or operation has two arguments that are either variables or negation of variables
- The problem in 2 CNF SAT is to find true/false(0 or 1) assignments to the variables in order to make the entire formula true.

$$(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$$

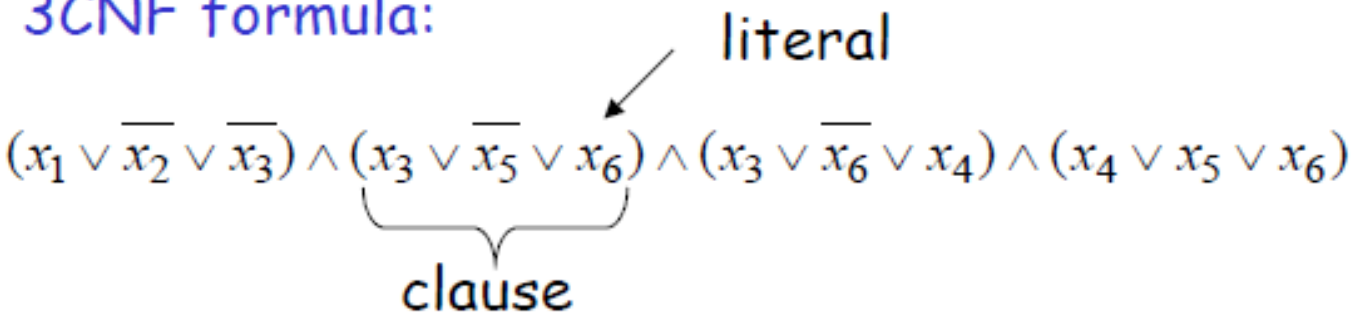
- Any of the OR clauses can be converted to implication clauses

# 2-SAT is in P

- Create the implication graph



3CNF formula:




Each clause has three literals

**Language:**

3CNF-SAT = { $w$ : $w$ is a satisfiable 3CNF formula}

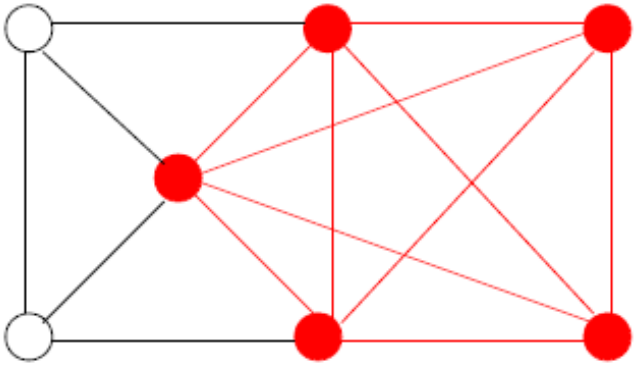
# CLIQUE problem

- A clique in an undirected graph G=(V,E) is a subset of vertices V’ $\subseteq$ V of vertices, each pair of which is connected by an edge in E
- A Clique is a **complete subgraph of G**

# Clique Problem

- Clique Problem: It is a subgraph of a graph which contains all possible edges between each pair of vertices in the subgraph

A 5-clique in graph $G$



**Language:**

$$\text{CLIQUE} = \{< G, k >: \text{ graph } G \text{ contains a } k\text{-clique}\}$$

# Reducing 3CNF SAT to CLIQUE

- Given – A boolean formula in 3 CNF SAT
- Goal – Produce a graph (in polynomial time) such that

$$\text{Satisfiabiality} \Leftrightarrow \text{Clique of a certain size}$$

- We will construct a graph where satisfying formula with k clauses is equivalent to finding a k vertex clique.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with $k$ clauses. For $r = 1, 2, \ldots, k$, each clause $C_r$ contains exactly three distinct literals: $l_1^r$, $l_2^r$, and $l_3^r$. We will construct a graph $G$ such that $\phi$ is satisfiable if and only if $G$ contains a clique of size $k$.

We construct the undirected graph $G = (V, E)$ as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in $\phi$, place a triple of vertices $v_1^r$, $v_2^r$, and $v_3^r$ into $V$. Add edge $(v_i^r, v_j^s)$ into $E$ if both of the following hold:

- $v_i^r$ and $v_j^s$ are in different triples, that is, $r \neq s$, and
- their corresponding literals are ***consistent***, that is, $l_i^r$ is not the negation of $l_j^s$.

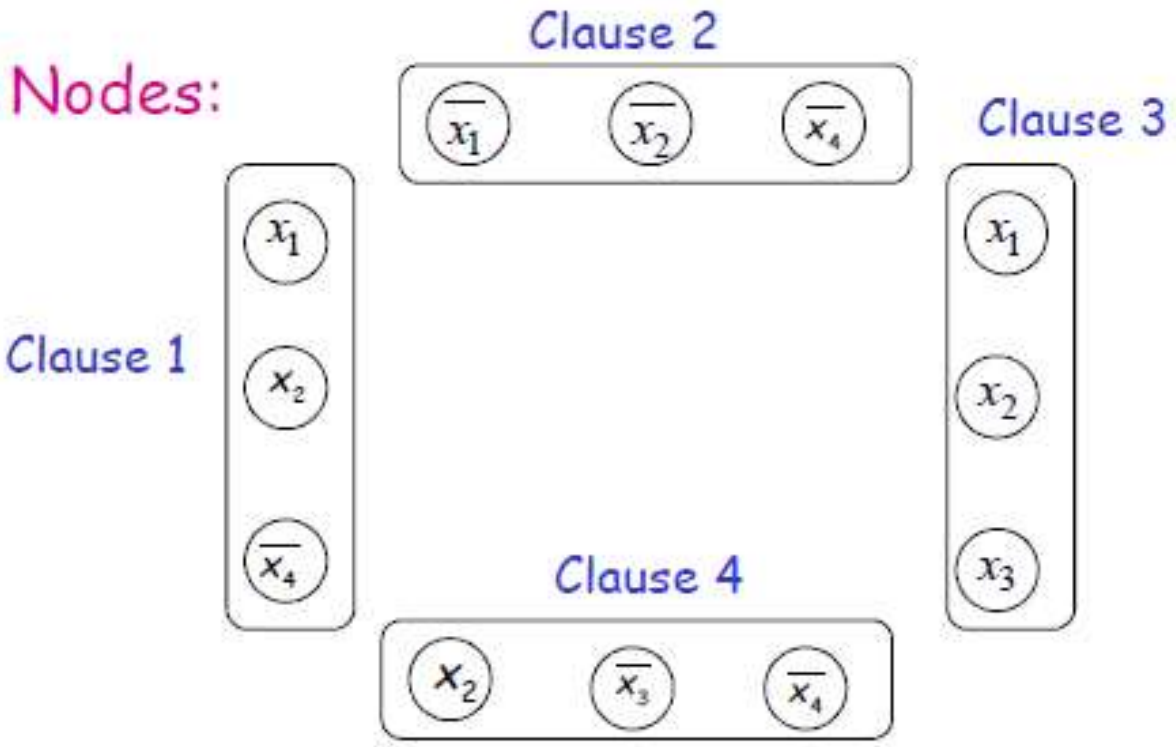
We can build this graph from $\phi$ in polynomial time.

Transform formula to graph.

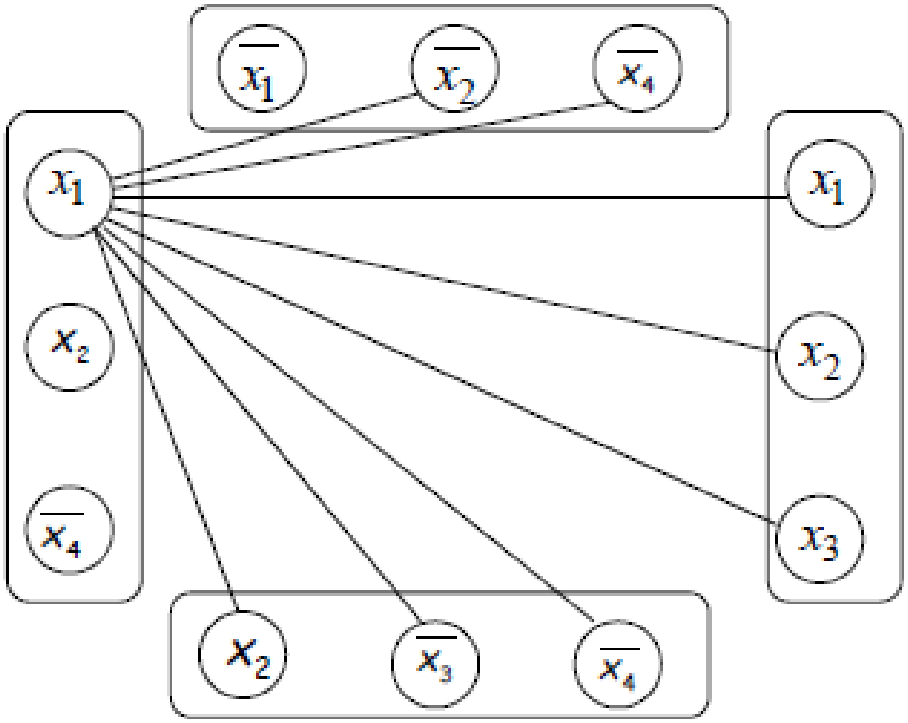
Example:

$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$

Create Nodes:

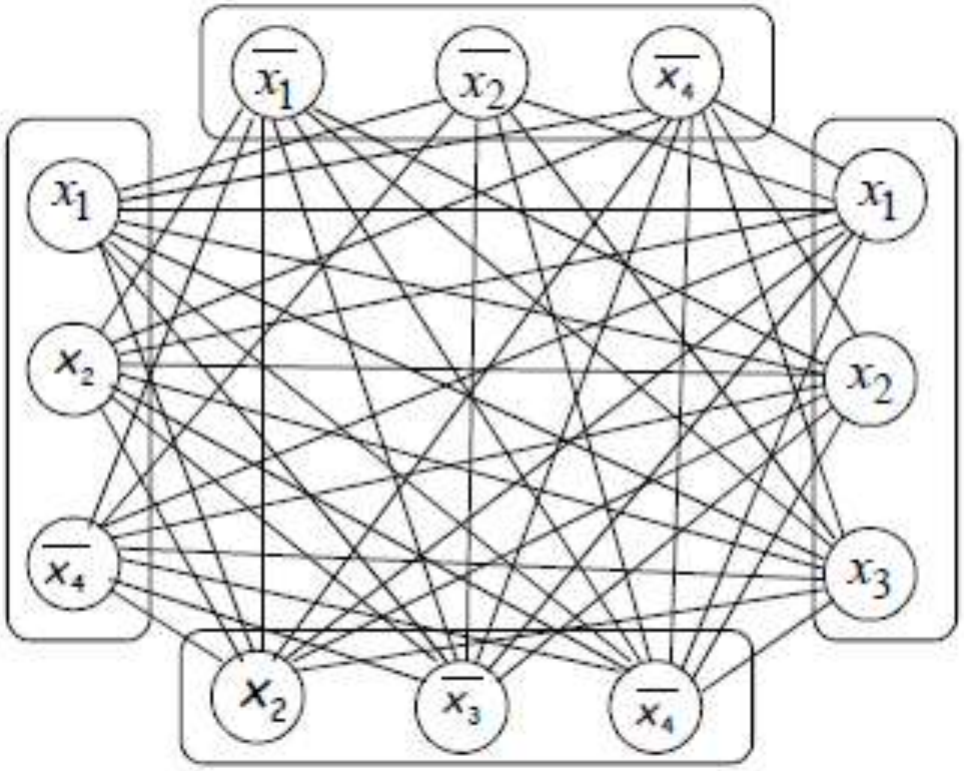


$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$$




Add link from a literal $\xi$ to a literal in every other clause, except the complement $\overline{\xi}$

$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$$




Resulting Graph

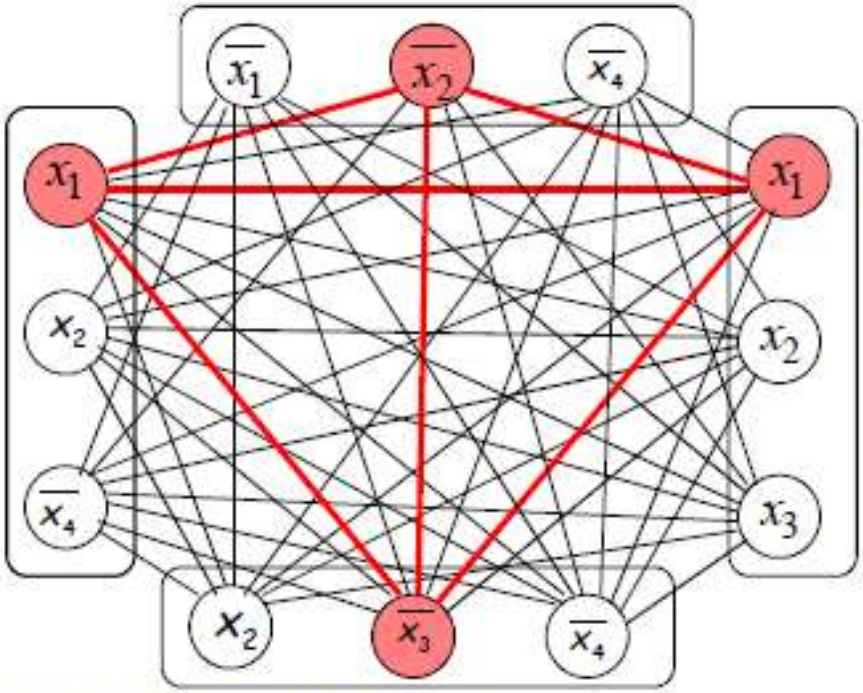
$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) = 1$$

$x_1 = 1$

$x_2 = 0$

$x_3 = 0$

$x_4 = 1$




The formula is satisfied if and only if the Graph has a 4-clique

The objective is to find a clique of size 4, where 4 is the number of clauses.

End of Proof

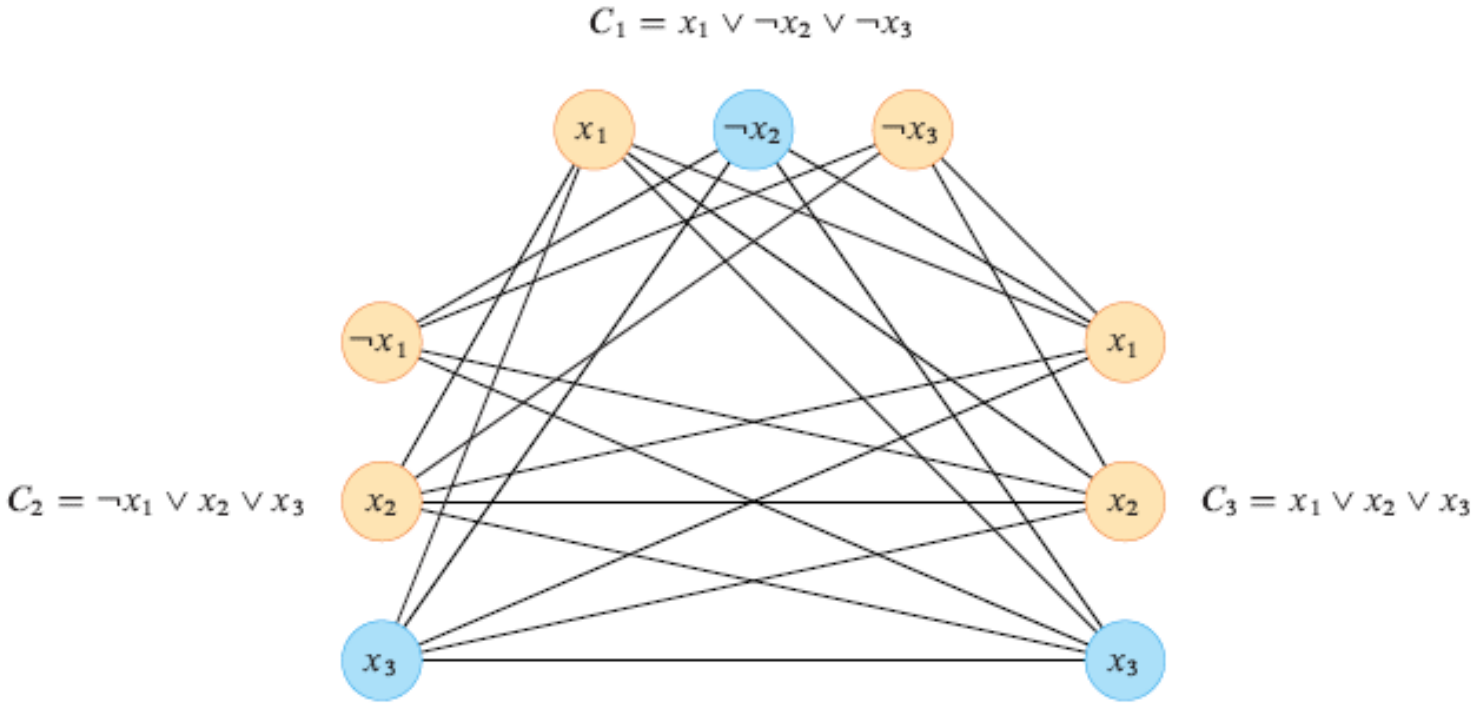



**Figure 34.14** The graph $G$ derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and $x_1$ set to either 0 or 1. This assignment satisfies $C_1$ with $\neg x_2$, and it satisfies $C_2$ and $C_3$ with $x_3$, corresponding to the clique with blue vertices.

# Vertex Cover Problem

- A ***vertex cover*** of an undirected graph G=(V,E) is a subset V' $\subseteq$ V such that if (u,v) $\in$ E, the

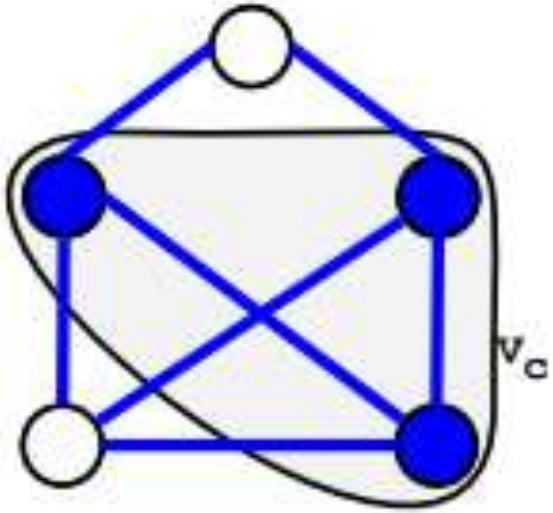
  u $\in$ V' or v $\in$ V' or (both)
- That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E.
- The ***size*** of a vertex cover is the number of vertices in it

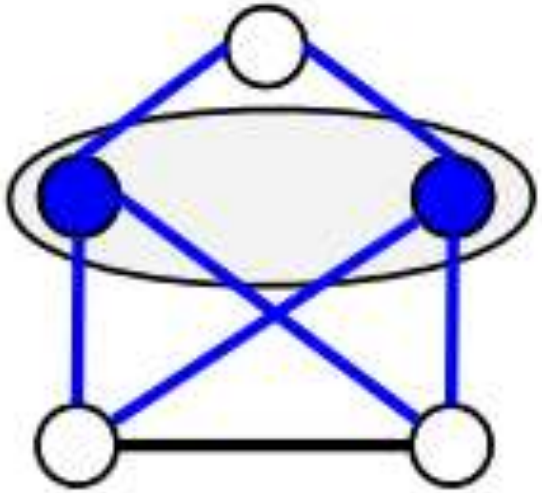
# Vertex cover problem

- We’re typically interested in finding the minimum sized vertex cover
- To show vertex cover is NP-complete
- What problem should we try to reduce to it
- It sounds like the ‘reverse’ of CLIQUE
- Reduction is done from CLIQUE to vertex cover

# Vertex Cover

- A *vertex cover* of a graph $G = (V, E)$ is a $V_C \subseteq V$ such that every $(a, b) \in E$ is incident to at least a $u \in V_C$.



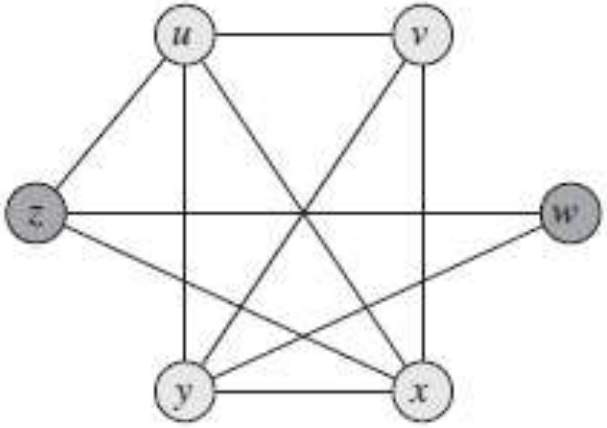
$\longrightarrow$ Vertices in $V_C$ 'cover' all the edges of $G$.

- The VERTEX COVER (VC) decision problem:
  Does $G$ have a vertex cover of size $k$?

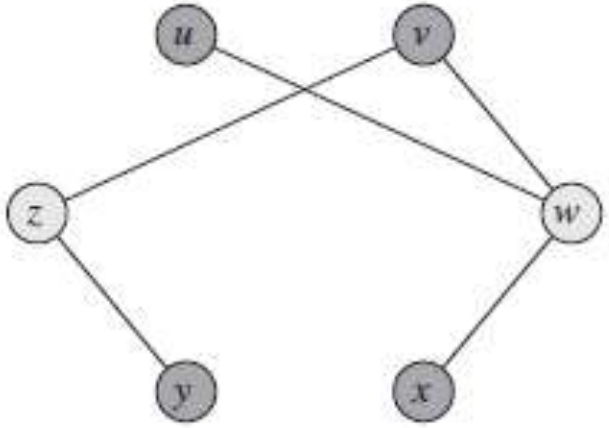
To prove that the vertex-cover problem is NP-hard, we reduce from the clique problem, showing that CLIQUE $\leq_P$ VERTEX-COVER. This reduction relies on the notion of the complement of a graph. Given an undirected graph $G = (V, E)$, we define the ***complement*** of $G$ as a graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$. In other words, $\overline{G}$ is the graph containing exactly those edges that are not in $G$. Figure : shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

G

G' = Complement Of G




(a)

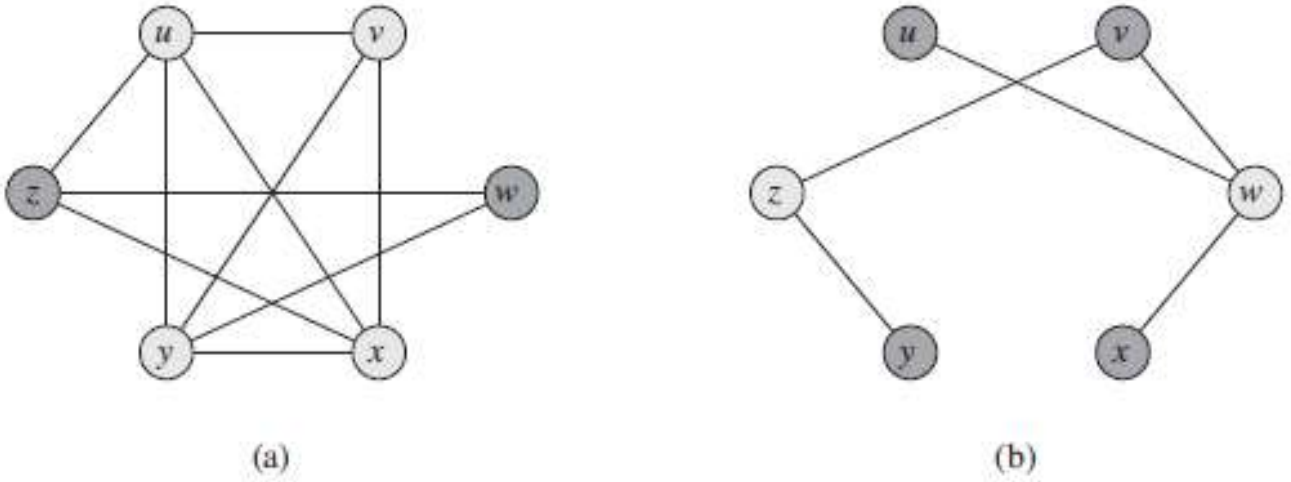



(b)

The reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem and computes the complement $\overline{G}$ in polynomial time. The output of the reduction algorithm is the instance $\langle \overline{G}, |V| - k \rangle$ of the vertex-cover problem. To complete the proof, we show that this transformation is indeed a reduction: the graph $G$ contains a clique of size $k$ if and only if the graph $\overline{G}$ has a vertex cover of size $|V| - k$.

Suppose that $G$ contains a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\overline{G}$. Let $(u, v)$ be any edge in $\overline{E}$. Then, $(u, v) \notin E$, which implies that at least one of $u$ or $v$ does not belong to $V'$, since every pair of vertices in $V'$ is connected by an edge of $E$. Equivalently, at least one of $u$ or $v$ belongs to $V - V'$, which means that edge $(u, v)$ is covered by $V - V'$. Since $(u, v)$ was chosen arbitrarily from $\overline{E}$, every edge of $\overline{E}$ is covered by a vertex in $V - V'$. Hence the set $V - V'$, which has size $|V| - k$, forms a vertex cover for $\overline{G}$.

Conversely, suppose that $\overline{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then for all $u, v \in V$, if $(u, v) \in \overline{E}$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words, $V - V'$ is a clique, and it has size $|V| - |V'| = k$. ■

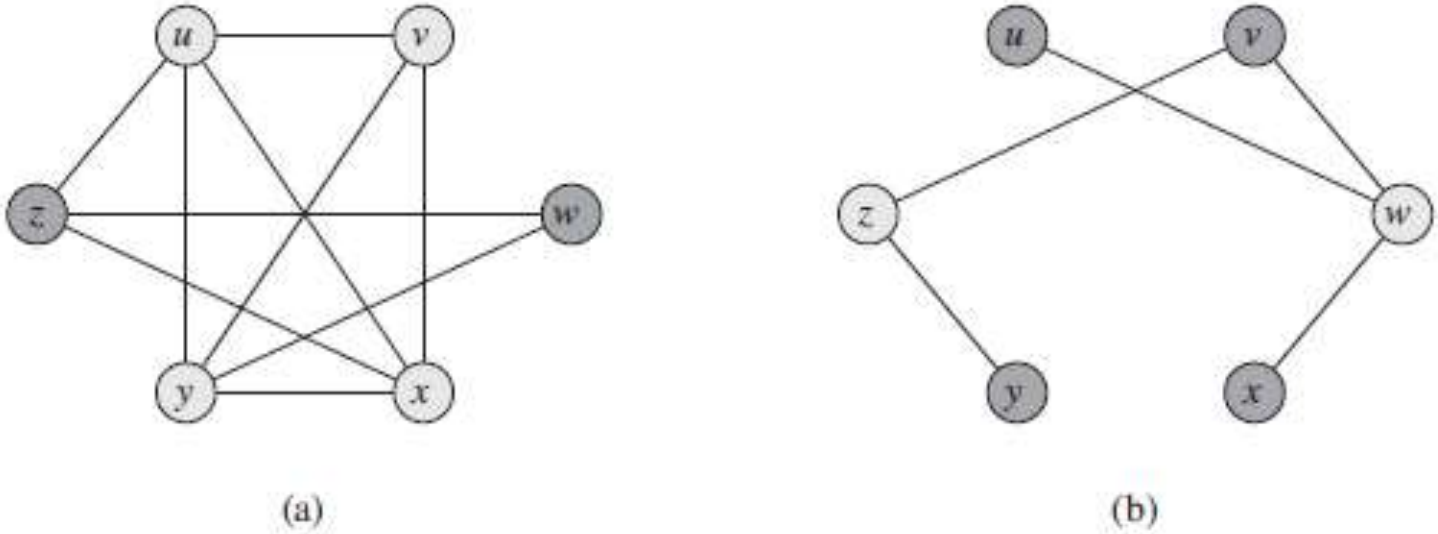



The original graph has a u,v,x,y CLIQUE. That is a clique of size 4

The complement graph has a vertex cover of size 6 (number of vertices) – 4 (clique size). z,w is one such vertex cover.

G

G' = Complement Of G




Clique of size k in G exists iff a vertex cover of size |V| - k exists in G' where G' is the complement graph (vertices that had an edge between then in G do not have one in G' and vice versa)

$$\text{CLIQUE} \leq_p \text{VC}$$